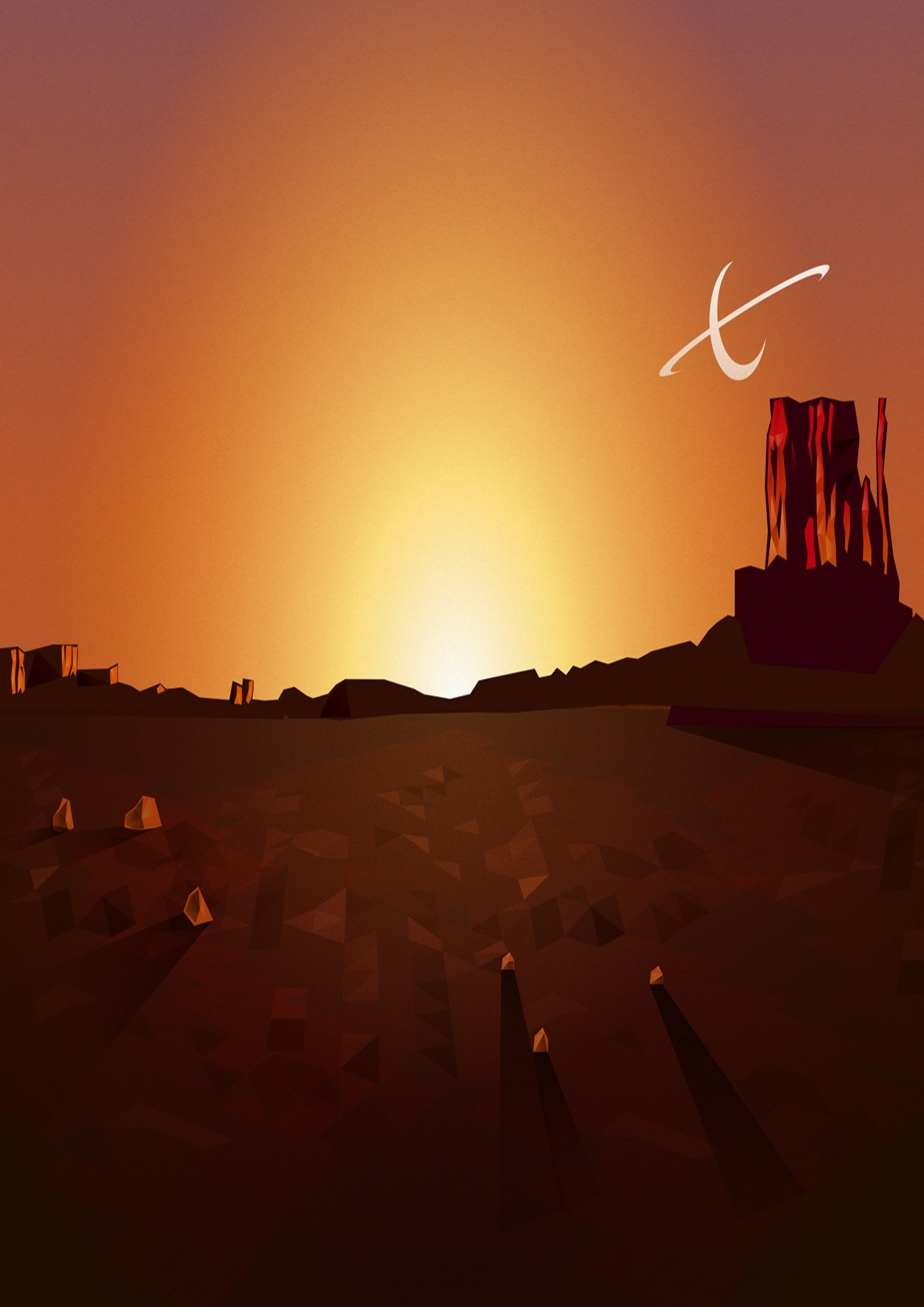


Table of Contents

Ethereum Frontier Guide

Welcome to the Frontier!

The Frontier is the first live release of the Ethereum network. As such you are entering uncharted territory and you are invited to test the grounds and explore. There is a lot of danger, there may still be undiscovered traps, there may be ravaging bands of pirates waiting to attack you, but there also is vast room for opportunities.

In order to navigate the Frontier, you’ll need to use the command line. If you are not comfortable using it, we strongly advise you to step back, watch from a distance for a while and wait until more user friendly tools are made available. Remember, there are no safety nets and for everything you do here, you are mostly on your own.

What is Ethereum?

Ethereum, like any advanced system, will mean different things to different people. As you read this section, some bits may not resonate with you or even make sense. That is fine, just skip to the next paragraph and hopefully that one will be more enlightening. If you reach the end of this section and still feel confused, then jump on a forum and start asking questions.

A World Computer

> "It is very possible that ... one machine would suffice to solve all the problems ... of the whole [world]" - Sir Charles Darwin, 1946*

In a technical sense, Ethereum is a "world computer". Harking back to the days of the mainframe, and probably about as fast, Ethereum can be viewed as a single computer that the whole world can use. It notionally has only a single processor (no multi-threading or parallel execution), but as much memory as required. Anybody can upload programs to the Ethereum World Computer and anybody can request that a program that has been uploaded be executed. This does not mean that anyone can ask any program to do anything; on the contrary: the author of the program can specify that requests from anyone but themselves be ignored, for example. Also, in a very strong sense, every program has its own permanent storage that persists between executions. Furthermore, as long as it is in demand, the Ethereum World Computer will always be there: it can't be shut down or turned off.

You may ask, "why would anyone use such a system?" and again there are many reasons. The main reason is because it makes what you want to do cheaper and easier. This statement needs to be broken down somewhat, which is what the following paragraphs explore.

An Internet Service Platform

> "Technology gives us the facilities that lessen the barriers of time and distance - the telegraph and cable, the telephone, radio, and the rest." - Emily Greene Balch

In a more practical sense, Ethereum is an internet service platform for guaranteed computation. More than that, as a platform, it provides a set of integral features which are very useful to the developer:

- user authentication, via seamless integration of cryptographic signatures
- fully customizable payment logic; easily create your own payment system without any

reliance on third parties
- 100% ddos resistant up-time, guaranteed by being a fully decentralized blockchain-based platform
- no-fuss storage: forget about having to set up secure databases; Ethereum gives you as much storage as you want
- ultimate interoperability: everything in the Ethereum ecosystem can trivially interact with everything else, from reputation to custom currencies
- server free zone: your whole application can be deployed on the blockchain meaning no need for setting up or maintaining servers; let your users pay for the cost of their using your service.

Over the last twenty years in particular, we have seen an acceleration in the development of services and infrastructure to make the overhead of working as a team or running businesses simpler and less expensive, mainly thanks to the internet. The likes of eBay, Drivy and Airbnb have made setting up a shop, car rental company or hotel much easier. These are platforms that allow people to realize their idea quickly, as long as the service they want to provide fits the template offered by the platform. Outside Ethereum, it is very costly to create a new platform if those that already exist do not fit your needs. Ethereum can be seen as a platform for platforms: it allows people to easily create the infrastructure to make it easy to set up new services on the internet. Furthermore, any infrastructure created on Ethereum sits alongside everyone else's creations, and so can interact with those other platforms in a guaranteed and seamless manner. Importantly, because there is not a company or indeed any entity in charge of or controlling Ethereum, the cost of running the infrastructure doesn't have to include any profit margin, so we are likely to see lower costs.

With the coming of the Mix IDE and the Mist browser, the functionality of Ethereum as a deployment platform for internet services will become more clear. The take-home message from this section, however, is that Ethereum is poised to disrupt industries as diverse as finance and supply chains.

Opt-in Social Contracts

> "This is an era of organization" - Theodore Roosevelt, 1912**

At a more abstract level, it is a facility for enabling smart organization, in the sense of groups of entities working together for a particular cause. In the simplest scenario, we have two people working together to achieve a trade. Ultimately, Ethereum could be used to run countries. Somewhere in between there are groups of people wanting to organize baby-sitting circles, film-making collectives, discussion groups, communal houses, etc, and they all have to decide the rules with which they will operate together. Arguably, the harder task is how to implement and enforce the rules, especially given the variety of characters, abilities and motivations that the human race provides. In other words, how do you stop Jo hogging the camera equipment, or Dave never doing his turn of the childcare? Ethereum provides a platform on which rules can be defined and, to an ever increasing degree as technology improves, see enforced. For example, the camera could be listening to the blockchain and only record if the film-making collective's DApp approves Jo's access code.

Crowdfunding is a key example in the advancement organizational tools. It provides a couple of really important functions: a way for individuals to work together for a particular cause (in this case giving a wedge of cash to a person or group) and a mechanism for individuals to interact meaningfully with potentially large companies (such as a games studio). The first follows from the previous paragraph, but the second point is impressive also, because in general individuals can only communicate with large organizations on the organization's terms, which often default to "ignore"; in the same way you ignore the bacteria on your skin. As it stands, you might be disgruntled by the crowdfunding recipient taking your money and spending it in a wholey inappropriate and therefore inefficient manner. However, it's hard to get the operating company to act on your behalf against the recipient. After all, the crowdfunding service provider is likely to be a large company and there isn't a universal mechanism with which you can meaningfully communicate with it if it does not want you to. Ethereum can help by allowing you to define post-funding milestones or conditions to stage the payment of the total amount raised, and then enforce those conditions for you. As times goes on, we will get more creative in the ways in which Ethereum can interact with the real world and the ability of Ethereum to check that milestones have been completed will extend beyond the obvious such as "30% of the crowd that funded the project (by value) have voted that the milestone has been passed".

Part of the Decentralization Revolution

> "No matter who you vote for, the government always gets in" - The Bonzo Dog Doo-Dah Band, 1992

Philosophically, it is the next step in re-decentralizing the internet. A decentralized system is one that anyone can unilaterally join and participate in, one in which all participants contribute to the running and maintanence, and one in which any participant can unilaterally leave and when they do, the system continues regardless. In a decentralized system, there is no entity that can prevent participation or arbitrarily censor the content or usage. The internet was designed to be decentralized, but the way we use it has become increasingly centralized, to the point where censorship and exclusion are accepted and expected. Blockchain technology, introduced by Satoshi Nakamoto with the proof-of-concept implementation of a simple value transfer system known as bitcoin, represents the best digital system we have (after the internet itself) for administering multi-user interactions without any need for centralized coordination or oversight. Effectively, a decentralized system is its own authority for enforcing the rules (e.g. 'you can only spend your money once' in bitcoin, or 'whatever rule you programmed in to your smart contract' in Ethereum), so participants can be confident that the rules they expect to be enforced will be, without any danger of corruption, bribery, nepotism, political bias, exclusion, arbitrary exceptions, human oversight or absence of staff.

Ethereum allows people to safely interact trustlessly by entering into neutrally enforceable agreements in a completely peer-to-peer fashion. Now, it must be remembered that Ethereum can only enforce within its own digital limits; Ethereum does not remove the need for an external authority for adjudication over disputes outside its realm---"the other party punched me in the face after putting in the Ethereum contract that he wouldn't" is non-sense, but rules exist elsewhere to cover this---but what Ethereum does do is allow us to push the boundary on what the digital realm can cover.

Conclusion

Gavin Wood has distilled the description of Ethereum to being a collection of non-localized singleton programmable data structures. What this means will depend on where you are coming from, but wherever you are, it's probably going to be better with Ethereum.

Notes

*Copeland, Jack (2006). Colossus: The Secrets of Bletchley Park's Codebreaking Computers. Oxford University Press. p.109

Note that T.J.Watson, the former head of IBM, almost certainly never said, "I think there is a world market for maybe five computers", although for a time (late 1940s, early 1950s) it was indeed the case that there was, in the United States, a market for about five computers.

**Presidential speech, Milwaukee, Wisconsin.

FRONTIER IS COMING

- Frontier launch final steps
- Etherchain API look out for block #1028201
- Olympic network stats

Ethereum Frontier Release

Introduction

Frontier is the first in a series of releases that punctuate the roadmap for the development of Ethereum. Frontier will be followed by ‘Homestead’, ‘Metropolis’ and ‘Serenity’ throughout the coming year, each adding new features and improving the user friendliness and security of the platform.

Ethereum is special and different from other software projects in that its release also involves launching a live network. After a year and a half of development the Proof of Concept series completed 9 cycles. The 10th iteration resulted in the Olympic testnet, which gradually led to the Release Candidate client for Frontier.

The Ethereum network goes live when the clients consent on the **genesis block** and start mining transactions on it. The genesis block will reference an initial system state where all the accounts set up by the presale exist with the correct amount of pre-issued ether allocated. Initially, the network will be in a "thawing" state allowing only blocks to be mined, but not transactions to be processed. This allows for users to have a break-in period to connect to the network while also building up its security.

In conjunction with the Frontier launch several exchanges will likely start enabling trade of Ether, which will provide necessary liquidity to the marketplace, allowing users and miners to transfer their holdings to other users requiring more or less Ether. As opposed to an earlier strategy, there is no plan to remove any contracts from the blockchain or otherwise alter the network to carry balances over to Homestead. In other words, the state in Homestead will be a direct and unmodified continuation of the state in Frontier.

Mining reward is the full amount of 5 ether per block (as opposed to our earlier proposal of a reduced amount). Mining rewards are discussed in detail here

Safety warnings

- **You are responsible for your own computer security.** If your machine is

compromised you **will** lose your ether, access to any contracts and maybe more.
- **You are responsible for your own actions.** If you mess something up or break any laws while using this software, it's your fault, and your fault only.
- **You are responsible for your own karma.** Don't be a jerk and respect others.

WARNING: Before you interact with the ethereum Frontier network, make sure you read the documentation and understand the caveats and risks. Please read the legal disclaimer

Components released

The focus of Frontier is the Go implementation of an ethereum full node, with a command line interface codenamed "Geth".

By installing and running `geth`, you can take part in the ethereum live network, mine ether on the blockchain, transfer funds between addresses, create contracts and send transactions.

WARNING: before you use `geth` or interact with the ethereum Frontier live network, make sure you read the documentation and fully understand the caveats and risks.

Apart from `geth`, the Go CLI, the Frontier release contains the following components:

- `web3.js` library implementing the JavaScript API for Dapps to conveniently interact with an ethereum node
- `solc` a standalone solidity compiler. You only need this if you want to use your Dapp or console to compile solidity code.
- `ethminer` a standalone miner for openCL GPU mining
- `netstat` a network monitoring GUI allows you to add your node to the http://stats.ethdev.com page

The actual launch process

Ethereum is not something that's centrally 'launched', but instead emerges from consensus. Users will have to voluntarily download and run a specific version of the software, then manually generate and load the Genesis block to join the official project's network.

Once Frontier has been installed on their machines, users will need to generate the Genesis block themselves, then load it into their Frontier clients. A script and instructions on how to do this will be provided as part of the new Ethereum website, as well as our various wikis.

We're often asked how existing users will switch from the test network to the live network: it will be done through a switch at the Geth console (--networkId). By default the new build will aim to connect to the live network, to switch back to the test network you'll simply indicate a network id of ‘0’.

Bugs, Issues and Complications

The work on the Frontier software is far from over. Expect weekly updates which will give you access to better, more stable clients. Many of the planned Frontier gotchas (which included a chain reset at Homestead, limiting mining rewards to 10%, and centralized checkpointing) were deemed unnecessary. However, there are still big differences between Frontier and Homestead. In Frontier, we’re going to have issues, we’re going to have updates, and there will be bugs – users are taking their chances when using the software. There will be big (BIG) warning messages before developers are able to even install it. In Frontier, documentation is limited, and the tools provided require advanced technical skills.

The Canary Contracts

The Canary contracts are simple switches holding a value equal to 0 or 1. Each contract is controlled by a different member of the Eth/Dev team and will be updated to ‘1’ if the internal Frontier Disaster Recovery Team flags up a consensus issue, such as a fork.

Within each Frontier client, a check is made after each block against 4 contracts. If two out of four of these contracts have a value switched from 0 to 1, mining stops and a message urging the user to update their client is displayed. This is to prevent “fire and forget” miners from preventing a chain upgrade.

This process is centralized and will only run for the duration of Frontier. It helps preventing the risk of a prolonged period (24h+) of outage.

Stats, Status and Badblock websites

You probably are already familiar with our network stats monitor, https://stats.ethdev.com/. It gives a quick overview of the health of the network, block resolution time and Gas statistics. Remember that participation in the stats page is voluntary, and nodes have to add themselves before they appear on the panel. See details on network monitoring

In addition to the stats page, we will have a status page at https://status.ethereum.org/ (no link as the site is not live yet) which will gives a concise overview of any issue that might be affecting Frontier. Use it as your first port of call if you think something might not be right.

Finally, if any of the clients receive an invalid block, they will refuse to process it send it to the bad block website (AKA ‘Sentinel’). This could mean a bug, or something more serious, such as a fork. Either way, this process will alert our developers to potential issues on the network. The website itself is public and available at https://badblocks.ethdev.com (currently operating on the testnet).

A Clean Testnet

During the last couple of months, the Ethereum test network was pushed to its limits in order to test scalability and block propagation times. As part of this test we encouraged users to spam the network with transactions, contract creation code and call to contracts, at times reaching over 25 transactions per second. This has led the test network chain to grow to a rather unwieldy size, making it difficult for new users to catch up. For this reason, and shortly after the Frontier release, there will be a new test network following the same rules as Frontier.

Olympic rewards distribution

During the Olympic phase there were a number of rewards for various achievements including mining prowess. These rewards will not be part of the Frontier Genesis block, but instead will be handed out by a Foundation bot during the weeks following the release.

Resources:

- Frontier launch final steps
- Frontier is coming blogpost by Stephan Tual announcing the launch.
- The frontier website
- Original announcement of the release scheme by Vinay Gupta
- Follow-up blogpost
- Least Authority audit blogpost with links to the audit report,
- Deja Vu audit blogpost
- Olympic. Frontier prerelease, Vitalik's blogpost detailing olympic rewards.

Disclaimer

Safety caveats

Security warnings

- **You are responsible for your own computer security.** If your machine is compromised you **will** lose your ether, access to any contracts and maybe more.
- **You are responsible for your own actions.** If you mess something up or break any laws while using this software, it's your fault, and your fault only.
- **You are responsible for your own karma.** Don't be a jerk and respect others.
- This software is open source under a GNU Lesser General Public License license.

Legal warning: Disclaimer of Liabilites and Warranties

Short version

- **The user expressly knows and agrees that the user is using the ethereum platform at the user's sole risk.**
- **The user represents that the user has an adequate understanding of the risks, usage and intricacies of cryptographic tokens and blockchain-based open source software, eth platform and eth.**
- **The user acknowledges and agrees that, to the fullest extent permitted by any applicable law, the disclaimers of liability contained herein apply to any and all damages or injury whatsoever caused by or related to risks of, use of, or inability to use, eth or the ethereum platform under any cause or action whatsoever of any kind in any jurisdiction, including, without limitation, actions for breach of warranty, breach of contract or tort (including negligence) and that neither stiftung ethereum nor the ethereum team shall be liable for any indirect, incidental, special, exemplary or consequential damages, including for loss of profits, goodwill or data.**
- **Some jurisdictions do not allow the exclusion of certain warranties or the limitation or exclusion of liability for certain types of damages. therefore, some of the above limitations in this section may not apply to a user. In particular, nothing**

in these terms shall affect the statutory rights of any user or exclude injury arising from any willful misconduct or fraud of stiftung ethereum.

Long Version: Terms and Conditions

The following Terms and Conditions (“Terms”) govern the use of the Ethereum open source software platform (“Ethereum Platform”). Prior to any use of the Ethereum Platform, the User confirms to understand and expressly agrees to all of the Terms. All capitalized terms in this agreement will be given the same effect and meaning as in the Terms. The group of developers and other personnel that is now, or will be, employed by, or contracted with, Stiftung Ethereum (“Stiftung Ethereum”) is termed the “Ethereum Team.” The Platform will be developed by persons and entities who support Ethereum, including both volunteers and developers who are paid by nonprofit entities interested in supporting the Ethereum Platform.

The user acknowledges the following serious risks to any use the Ethereum Platform and ETH and expressly agrees not to hold liable Ethereum Stiftung or Ethereum Team should any of these risks occur:

Risk of Regulatory Actions in One or More Jurisdictions

The Ethereum Platform and ETH could be impacted by one or more regulatory inquiries or regulatory action, which could impede or limit the ability of Stiftung Ethereum to continue to develop the Ethereum Platform, or which could impeded or limit the ability of User to use Ethereum Platform or ETH.

Risk of Alternative, Unofficial Ethereum Networks

It is possible that alternative Ethereum-based networks could be established, which utilize the same open source source code and open source protocol underlying the Ethereum Platform. The Ethereum network may compete with these alternative Ethereum-based networks, which could potentially negatively impact the Ethereum Platform and ETH.

Risk of Insufficient Interest in the Ethereum Platform or Distributed Applications

It is possible that the Ethereum Platform will not be used by a large number of external businesses, individuals, and other organizations and that there will be limited public interest in the creation and development of distributed applications. Such a lack of interest could impact the development of the Ethereum Platform and potential uses of ETH. It cannot predict the success of its own development efforts or the efforts of other third parties.

Risk that the Ethereum Platform, As Developed, Will Not Meet the Expectations of User

The User recognizes that the Ethereum Platform is under development and may undergo significant changes before release. User acknowledges that any expectations regarding the form and functionality of the Ethereum Platform held by the User may not be met upon release of the Ethereum Platform, for any number of reasons including a change in the design and implementation plans and execution of the implementation of the Ethereum Platform.

Risk of Security Weaknesses in the Ethereum Platform Core Infrastructure Software

The Ethereum Platform rests on open-source software, and there is a risk that the Ethereum Stiftung or the Ethereum Team, or other third parties not directly affiliated with the Stiftung Ethereum, may introduce weaknesses or bugs into the core infrastructural elements of the Ethereum Platform causing the system to lose ETH stored in one or more User accounts or other accounts or lose sums of other valued tokens issued on the Ethereum Platform.

Risk of Weaknesses or Exploitable Breakthroughs in the Field of Cryptography

Cryptography is an art, not a science. And the state of the art can advance over time Advances in code cracking, or technical advances such as the development of quantum computers, could present risks to cryptocurrencies and the Ethereum Platform, which could result in the theft or loss of ETH. To the extent possible, Stiftung Ethereum intends to update the protocol underlying the Ethereum Platform to account for any advances in cryptography and to incorporate additional security measures, but cannot it cannot predict the future of cryptography or the success of any future security updates.

Risk of Ether Mining Attacks

As with other cryptocurrencies, the blockchain used for the Ethereum Platform is susceptible to mining attacks, including but not limited to double-spend attacks, majority mining power attacks, “selfish-mining” attacks, and race condition attacks. Any successful attacks present a risk to the Ethereum Platform, expected proper execution and sequencing of ETH transactions, and expected proper execution and sequencing of contract computations. Despite the efforts of the Ethereum Stiftung and Team, known or novel mining attacks may be successful.

Risk of Rapid Adoption and Increased Demand

If the Ethereum Platform is rapidly adopted, the demand for ETH could rise dramatically and at a pace that exceeds the rate with which ETH miners can create new ETH tokens. Under such a scenario, the entire Ethereum Platform could become destabilized, due to the increased cost of running distributed applications. In turn, this could dampen interest in the Ethereum Platform and ETH. Instability in the demand of for ETH may lead to a negative change of the economical parameters of an Ethereum based business which could result in the business being unable to continue to operate economically or to cease operation.

Risk of Rapid Adoption and Insufficiency of Computational Application Processing Power on the Ethereum Platform

If the Ethereum Platform is rapidly adopted, the demand for transaction processing and distributed application computations could rise dramatically and at a pace that exceeds the rate with which ETH miners can bring online additional mining power. Under such a scenario, the entire Ethereum Platform could become destabilized, due to the increased cost of running distributed applications. In turn, this could dampen interest in the Ethereum Platform and ETH. Insufficiency of computational resources and an associated rise in the price of ETH could result in businesses being unable to acquire scarce computational resources to run their distributed applications. This would represent revenue losses to businesses or worst case, cause businesses to cease operations because such operations have become uneconomical due to distortions in the crypto-economy.

Acknowledgment, Acceptance of all Risks and Disclaimer of Warranties and Liabilities **THE USER EXPRESSLY KNOWS AND AGREES THAT THE USER IS USING THE ETHEREUM PLATFORM AT THE USER'S SOLE RISK. THE USER REPRESENTS THAT THE USER HAS AN ADEQUATE UNDERSTANDING OF THE RISKS, USAGE AND INTRICACIES OF CRYPTOGRAPHIC TOKENS AND BLOCKCHAIN-BASED OPEN SOURCE SOFTWARE, ETH PLATFORM AND ETH. THE USER ACKNOWLEDGES AND AGREES THAT, TO THE FULLEST EXTENT PERMITTED BY ANY APPLICABLE LAW, THE DISCLAIMERS OF LIABILITY CONTAINED HEREIN APPLY TO ANY AND ALL DAMAGES OR INJURY WHATSOEVER CAUSED BY OR RELATED TO RISKS OF, USE OF, OR INABILITY TO USE, ETH OR THE ETHEREUM PLATFORM UNDER ANY CAUSE OR ACTION WHATSOEVER OF ANY KIND IN ANY JURISDICTION, INCLUDING, WITHOUT LIMITATION, ACTIONS FOR BREACH OF WARRANTY, BREACH OF CONTRACT OR TORT (INCLUDING NEGLIGENCE) AND THAT NEITHER STIFTUNG ETHEREUM NOR ETHERUM TEAM SHALL BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY OR CONSEQUENTIAL DAMAGES, INCLUDING FOR LOSS OF PROFITS, GOODWILL OR DATA. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF CERTAIN WARRANTIES OR THE LIMITATION OR**

EXCLUSION OF LIABILITY FOR CERTAIN TYPES OF DAMAGES. THEREFORE, SOME OF THE ABOVE LIMITATIONS IN THIS SECTION MAY NOT APPLY TO A USER. IN PARTICULAR, NOTHING IN THESE TERMS SHALL AFFECT THE STATUTORY RIGHTS OF ANY USER OR EXCLUDE INJURY ARISING FROM ANY WILLFUL MISCONDUCT OR FRAUD OF STIFTUNG ETHEREUM.

Dispute Resolution

All disputes or claims arising out of, relating to, or in connection with the Terms, the breach thereof, or use of the Ethereum Platform shall be finally settled under the Rules of Arbitration of the International Chamber of Commerce by one or more arbitrators appointed in accordance with said Rules. All claims between the parties relating to these Terms that are capable of being resolved by arbitration, whether sounding in contract, tort, or otherwise, shall be submitted to ICC arbitration. Prior to commencing arbitration, the parties have a duty to negotiate in good faith and attempt to resolve their dispute in a manner other than by submission to ICC arbitration. The arbitration panel shall consist of one arbitrator only, unless the ICC Court of Arbitration determines that the dispute is such as to warrant three arbitrators. If the Court determines that one arbitrator is sufficient, then such arbitrator shall be Swiss resident. If the Court determines that three arbitrators are necessary, then each party shall have 30 days to nominate an arbitrator of its choice -- in the case of the Claimant, measured from receipt of notification of the ICC Court’s decision to have three arbitrators; in the case of Respondent, measured from receipt of notification of Claimant’s nomination. All nominations must be Swiss resident. If a party fails to nominate an arbitrator, the Court will do so. The Court shall also appoint the chairman. All arbitrators shall be and remain “independent” of the parties involved in the arbitration. The place of arbitration shall be Zug, Switzerland. The language of the arbitration shall be English. In deciding the merits of the dispute, the tribunal shall apply the laws of Switzerland and any discovery shall be limited and shall not involve any depositions or any other examinations outside of a formal hearing. The tribunal shall not assume the powers of amiable compositeur or decide the case ex aequo et bono. In the final award, the tribunal shall fix the costs of the arbitration and decide which of the parties shall bear such costs in what proportion. Every award shall be binding on the parties. The parties undertake to carry out the award without delay and waive their right to any form of recourse against the award in so far as such waiver can validly be made.

Force Majeure

STIFTUNG ETHEREUM is finally not liable for:

- unavoidable casualty,
- delays in delivery of materials,
- embargoes,

- government orders,
- acts of civil or military authorities,
- lack of energy, or
- any similar unforeseen event that renders performance commercially implausible.

Capabilities

By installing and running `geth`, you can take part in the ethereum frontier live network and

- mine real ether
- transfer funds between addresses
- create contracts and send transactions
- explore block history
- and much much more

Interfaces

- Javascript Console: `geth` can be launched with an interactive console, that provides a javascript runtime environment exposing a javascript API to interact with your node. Javascript Console API includes the `web3` javascript Ðapp API as well as an additional admin API.
- JSON-RPC server: `geth` can be launched with a json-rpc server that exposes the JSON-RPC API
- Command line options documents command line parameters as well as subcommands.

Basic Use Case Documentation

- Managing accounts
- Mining
- Contracts and Transactions

Note buying and selling ether through exchanges is not discussed here.

License

The Ethereum Core Protocol licensed under the GNU Lesser General Public License. All frontend client software (under cmd) is licensed under the GNU General Public License.

Contributors

Ethereum is joint work of ETHDEV and the community.

Name or blame = list of contributors:

- go-ethereum
- cpp-ethereum
- web3.js
- ethash
- netstats, netintelligence-api

Troubleshooting

If something went wrong first read our Troubleshooting checklist as well as the FAQ. If you still didn't find your answer please open an issue on GitHub or contact our help desk on helpdesk@ethereum.org.

Reporting

Security issues are best sent to security@ethereum.org or shared in PM with devs on one of the channels (see Community and Suppport).

Non-sensitive bug reports are welcome on github. Please always state the version (on master) or commit of your build (if on develop), give as much detail as possible about the situation and the anomaly that occurred. Provide logs or stacktrace if you can.

Community and support

Ethereum on social media

- Main site: https://www.ethereum.org
- ETHDEV: https://ethdev.com
- Forum: https://forum.ethereum.org
- Github: https://github.com/ethereum
- Blog: https://blog.ethereum.org
- Wiki: http://wiki.ethereum.org
- Twitter: http://twitter.com/ethereumproject
- Reddit: http://reddit.com/r/ethereum
- Meetups: http://ethereum.meetup.com
- Facebook: https://www.facebook.com/ethereumproject
- Youtube: http://www.youtube.com/ethereumproject
- Google+: http://google.com/+EthereumOrgOfficial

IRC

IRC Freenode channels:

- `#ethereum` : for general discussion
- `#ethereum-dev` : for development specific questions and discussions

- `##ethereum` : for offtopic and banter
- `#ethereumjs` : for questions related to web3.js and node-ethereum
- `#ethereum-markets` : Trading
- `#ethereum-mining` Mining
- `#dappdevs` : Dapp developers channel
- `#ethdev` : buildserver etc

IRC Logs by ZeroGox

Gitter

- go-ethereum Gitter
- cpp-ethereum Gitter
- web3.js Gitter
- ethereum documentation project Gitter

Forum

- Forum

The ZeroGox Bot

ZeroGox Bot

Dapp developers' mailing list

https://dapplist.net/

Helpdesk

On gitter, irc, skype or mail to helpdesk@ethereum.org

NOTE: These instructions are for people who want to contribute Go source code changes. If you just want to run ethereum, use the normal Installation Instructions

Developers' guide

This document is the entry point for developers of the etherum go implementation. Developers here refer to the hands-on: who are interested in build, develop, debug, submit a bug report or pull request or contribute to `go-ethereum` with code.

Build and Test

Go environment

We assume that you have `go` v1.4 installed, and `GOPATH` is set.

Note:You must have your working copy under `$GOPATH/src/github.com/ethereum/go-ethereum`. You also usually want to checkout the `develop` branch (instead of master).

Since `go` does not use relative path for import, working in any other directory will have no effect, since the import paths will be appended to `$GOPATH/src`, and if the lib does not exist, the version at master HEAD will be downloaded.

Most likely you will be working from your fork of `go-ethereum`, let's say from `github.com/nirname/go-ethereum`. Clone or move your fork into the right place:

```
git clone git@github.com:nirname/go-ethereum.git $GOPATH/src/github.com/ethereum/go-ether
```

Godep for dependency management

go-ethereum uses Godep to manage dependencies.

Install godep:

```
go get github.com/tools/godep
```

Make sure that go binaries are on your executable path:

```
PATH=$GOPATH/bin:$PATH
```

`godep` should be prepended to all go calls `build` , `install` and `test` .

Alternatively, you can prepend the go-ethereum Godeps directory to your current `GOPATH` :

```
GOPATH=`godep path`:$GOPATH
```

Building executables

Switch to the go-ethereum repository root directory (Godep expects a local Godeps folder).

Each wrapper/executable found in the `cmd` directory can be built individually.

Building Geth (CLI)

Note: Geth (the ethereum command line client) is the focus of the Frontier release.

To build the CLI:

```
godep go install -v ./cmd/geth
```

See the documentation on how to use Geth

Read about cross compilation of go-ethereum here.

Git flow

To make life easier try git flow it sets this all up and streamlines your work flow.

Testing

Testing one library:

```
godep go test -v -cpu 4 ./eth
```

Using options `-cpu` (number of cores allowed) and `-v` (logging even if no error) is recommended.

Testing only some methods:

```
godep go test -v -cpu 4 ./eth -run TestMethod
```

Note: here all tests with prefix *TestMethod* will be run, so if you got TestMethod, TestMethod1, then both!

Running benchmarks, eg.:

```
cd bzz
godep go test -v -cpu 4 -bench . -run BenchmarkJoin
```

for more see go test flags

See integration testing information on the Testing wiki page

Metrics and monitoring

`geth` can do node behaviour monitoring, aggregation and show performance metric charts. Read about metrics and monitoring

Add and update dependencies

To update a dependency version (for example, to include a new upstream fix), run

```
go get -u <foo/bar>
godep update <foo/...>
```

To track a new dependency, add it to the project as normal than run

```
godep save ./...
```

Changes to the Godeps folder should be manually verified then committed.

To make life easier try git flow it sets this all up and streamlines your work flow.

Contributing

Only github is used to track issues. (Please include the commit and branch when reporting an issue.)

Pull requests should by default commit on the `develop` branch. The `master` branch is only used for finished stable major releases.

Stacktrace

The code uses `pprof` on localhost port 6060 by default if `geth` is started with the `--pprof` option. So bring up http://localhost:6060/debug/pprof to see the heap, running routines etc. By clicking full goroutine stack dump (clicking http://localhost:6060/debug/pprof/goroutine?debug=2) you can generate trace that is useful for debugging.

Note that if you run multiple instances of `geth`, this port will only work for the first instance that was launched. If you want to generate stacktraces for these other instances, you need to start them up choosing an alternative pprof port. Make sure you are redirecting stderr to a logfile.

```
geth -port=30300 -loglevel 5 --pprof --pprofport 6060 2>> /tmp/00.glog
geth -port=30301 -loglevel 5 --pprof --pprofport 6061 2>> /tmp/01.glog
geth -port=30302 -loglevel 5 --pprof --pprofport 6062 2>> /tmp/02.glog
```

Alternatively if you want to kill the clients (in case they hang or stalled synching, etc) but have the stacktrace too, you can use the `-QUIT` signal with `kill`:

```
killall -QUIT geth
```

This will dump stracktraces for each instance to their respective log file.

Code formatting

Sources are formatted according to the Go Formatting Style.

Dev Tutorials

- Private networks, local clusters and monitoring
- P2P 101: a tutorial about setting up and creating a p2p server and p2p sub protocol.
- How to Whisper: an introduction to whisper.

Getting Geth

The Frontier tool is called Geth (the old english third person singular conjugation of "to go". Quite appropriate given geth is written in Go. Geth is a multipurpose command line tool that runs a full Ethereum node implemented in Go. It offers three interfaces: the command line subcommands and options, a Json-rpc server and an interactive console.

In order to install Geth, open a command line or terminal tool (if you are unsure how to do this, consider waiting for a more user friendly release) and paste the command below:

```
bash <(curl https://install-geth.ethereum.org)
```

This script will detect your OS and will attempt to install the ethereum CLI. For more options including package managers, check the OS-specific subsections.

First run

For the purposes of this guide, we will focus on the interactive console, a JavaScript environment. The history of the console is persisted between sessions, providing for a powerful and flexible way of using the client. You can navigate your command history by using the up and down arrow keys, like a standard command line. To get started Type the code below on your terminal

```
geth console
```

Once geth is fully started, you should see a `>` prompt, letting you know the console is ready. To exit, type `exit` at the prompt and hit `[enter]`.

Using stderr

Output from the console can be logged or redirected:

`geth console 2>>geth.log`

Using standard tools, the log can be monitored in a separate window:

`tail -f geth.log`

Alternatively, you can also run one terminal with the interactive console and a second one with the logging output directly.

1. Open two terminals
2. In the **second** terminal type `tty`. The output will be something like `/dev/pts/13`
3. In your main terminal, type: `geth console 2>> /dev/pts/13`

This will allow you to monitor your node without cluttering the interactive console.

Installation Instructions

Follow the appropriate link below to find installation instructions for your platform.

- Installation Instructions for Mac OS X
- Installation Instructions for Windows
- Installation Instructions for Linux/Unix
    - Ubuntu
    - Arch
    - FreeBSD
- Setup for Raspberry Pi
    - ARM
- Usage instructions for Docker

You can also use a one-line script install Geth. Open a command line or terminal tool (if you are unsure how to do this, consider waiting for a more user friendly release) and paste the command below:

```
bash <(curl -L https://install-geth.ethereum.org)
```

This script will detect your OS and will attempt to install the ethereum CLI. For more options including package managers, check the OS-specific subsections.

Further links

- Compiled binaries on launchpad.net buildbot listings
- Ethereum buildbot

Linux installation options

Debian/Ubuntu

Installing from PPA

For the latest development snapshot, both `ppa:ethereum/ethereum` and `ppa:ethereum/ethereum-dev` are needed. If you want the stable version from the last PoC release, add only the first one.

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo add-apt-repository -y ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install ethereum
```

After installing, run `geth account new` to create an account on your node.

You should now be able to run `geth` and connect to the network.

Make sure to check the different options and commands with `geth --help`

You can alternatively install only the `geth` CLI with `apt-get install geth` if you don't want to install the other utilities (`bootnode` , `evm` , `disasm` , `rlpdump` , `ethtest`).

Building from source

Building Geth (command line client)

Clone the repository to a directory of your choosing:

```
git clone https://github.com/ethereum/go-ethereum
```

Install latest distribution of Go (v1.4) if you don't have it already:

See instructions

Building `geth` requires some external libraries to be installed:

```
sudo apt-get install -y build-essential libgmp3-dev golang
```

Finally, build the `geth` program using the following command.

```
cd go-ethereum
make geth
```

You can now run `build/bin/geth` to start your node.

Arch

Installing from source

Install dependencies

```
pacman -S git go gcc gmp
```

Download and build geth

```
git clone https://github.com/ethereum/go-ethereum
cd go-ethereum
make geth
```

Mac installation options

Installing with Homebrew

By far the easiest way to install go-ethereum is to use our Homebrew tap. If you don't have Homebrew, install it first.

Then run the following commands to add the tap and install `geth` :

```
brew tap ethereum/ethereum
brew install ethereum
```

You can install the develop branch by running `--devel` :

```
brew install ethereum --devel
```

After installing, run `geth account new` to create an account on your node.

You should now be able to run `geth` and connect to the network.

Make sure to check the different options and commands with `geth --help`

For options and patches, see: https://github.com/ethereum/homebrew-ethereum

Building from source

Building Geth (command line client)

Clone the repository to a directory of your choosing:

```
git clone https://github.com/ethereum/go-ethereum
```

Building `geth` requires some external libraries to be installed:

- GMP
- Go

```
brew install gmp go
```

Finally, build the `geth` program using the following command.

```
cd go-ethereum
make geth
```

You can now run `build/bin/geth` to start your node.

Windows installation options

Installing from Chocolatey

For master branch:

```
choco install geth-stable
```

For more information see https://chocolatey.org/packages/geth-stable

For develop branch:

```
choco install geth-latest
```

For more information see https://chocolatey.org/packages/geth-latest

Building from source

1. Install Git from http://git-scm.com/downloads
2. Install Golang from https://storage.googleapis.com/golang/go1.4.2.windows-amd64.msi
3. Install winbuilds from http://win-builds.org/1.5.0/win-builds-1.5.0.exe to `c:\winbuilds`
4. Run win builds here. It's safe to remove big dependencies like QT and GTK which aren't needed. *An exact list of dependencies should be determined*
5. Setup environment paths
    i. Add `GOROOT` pointed to `c:\go` and `GOPATH` to `c:\godev` (you are free to pick these paths).
    ii. Set `PATH` to `%PATH%;%GOROOT%\bin;%GOPATH%\bin;c:\winbuilds\bin`
6. Open a terminal and install godep first: `go get -u github.com/tools/godep`
7. Open a terminal and download go-ethereum `go get -d -u github.com/ethereum/go-ethereum`
8. Try building ethereum with go dep, navigate to `c:\godev\src\github.com\ethereum\go-ethereum\cmd\geth` and run `git checkout develop && godep go install`

If you want to build from an other branch bypass `godep go install` for `go install` and checkout the dependencies manually.

Powershell script for building with Cygwin

Warning: *This installation method currently fails to link properly. Giving the message "ld: cannot find -lmingwex" and "ld: cannot find -lmingw32"*

```
#REQUIRES -Version 3.0

# Set local directory paths
$basedir = $env:USERPROFILE
$downloaddir = "$basedir\Downloads"

# Set Go variables
$golangroot = "$basedir\golang"
$gosrcroot = "$basedir\go"
$golangdl = "https://storage.googleapis.com/golang/"

# Set cygwin variables
$cygwinroot = "$basedir\cygwin"
$cygwinpackages = "gcc-g++,binutils,make,git,gmp,libgmp10,libgmp-devel"
$cygwinmirror = "http://cygwin.mirrorcatalogs.com"
$cygwindl = "https://cygwin.com/"

# Finalize paths based on processor architecture
if ($ENV:PROCESSOR_ARCHITECTURE -eq "AMD64") {
  $golangdl = $golangdl + "go1.5.1.windows-amd64.zip"
  $cygwindl = $cygwindl + "setup-x86_64.exe"
} else {
  $golangdl = $golangdl + "go1.5.1.windows-386.zip"
  $cygwindl = $cygwindl + "setup-x86.exe"
}

# Download dependencies
Invoke-WebRequest $cygwindl -UseBasicParsing -OutFile "$downloaddir\cygwin-setup.exe"
Invoke-WebRequest $golangdl -UseBasicParsing -OutFile "$downloaddir\golang.zip"

# Install Cygwin & dependencies
Invoke-Expression "$downloaddir\cygwin-setup.exe --root $cygwinroot --site $cygwinmirror
# Install Golang
Add-Type -AssemblyName System.IO.Compression.FileSystem
[System.IO.Compression.ZipFile]::ExtractToDirectory("$downloaddir\golang.zip", $golangroo

# Set environment variables
# Only works locally
$env:GOROOT = "$golangroot\go"
$env:GOPATH = $gosrcroot
$env:PATH = "$env:PATH;$cygwinroot\bin;$golangroot\go\bin;$gosrcroot\bin"
# Only works in new sessions
[Environment]::SetEnvironmentVariable("GOROOT", $env:GOROOT, "User")
[Environment]::SetEnvironmentVariable("GOPATH", $env:GOPATH, "User")
[Environment]::SetEnvironmentVariable("PATH", $env:PATH, "User")


# Download go-ethereum source
go get github.com/tools/godep
```

```
git clone https://github.com/ethereum/go-ethereum $env:GOPATH/src/github.com
cd $env:GOPATH/src/github.com/ethereum/go-ethereum
godep go install .\cmd\geth
```

Running in Docker

We keep a Docker image with recent snapshot builds from the `develop` branch on DockerHub. Run this first:

```
docker pull ethereum/client-go
```

Start a node with:

```
docker run -it -p 30303:30303 ethereum/client-go
```

To start a node that runs the JSON-RPC interface on port **8545**, run:

```
docker run -it -p 8545:8545 -p 30303:30303 ethereum/client-go --rpc
```

WARNING: This opens your container to external calls

To use the interactive JavaScript console, run:

```
docker run -it -p 30303:30303 ethereum/client-go console
```

Installation Instructions for ARM

Geth is built for ARM using cross-compilation. See Cross compiling Ethereum for more details.

RasPi 2

1. Download the precompiled binary from master branch
2. Copy it to a location in $PATH (i.e. /usr/bin/local)
3. Run `geth`

Further details: https://github.com/ethereum/wiki/wiki/Raspberry-Pi-instructions

DO NOT FORGET YOUR PASSWORD and **BACKUP YOUR KEYSTORE**

Backup & restore

Data directory

Everything `geth` persists gets written inside its data directory (except for the PoW Ethash DAG, see note below). The default data directory locations are platform specific:

- Mac: `~/Library/Ethereum`
- Linux: `~/.ethereum`
- Windows: `%APPDATA%/Ethereum`

Accounts are stored in the `keystore` subdirectory. The contents of this directories should be transportable between nodes, platforms, implementations (C++, Go, Python).

To configure the location of the data directory, the `--datadir` parameter can be specified. See CLI Options for more details.

Note: *The Ethash DAG is stored at `~/.ethash` (Mac/Linux) or `~/AppData/Ethash` (Windows) so that it can be reused by all clients. You can store this in a different location by using a symbolic link.*

Upgrades

Sometimes the internal database formats need updating (for example, when upgrade from before 0.9.20). This can be run with the following command (geth should not be otherwise running):

```
geth upgradedb
```

Cleanup

Geth's blockchain and state databases can be removed with:

```
geth removedb
```

This is useful for deleting an old chain and sync'ing to a new one. It only affects data directories that can be re-created on synchronisation and does not touch the keystore.

Blockchain import/export

Export the blockchain in binary format with:

```
geth export <filename>
```

Or if you want to back up portions of the chain over time, a first and last block can be specified. For example, to back up the first epoch:

```
geth export <filename> 0 29999
```

Note that when backing up a partial chain, the file will be appended rather than truncated.

Import binary-format blockchain exports with:

```
geth import <filename>
```

See https://github.com/ethereum/wiki/wiki/Blockchain-import-export for more info

And finally: **DO NOT FORGET YOUR PASSWORD** and **BACKUP YOUR KEYSTORE**

Connecting to the network

How Peers Are Found

Geth continuously attempts to connect to other nodes on the network until it has peers. If you have UPnP enabled on your router or run ethereum on an Internet-facing server, it will also accept connections from other nodes.

Geth finds peers through something called the discovery protocol. In the discovery protocol, nodes are gossipping with each other to find out about other nodes on the network. In order to get going initially, geth uses a set of bootstrap nodes whose endpoints are recorded in the source code.

To change the bootnodes on startup, use the `--bootnodes` option and separate the nodes by spaces. For example:

```
geth --bootnodes "enode://pubkey1@ip1:port1 enode://pubkey2@ip2:port2 enode://pubkey3@ip3
```

Common Problems With Connectivity

Sometimes you just can't get connected. The most common reasons are as follows:

- Your local time might be incorrect. An accurate clock is required to participate in the Ethereum network. Check your OS for how to resync your clock (example sudo ntpdate -s time.nist.gov) because even 12 seconds too fast can lead to 0 peers.
- Some firewall configurations can prevent UDP traffic from flowing. You can use the static nodes feature or `admin.addPeer()` on the console to configure connections by hand.

To start geth without the discovery protocol, you can use the `--nodiscover` parameter. You only want this is you are running a test node or an experimental test network with fixed nodes.

Checking Connectivity

To check how many peers the client is connected to in the interactive console, the `net` module has two attributes give you info about the number of peers and whether you are a listening node.

```
> net.listening
true
> net.peerCount
4
```

To get more information about the connected peers, such as IP address and port number, supported protocols, use the `peers()` function of the `admin` object. `admin.peers()` returns the list of currently connected peers.

```
> admin.peers
[{
  ID: 'a4de274d3a159e10c2c9a68c326511236381b84c9ec52e72ad732eb0b2b1a2277938f78593cdbe734e
  Name: 'Geth/v0.9.14/linux/go1.4.2',
  Caps: 'eth/60',
  RemoteAddress: '5.9.150.40:30301',
  LocalAddress: '192.168.0.28:39219'
}, {
  ID: 'a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69ad0dce72a4d8db
  Name: 'Geth/v0.9.15/linux/go1.4.2',
  Caps: 'eth/60',
  RemoteAddress: '52.16.188.185:30303',
  LocalAddress: '192.168.0.28:50995'
}, {
  ID: 'f6ba1f1d9241d48138136ccf5baa6c2c8b008435a1c2bd009ca52fb8edbbc991eba36376beaee9d45f
  Name: 'pyethapp_dd52/v0.9.13/linux2/py2.7.9',
  Caps: 'eth/60, p2p/3',
  RemoteAddress: '144.76.62.101:30303',
  LocalAddress: '192.168.0.28:40454'
}, {
  ID: 'f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0ceae2e826f293c
  Name: '++eth/Zeppelin/Rascal/v0.9.14/Release/Darwin/clang/int',
  Caps: 'eth/60, shh/2',
  RemoteAddress: '129.16.191.64:30303',
  LocalAddress: '192.168.0.28:39705'
} ]
```

To check the ports used by geth and also find your enode URI run:

```
> admin.nodeInfo
{
  Name: 'Geth/v0.9.14/darwin/go1.4.2',
  NodeUrl: 'enode://3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2ba60f1499
  NodeID: '3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2ba60f14998a3a98c0c
  IP: '::',
  DiscPort: 30303,
  TCPPort: 30303,
  Td: '2044952618444',
  ListenAddr: '[::]:30303'
}
```

Custom Networks

Sometimes you might not need to connect to the live public network, you can instead choose to create your own private testnet. This is very useful if you don't need to test external contracts and want just to test the technology, because you won't have to compete with other miners and will easily generate a lot of test ether to play around (replace 12345 with any non-negative number):

```
geth -–networkid="12345" console
```

It is also possible to run geth with a custom genesis block from a JSON file by supplying the `--genesis` flag. The genesis JSON file should have the following format:

```
{
  "alloc": {
    "dbdbdb2cbd23b783741e8d7fcf51e459b497e4a6": {
        "balance": "1606938044258990275541962092341162602522202993782792835301376"
    },
    "e6716f9544a56c530d868e4bfbacb172315bdead": {
      "balance": "1606938044258990275541962092341162602522202993782792835301376"
    },
    ...
  },
  "nonce": "0x000000000000002a",
  "difficulty": "0x020000",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x0000000000000000000000000000000000000000",
  "timestamp": "0x00",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "0x",
  "gasLimit": "0x2fefd8"
}
```

Static nodes

Geth also supports a feature called static nodes if you have certain peers you always want to connect to. Static nodes are re-connected on disconnects. You can configure permanent static nodes by putting something like the following into `<datadir>/static-nodes.json` :

```
[
  "enode://f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0ceae2e826f
  "enode://pubkey@ip:port"
]
```

You can also add static nodes at runtime via the js console using `admin.addPeer()` :

```
admin.addPeer("enode://f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102
```

Caveat: Currently the console is lacking support for removing a peer, increasing peercount or adding a non-static peer but not to keep try reconnecting.

Network Status Monitoring

The Ethereum (centralised) network status monitor (known sometimes as "eth-netstats") is a web-based application to monitor the health of the testnet/mainnet through a group of nodes.

Listing

To list your node, you must install the client-side information relay, a node module. Instructions given here work on Ubuntu (Mac OS X follow same instructions, but sudo may be unnecessary). Other platforms vary (please make sure that nodejs-legacy is also installed, otherwise some modules might fail).

Clone the git repo, then install pm2:

```
git clone https://github.com/cubedro/eth-net-intelligence-api
cd eth-net-intelligence-api
npm install
sudo npm install -g pm2
```

Then edit the `app.json` file in it to configure for your node:

- alter the value to the right of `LISTENING_PORT` to the ethereum listening port (default: 30303)
- alter the value to the right of `INSTANCE_NAME` to whatever you wish to name your node;
- alter the value to the right of `CONTACT_DETAILS` if you wish to share your contact details
- alter the value to the right of `RPC_PORT` to the rpc port for your node (by default 8545 for both cpp and go);
- and alter the value to the right of `WS_SECRET` to the secret (you'll have to get this off the official skype channel).

Finally run the process with:

```
pm2 start app.json
```

Several commands are available:

- `pm2 list` to display the process status;
- `pm2 logs` to display logs;
- `pm2 gracefulReload node-app` for a soft reload;
- `pm2 stop node-app` to stop the app;

- `pm2 kill` to kill the daemon.

Updating

In order to update you have to do the following:

- `git pull` to pull the latest version
- `sudo npm update` to update the dependencies
- `pm2 gracefulReload node-app` to reload the client

Auto-installation on a fresh Ubuntu install

Fetch and run the build shell. This will install everything you need: latest ethereum - CLI from develop branch (you can choose between eth or geth), node.js, npm & pm2.

```
bash <(curl https://raw.githubusercontent.com/cubedro/eth-net-intelligence-api/master/bin
```

Configuration

Configure the app modifying processes.json. Note that you have to modify the backup processes.json file located in `./bin/processes.json` (to allow you to set your env vars without being rewritten when updating).

```
"env":
    {
        "NODE_ENV"        : "production", // tell the client we're in production environm
        "RPC_HOST"        : "localhost", // eth JSON-RPC host the default is 8545
        "RPC_PORT"        : "8545", // eth JSON-RPC port
        "LISTENING_PORT"  : "30303", // eth listening port (only used for display)
        "INSTANCE_NAME"   : "", // whatever you wish to name your node
        "CONTACT_DETAILS" : "", // add your contact details here if you wish (email/skype
        "WS_SERVER"       : "wss://stats.ethdev.com", // path to eth-netstats WebSockets
        "WS_SECRET"       : "", // WebSockets api server secret used for login
    }
```

Run

Run it using pm2:

```
cd ~/bin
pm2 start processes.json
```

ethereum (eth or geth) must be running with rpc enabled.

```
geth --rpc
```

the default port (if one is not specified) for rpc under geth is 8545

Updating

To update the API client use the following command:

```
~/bin/www/bin/update.sh
```

It will stop the current netstats client processes, automatically detect your ethereum implementation and version, update it to the latest develop build, update netstats client and reload the processes.

Setting up a cluster

This page describes how to set up a local cluster of nodes, advise how to make it private, and how to hook up your nodes on the eth-netstat network monitoring app. A fully controlled ethereum network is useful as a backend for network integration testing (core developers working on issues related to networking/blockchain synching/message propagation, etc or DAPP developers testing multi-block and multi-user scenarios).

We assume you are able to build `geth` following the build instructions

Setting up multiple nodes

In order to run multiple ethereum nodes locally, you have to make sure:

- each instance has a separate data directory (`--datadir`)
- each instance runs on a different port (both eth and rpc) (`--port and --rpcport`)
- in case of a cluster the instances must know about each other
- the ipc endpoint is unique or the ipc interface is disabled (`--ipcpath or --ipcdisable`)

You start the first node (let's make port explicit and disable ipc interface)

```
geth --datadir="/tmp/eth/60/01" -verbosity 6 --ipcdisable --port 30301 --rpcport 8101 con
```

We started the node with the console, so that we can grab the enode url for instance:

```
> admin.nodeInfo.NodeUrl
enode://8c544b4a07da02a9ee024def6f3ba24b2747272b64e16ec5dd6b17b55992f8980b77938155169d9d3
```

`[::]` will be parsed as localhost (`127.0.0.1`). If your nodes are on a local network check each individual host machine and find your ip with `ifconfig` (on Linux and MacOS):

```
$ ifconfig|grep netmask|awk '{print $2}'
127.0.0.1
192.168.1.97
```

If your peers are not on the local network, you need to know your external IP address (use a service) to construct the enode url.

Now you can launch a second node with:

```
geth --datadir="/tmp/eth/60/02" --verbosity 6 --ipcdisable --port 30302 --rpcport 8102 co
```

If you want to connect this instance to the previously started node you can add it as a peer from the console with `admin.addPeer(enodeUrlOfFirstInstance)` .

You can test the connection by typing in geth console:

```
> net.listening
true
> net.peerCount
1
> admin.peers
...
```

Local cluster

As an extention of the above, you can spawn a local cluster of nodes easily. It can also be scripted including account creation which is needed for mining. See `gethcluster.sh` script, and the README there for usage and examples.

Private network

An ethereum network is a private network if the nodes are not connected to the main network nodes. In this context private only means reserved or isolated, rather than protected or secure. Since connections between nodes are valid only if peers have identical protocol version and network id, you can effectively isolate your network by setting either of these to a non default value. We recommend using the semantic `networkid` command line option for this. Its argument is an integer, the main network has id 1 (the default). So if you supply your own custom network id which is different than the main network your nodes will not connect to other nodes and form a private network.

Monitoring your nodes

This page describes how to use the The Ethereum (centralised) network status monitor (known sometimes as "eth-netstats") to monitor your nodes.

This page or this README describes how you set up your own monitoring service for a (private or public) local cluster.

Managing accounts

WARNING Remember your password.

If you lose the password you use to encrypt your account, you will not be able to access that account. Repeat: It is NOT possible to access your account without a password and there is no *forgot my password* option here. Do not forget it.

Note: the key file name naming convention changed as of `0.9.36`. This document is meant to reflect accurate information on accounts as used by the frontier release.

The ethereum CLI `geth` provides account management via the `account` subcommand:

```
account command [arguments...]
```

Manage accounts lets you create new accounts, list all existing accounts, import a private key into a new account, migrate to newest key format and change your password.

It supports interactive mode, when you are prompted for password as well as non-interactive mode where passwords are supplied via a given password file. Non-interactive mode is only meant for scripted use on test networks or known safe environments.

Make sure you remember the password you gave when creating a new account (with new, update or import). Without it you are not able to unlock your account.

Note that exporting your key in unencrypted format is NOT supported.

Keys are stored under `<DATADIR>/keystore`. Make sure you backup your keys regularly! See DATADIR backup & restore for more information. The newest format of the keyfiles is: `UTC--<created_at UTC ISO8601>-<address hex>`. The order of accounts when listing, is lexicographic, but as a consequence of the timespamp format, it is actually order of creation

It is safe to transfer the entire directory or the individual keys therein between ethereum nodes. Note that in case you are adding keys to your node from a different node, the order of accounts may change. So make sure you do not rely or change the index in your scripts or code snippets.

And finally. **DO NOT FORGET YOUR PASSWORD**

```
SUBCOMMANDS:

        list    print account addresses
        new     create a new account
        update  update an existing account
        import  import a private key into a new account
```

You can get info about further subcommands by `geth account help <subcommand>`

Accounts can also be managed via the Javascript Console

Examples

Interactive use

creating an account

```
$ geth account new
Your new account is locked with a password. Please give a password. Do not forget this pa
Passphrase:
Repeat Passphrase:
Address: {168bc315a2ee09042d83d7c5811b533620531f67}
```

Listing accounts

```
$ geth account list
Account #0: {a94f5374fce5edbc8e2a8697c15331677e6ebf0b}
Account #1: {c385233b188811c9f355d4caec14df86d6248235}
Account #2: {7f444580bfef4b9bc7e14eb7fb2a029336b07c9d}
```

Import private key

```
$ geth --datadir /someOtherEthDataDir  account import ./key.prv
The new account will be encrypted with a passphrase.
Please enter a passphrase now.
Passphrase:
Repeat Passphrase:
Address: {7f444580bfef4b9bc7e14eb7fb2a029336b07c9d}
```

Account update

```
$ geth account update a94f5374fce5edbc8e2a8697c15331677e6ebf0b
Unlocking account a94f5374fce5edbc8e2a8697c15331677e6ebf0b | Attempt 1/3
Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
Account 'a94f5374fce5edbc8e2a8697c15331677e6ebf0b' unlocked.
Please give a new password. Do not forget this password.
Passphrase:
Repeat Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
```

Non-interactive use

You supply a plaintext password file as argument to the `--password` flag. The data in the file consists of the raw characters of the password, followed by a single newline.

Note: Supplying the password directly as part of the command line is not recommended, but you can always use shell trickery to get round this restriction.

```
$ geth --password /path/to/password account new

$ geth --password /path/to/password account update b0047c606f3af7392e073ed13253f8f4710b08

$ geth account list

$ geth --datadir /someOtherEthDataDir --password /path/to/anotherpassword account import
```

Creating accounts

Creating a new account

```
geth account new
```

Creates a new account and prints the address.

On the console, use:

```
> personal.newAccount("passphrase")
```

The account is saved in encrypted format. You **must** remember this passphrase to unlock your account in the future.

For non-interactive use the passphrase can be specified with the `--password` flag:

```
geth --password <passwordfile> account new
```

Note, this is meant to be used for testing only, it is a bad idea to save your password to file or expose in any other way.

Creating an account by importing a private key

```
    geth account import <keyfile>
```

Imports an unencrypted private key from `<keyfile>` and creates a new account and prints the address.

The keyfile is assumed to contain an unencrypted private key as canonical EC raw bytes encoded into hex.

The account is saved in encrypted format, you are prompted for a passphrase.

You must remember this passphrase to unlock your account in the future.

For non-interactive use the passphrase can be specified with the `--password` flag:

```
geth --password <passwordfile> account import <keyfile>
```

Note: Since you can directly copy your encrypted accounts to another ethereum instance, this import/export mechanism is not needed when you transfer an account between nodes.

Warning: when you copy keys into an existing node's keystore, the order of accounts you are used to may change. Therefore you make sure you either do not rely on the account order or doublecheck and update the indexes used in your scripts.

Warning: If you use the password flag with a password file, best to make sure the file is not readable or even listable for anyone but you. You achieve this with:

```
touch /path/to/password
chmod 700 /path/to/password
cat > /path/to/password
>I type my pass here^D
```

Updating an existing account

You can update an existing account on the command line with the `update` subcommand with the account address or index as parameter.

```
geth account update b0047c606f3af7392e073ed13253f8f4710b08b6
geth account update 2
```

The account is saved in the newest version in encrypted format, you are prompted for a passphrase to unlock the account and another to save the updated file.

This same command can therefore be used to migrate an account of a deprecated format to the newest format or change the password for an account.

For non-interactive use the passphrase can be specified with the `--password` flag:

```
geth --password <passwordfile> account new
```

Since only one password can be given, only format update can be performed, changing your password is only possible interactively.

Note: Account update has the a side effect that the order of your accounts changes.

After a successful update, all previous formats/versions of that same key will be removed!

Importing your presale wallet

Importing your presale wallet is very easy. If you remember your password that is:

```
geth wallet import /path/to/my/presale.wallet
```

will prompt for your password and imports your ether presale account. It can be used non-interactively with the --password option taking a passwordfile as argument containing the wallet password in cleartext.

Listing accounts and checking balances

Listing your current accounts

From the command line, call the CLI with:

```
$ geth account list
Account #0: {d1ade25ccd3d550a7eb532ac759cac7be09c2719}
Account #1: {da65665fc30803cb1fb7e6d86691e20b1826dee0}
Account #2: {e470b1a7d2c9c5c6f03bbaa8fa20db6d404a0c32}
Account #3: {f4dd5c3794f1fd0cdc0327a83aa472609c806e99}
```

to list your accounts in order of creation.

Note: This order can change if you copy keyfiles from other nodes, so make sure you either do not rely on indexes or make sure if you copy keys you check and update your account indexes in your scripts.

When using the console:

```
> eth.accounts
['0x407d73d8a49eeb85d32cf465507dd71d507100c1']
```

or via RPC:

```
# Request
$ curl -X POST --data '{"jsonrpc":"2.0","method":"eth_accounts","params":[],"id":1} http:

# Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": ["0x407d73d8a49eeb85d32cf465507dd71d507100c1"]
}
```

If you want to use an account non-interactively, you need to unlock it. You can do this on the command line with the `--unlock` option which takes a whitespace separated list of accounts (in hex or index) as argument so you can unlock the accounts programmatically for one session. This is useful if you want to use your account from Dapps via RPC. `--unlock` will unlock the first account. This is useful when you created your account programmatically, you do not need to know the actual account to unlock it.

Unlocking one account:

```
geth --password <(echo this is not secret!) account new
geth --password <(echo this is not secret!) --unlock primary --rpccorsdomain localhost --
```

Instead of the account address, you can use integer indexes which refers to the address position in the account listing (and corresponds to order of creation)

The command line allows you to unlock multiple accounts. In this case the argument to unlock is a whitespace delimited list of accounts addresses or indexes.

```
geth --unlock "0x407d73d8a49eeb85d32cf465507dd71d507100c1 0 5 e470b1a7d2c9c5c6f03bbaa8fa2
```

If this construction is used non-interactively, your password file will need to contain the respective passwords for the accounts in question, one per line.

On the console you can also unlock accounts (one at a time).

```
personal.unlockAccount(address, "password")
```

Note that we do NOT recommend using the password argument here, since the console history is logged, so you may compromise your account. You have been warned.

Checking account balances

To check your the etherbase account balance:

```
> web3.fromWei(eth.getBalance(eth.coinbase), "ether")
6.5
```

Print all balances with a JavaScript function:

```
function checkAllBalances() {
var i =0;
eth.accounts.forEach( function(e){
     console.log("  eth.accounts["+i+"]: " +  e + " \tbalance: " + web3.fromWei(eth.getBa
i++;
})
};
```

That can then be executed with:

```
> checkAllBalances();
  eth.accounts[0]: 0xd1ade25ccd3d550a7eb532ac759cac7be09c2719     balance: 63.11848 ether
  eth.accounts[1]: 0xda65665fc30803cb1fb7e6d86691e20b1826dee0     balance: 0 ether
  eth.accounts[2]: 0xe470b1a7d2c9c5c6f03bbaa8fa20db6d404a0c32     balance: 1 ether
  eth.accounts[3]: 0xf4dd5c3794f1fd0cdc0327a83aa472609c806e99     balance: 6 ether
```

Since this function will disappear after restarting geth, it can be helpful to store commonly used functions to be recalled later. The loadScript function makes this very easy.

First, save the `checkAllBalances()` function definition to a file on your computer. For example, `/Users/username/gethload.js`. Then load the file from the interactive console:

```
> loadScript("/Users/username/gethload.js")
true
```

The file will modify your JavaScript environment as if you has typed the commands manually. Feel free to experiment!

Sending ether

The basic way of sending a simple transaction of ether with the console is as follows:

```
> eth.sendTransaction({from:sender, to:receiver, value: amount})
```

Using the built-in JavaScript, you can easily set variables to hold these values. For example:

```
> var sender = eth.accounts[0];
> var receiver = eth.accounts[1];
> var amount = web3.toWei(0.01, "ether")
```

Alternatively, you can compose a transaction in a single line with:

```
> eth.sendTransaction({from:eth.coinbase, to:eth.accounts[1], value: web3.toWei(0.05, "et
Please unlock account d1ade25ccd3d550a7eb532ac759cac7be09c2719.
Passphrase:
Account is now unlocked for this session.
'0xeeb66b211e7d9be55232ed70c2ebb1bcc5d5fd9ed01d876fac5cff45b5bf8bf4'
```

The resulting transaction is `0xeeb66b211e7d9be55232ed70c2ebb1bcc5d5fd9ed01d876fac5cff45b5bf8bf4`

If the password was incorrect you will instead receive an error:

```
error: could not unlock sender account
```

Introduction

The word mining originates in the context of the gold analogy for crypto currencies. Gold or precious metals are scarce, so are digital tokens, and the only way to increase the total volume is through mining it. This is appropriate to the extent that in Ethereum too, the only mode of issuance post launch is via the mining. Unlike these examples however, mining is also the way to secure the network by creating, verifying, publishing and propagating blocks in the blockchain.

- Mining Ether = Securing the network = verify computation

So what is mining anyway?

Ethereum Frontier like all blockchain technologies uses an incentive-driven model of security. Consensus is based on choosing the block with the highest total difficulty. Miners produce blocks which the others check for validity. Among other well-formedness criteria, a block is only valid if it contains **proof of work** (PoW) of a given **difficulty**. Note that in Ethereum 1.1, this is likely gonna be replaced by a **proof of stake** model.

[The proof of work algorithm used is called Ethash (a modified version of Dagger-Hashimoto involves finding a nonce input to the algorithm so that the result is below a certain threshold depending on the difficulty. The point in PoW algorithms is that there is no better strategy to find such a nonce than enumerating the possibilities while verification of a solution is trivial and cheap. If outputs have a uniform distribution, then we can guarantee that on average the time needed to find a nonce depends on the difficulty threshold, making it possible to control the time of finding a new block just by manipulating difficulty.

The difficulty dynamically adjusts so that on average one block is produced by the entire network every 12 seconds (ie., 12 s block time). This heartbeat basically punctuates the synchronisation of system state and guarantees that maintaining a fork (to allow double spend) or rewriting history is impossible unless the attacker possesses more than half of the network mining power (so called 51% attack).

Any node participating in the network can be a miner and their expected revenue from mining will be directly proportional to their (relative) mining power or **hashrate**, ie., number of nonces tried per second normalised by the total hashrate of the network.

Ethash PoW is memory hard, making it basically ASIC resistant. This basically means that calculating the PoW requires choosing subsets of a fixed resource dependent on the nonce and block header. This resource (a few gigabyte size data) is called a **DAG**. The DAG is totally different every 30000 blocks (a 100 hour window, called an **epoch**) and takes a while to generate. Since the DAG only depends on block height, it can be pregenerated but if its not, the client need to wait the end of this process to produce a block. Until clients actually precache dags ahead of time the network may experience a massive block delay on each epoch transition. Note that the DAG does not need to be generated for verifying the PoW essentially allowing for verification with both low CPU and small memory.

As a special case, when you start up your node from scratch, mining will only start once the DAG is built for the current epoch.

Mining Rewards

Note that mining 'real' Ether will start with the Frontier release. On the Olympics testnet, the Frontier pre-release, the ether mined have no value (but see Olympic rewards).

The successful PoW miner of the winning block receives:

- A **static block reward** for the 'winning' block, consisting of exactly 5.0 Ether
- All of the gas expended within the block, that is, all the gas consumed by the execution of all the transactions in the block submitted by the winning miner is compensated for by the senders. The gascost incurred is credited to the miner's account as part of the consensus protocoll. Over time, it's expected these will dwarf the static block reward.
- An extra reward for including Uncles as part of the block, in the form of an extra 1/32 per Uncle included

Uncles are stale blocks, ie with parent that are ancestors (max 6 blocks back) of the including block. Valid uncles are rewarded in order to neutralise the effect of network lag on the dispersion of mining rewards, thereby increasing security. Uncles included in a block formed by the successful PoW miner receive 7/8 of the static block reward = 4.375 ether A maximum of 2 uncles allowed per block.

Ethash DAG

Ethash uses a **DAG** (directed acyclic graph) for the proof of work algorithm, this is generated for each **epoch**, i.e every 30000 blocks (100 hours). The DAG takes a long time to generate. If clients only generate it on demand, you may see a long wait at each epoch transition before the first block of the new epoch is found. However, the DAG only depends on block number, so it CAN and SHOULD be calculated in advance to avoid long wait at each epoch transition. `geth` implements automatic DAG generation and maintains two DAGS at a time for smooth epoch transitions. Automatic DAG generation is turned on and off when mining is controlled from the console. It is also turned on by default if `geth` is launched with the `--`

`mine` option. Note that clients share a DAG resource, so if you are running multiple instances of any client, make sure automatic dag generation is switched on in at most one client.

To generate the DAG for an arbitrary epoch:

```
geth makedag <block number> <outputdir>
```

For instance `geth makedag 360000 ~/.ethash`. Note that ethash uses `~/.ethash` (Mac/Linux) or `~/AppData/Ethash` (Windows) for the DAG so that it can shared between clients.

CPU Mining with Geth

At Frontier, the first release of Ethereum, you'll just need a) a GPU and b) an Ethereum client, Geth. CPU mining will be possible but too inefficient to hold any value.

At the moment, Geth only includes a CPU miner, and the team is testing a GPU miner branch, but this won't be part of Frontier.

The C++ implementation of Ethereum also offers a GPU miner, both as part of Eth (its CLI), AlethZero (its GUI) and EthMiner (the standalone miner).

***NOTE:** Ensure your blockchain is fully synchronised with the main chain before starting to mine, otherwise you will not be mining on the main chain.*

When you start up your ethereum node with `geth` it is not mining by default. To start it in mining mode, you use the `--mine` command line option. The `-minerthreads` parameter can be used to set the number parallel mining threads (defaulting to the total number of processor cores).

`geth --mine --minerthreads=4`

You can also start and stop CPU mining at runtime using the console. `miner.start` takes an optional parameter for the number of miner threads.

```
> miner.start(8)
true
> miner.stop()
true
```

Note that mining for real ether only makes sense if you are in sync with the network (since you mine on top of the consensus block). Therefore the eth blockchain downloader/synchroniser will delay mining until syncing is complete, and after that mining automatically starts unless you cancel your intention with `miner.stop()`.

In order to earn ether you must have your **etherbase** (or **coinbase**) address set. This etherbase defaults to your primary account. If you don't have an etherbase address, then `geth --mine` will not start up.

You can set your etherbase on the command line:

```
geth --etherbase 1 --mine  2>> geth.log // 1 is index: second account by creation order O
geth --etherbase '0xa4d8e9cae4d04b093aac82e6cd355b6b963fb7ff' --mine 2>> geth.log
```

You can reset your etherbase on the console too:

```
miner.setEtherbase(eth.accounts[2])
```

Note that your etherbase does not need to be an address of a local account, just an existing one.

There is an option to add extra Data (32 bytes only) to your mined blocks. By convention this is interpreted as a unicode string, so you can set your short vanity tag.

```
miner.setExtra("ΞTHΞЯSPHΞЯΞ")
...
debug.printBlock(131805)
BLOCK(be465b020fdbedc4063756f0912b5a89bbb4735bd1d1df84363e05ade0195cb1): Size: 531.00 B T
NoNonce: ee48752c3a0bfe3d85339451a5f3f411c21c8170353e450985e1faab0a9ac4cc
Header:
[
...
        Coinbase:           a4d8e9cae4d04b093aac82e6cd355b6b963fb7ff
        Number:             131805
        Extra:              ΞTHΞЯSPHΞЯΞ
...
}
```

See also this proposal

You can check your hashrate with miner.hashrate, the result is in H/s (Hash operations per second).

```
> miner.hashrate
712000
```

After you successfully mined some blocks, you can check the ether balance of your etherbase account. Now assuming your etherbase is a local account:

```
> eth.getBalance(eth.coinbase).toNumber();
'34698870000000'
```

In order to spend your earnings on gas to transact, you will need to have this account unlocked.

```
> personal.unlockAccount(eth.coinbase)
Password
true
```

You can check which blocks are mined by a particular miner (address) with the following code snippet on the console:

```
function minedBlocks(lastn, addr) {
  addrs = [];
  if (!addr) {
    addr = eth.coinbase
  }
  limit = eth.blockNumber - lastn
  for (i = eth.blockNumber; i >= limit; i--) {
    if (eth.getBlock(i).miner == addr) {
      addrs.push(i)
    }
  }
  return addrs
}
// scans the last 1000 blocks and returns the blocknumbers of blocks mined by your coinba
// (more precisely blocks the mining reward for which is sent to your coinbase).
minedBlocks(1000, eth.coinbase);
//[352708, 352655, 352559]
```

Note that it will happen often that you find a block yet it never makes it to the canonical chain. This means when you locally include your mined block, the current state will show the mining reward credited to your account, however, after a while, the better chain is discovered and we switch to a chain in which your block is not included and therefore no mining reward is credited. Therefore it is quite possible that as a miner monitoring their coinbase balance will find that it may fluctuate quite a bit.

The logs show locally mined blocks confirmed after 5 blocks. At the moment you may find it easier and faster to generate the list of your mined blocks from these logs.

Mining success depends on the set block difficulty. Block difficulty dynamically adjusts each block in order to regulate the network hashing power to produce a 12 second blocktime. Your chances of finding a block therefore follows from your hashrate relative to difficulty. The time you need to wait you are expected to find a block can be estimated with the following code:

INCORRECT...CHECKING

```
etm = eth.getBlock("latest").difficulty/miner.hashrate; // estimated time in seconds
Math.floor(etm / 3600.) + "h " + Math.floor((etm % 3600)/60) + "m " +  Math.floor(etm % 6
// 1h 3m 30s
```

Given a difficulty of 3 billion, a typical CPU with 800KH/s is expected to find a block every ....?

GPU mining

Hardware

The algorithm is memory hard and in order to fit the DAG into memory, it needs 1-2GB of RAM on each GPU. If you get `Error GPU mining. GPU memory fragmentation?` you havent got enough memory.

The GPU miner is implemented in OpenCL, so AMD GPUs will be 'faster' than same-category NVIDIA GPUs.

ASICs and FPGAs are relatively inefficient and therefore discouraged.

To get openCL for your chipset and platform, try:

- AMD SDK openCL
- NVIDIA CUDA openCL

On Ubuntu

AMD

- http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing
- http://developer.amd.com/tools-and-sdks/graphics-development/display-library-adl-sdk/

download: `ADL_SDK8.zip` and `AMD-APP-SDK-v2.9-1.599.381-GA-linux64.sh`

```
./AMD-APP-SDK-v2.9-1.599.381-GA-linux64.sh
ln -s /opt/AMDAPPSDK-2.9-1 /opt/AMDAPP
ln -s /opt/AMDAPP/include/CL /usr/include
ln -s /opt/AMDAPP/lib/x86_64/* /usr/lib/
ldconfig
reboot
```

```
apt-get install fglrx-updates
// wget, tar, opencl
sudo aticonfig --adapter=all --initial
sudo aticonfig --list-adapters
* 0. 01:00.0 AMD Radeon R9 200 Series

* - Default adapter
```

Nvidia

The following instructions are, for the most part, relevant to any system with Ubuntu 14.04 and a Nvidia GPU. Setting up an EC2 instance for mining

On MacOSx

```
wget http://developer.download.nvidia.com/compute/cuda/7_0/Prod/local_installers/cuda_7.0
sudo installer -pkg ~/Desktop/cuda_7.0.29_mac.pkg -target /
brew update
brew tap ethereum/ethereum
brew reinstall cpp-ethereum --with-gpu-mining --devel --headless --build-from-source
```

You check your cooling status:

```
aticonfig --adapter=0 --od-gettemperature
```

Mining Software

The official Frontier release of `geth` only supports a CPU miner natively. We are working on a GPU miner, but it may not be available for the Frontier release. Geth however can be used in conjunction with `ethminer`, using the standalone miner as workers and `geth` as scheduler communicating via JSON-RPC.

The C++ implementation of Ethereum (not officially released) however has a GPU miner. It can be used from `eth`, `AlethZero` (GUI) and `ethMiner` (the standalone miner).

You can install this via ppa on linux, brew tap on MacOS or from source.

On MacOS:

```
brew install cpp-ethereum --with-gpu-mining --devel --build-from-source
```

On Linux:

```
apt-get install cpp-ethereum
```

On Windows: https://github.com/ethereum/cpp-ethereum/wiki/Building-on-Windows

GPU mining with ethminer

To mine with `eth`:

```
eth -m on -G -a <coinbase> -i -v 8 //
```

To install `ethminer` from source:

```
cd cpp-ethereum
cmake -DETHASHCL=1 -DGUI=0
make -j4
make install
```

To set up GPU mining you need a coinbase account. It can be an account created locally or remotely.

Using ethminer with geth

```
geth account new
geth --rpc --rpccorsdomain localhost 2>> geth.log &
ethminer -G  // -G for GPU, -M for benchmark
tail -f geth.log
```

`ethminer` communicates with geth on port 8545 (the default RPC port in geth). You can change this by giving the `--rpcport` option to `geth`. Ethminer will find get on any port. Note that you need to set the CORS header with `--rpccorsdomain localhost`. You can also set port on `ethminer` with `-F http://127.0.0.1:3301`. Setting the ports is necessary if you want several instances mining on the same computer, although this is somewhat pointless. If you are testing on a private cluster, we recommend you use CPU mining instead.

Also note that you do **not** need to give `geth` the `--mine` option or start the miner in the console unless you want to do CPU mining on TOP of GPU mining.

If the default for `ethminer` does not work try to specify the OpenCL device with: `--opencl-device X` where X is 0, 1, 2, etc. When running `ethminer` with `-M` (benchmark), you should see something like:

```
Benchmarking on platform: { "platform": "NVIDIA CUDA", "device": "GeForce GTX 750 Ti", "v


Benchmarking on platform: { "platform": "Apple", "device": "Intel(R) Xeon(R) CPU E5-1620
```

To debug `geth`:

```
geth  --rpccorsdomain "localhost" --verbosity 6 2>> geth.log
```

To debug the miner:

```
make -DCMAKE_BUILD_TYPE=Debug -DETHASHCL=1 -DGUI=0
gdb --args ethminer -G -M
```

Note hashrate info is not available in `geth` when GPU mining. Check your hashrate with `ethminer`, `miner.hashrate` will always report 0.

ethminer and eth

`ethminer` can be used in conjunction with `eth` via rpc

```
eth -i -v 8 -j // -j for rpc
ethminer -G -M // -G for GPU, -M for benchmark
tail -f geth.log
```

or you can use `eth` to GPU mine by itself:

```
eth -m on -G -a <coinbase> -i -v 8 //
```

Interfaces

- Javascript Console: `geth` can be launched with an interactive console, that provides a javascript runtime environment exposing a javascript API to interact with your node. Javascript Console API includes the `web3` javascript Đapp API as well as an additional admin API.
- JSON-RPC server: `geth` can be launched with a json-rpc server that exposes the JSON-RPC API
- Command line options documents command line parameters as well as subcommands.

Command line options

```
geth [global options] command [command options] [arguments...]

VERSION:
   1.0.0

COMMANDS:
   recover      attempts to recover a corrupted database by setting a new block by number
   blocktest    loads a block test file
   import       import a blockchain file
   export       export blockchain into file
   upgradedb    upgrade chainblock database
   removedb     Remove blockchain and state databases
   dump         dump a specific block from storage
   monitor      Geth Monitor: node metrics monitoring and visualization
   makedag      generate ethash dag (for testing)
   version      print ethereum version numbers
   wallet       ethereum presale wallet
   account      manage accounts
   console      Geth Console: interactive JavaScript environment
   attach       Geth Console: interactive JavaScript environment (connect to node)
   js           executes the given JavaScript files in the Geth JavaScript VM
   help         Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --identity                                                      Custom node name
   --unlock                                                        Unlock the accoun
   --password                                                      Path to password
   --genesis                                                       Inserts/Overwrite
   --bootnodes                                                     Space-separated e
   --datadir "/Users/tron/Library/Ethereum"                        Data directory to
   --blockchainversion "3"                                         Blockchain versio
   --jspath "."                                                    JS library path t
   --port "30303"                                                  Network listening
   --maxpeers "25"                                                 Maximum number of
   --maxpendpeers "0"                                              Maximum number of
   --etherbase "0"                                                 Public address fo
   --gasprice "1000000000000"                                      Sets the minimal
   --minerthreads "8"                                              Number of miner t
   --mine                                                          Enable mining
   --autodag                                                       Enable automatic
   --nat "any"                                                     NAT port mapping
   --natspec                                                       Enable NatSpec co
   --nodiscover                                                    Disables the peer
   --nodekey                                                       P2P node key file
   --nodekeyhex                                                    P2P node key as h
   --rpc                                                           Enable the JSON-R
   --rpcaddr "127.0.0.1"                                           Listening address
```

```
     --rpcport "8545"                                                    Port on which the
     --rpcapi "db,eth,net,web3"                                          Specify the API's
     --ipcdisable                                                        Disable the IPC-R
     --ipcapi "admin,db,eth,debug,miner,net,shh,txpool,personal,web3"     Specify the API's
     --ipcpath "/Users/tron/Library/Ethereum/geth.ipc"                   Filename for IPC
     --exec                                                              Execute javascrip
     --shh                                                               Enable whisper
     --vmdebug                                                           Virtual Machine d
     --networkid "1"                                                     Network Id (integ
     --rpccorsdomain                                                     Domain on which t
     --verbosity "3"                                                     Logging verbosity
     --backtrace_at ":0"                                                 If set to a file
     --logtostderr                                                       Logs are written
     --vmodule ""                                                        The syntax of the
     --logfile                                                           Send log output t
     --logjson                                                           Send json structu
     --pprof                                                             Enable the profil
     --pprofport "6060"                                                  Port on which the
     --metrics                                                           Enables metrics c
     --solc "solc"                                                       solidity compiler
     --gpomin "1000000000000"                                            Minimum suggested
     --gpomax "100000000000000"                                          Maximum suggested
     --gpofull "80"                                                      Full block thresh
     --gpobasedown "10"                                                  Suggested gas pri
     --gpobaseup "100"                                                   Suggested gas pri
     --gpobasecf "110"                                                   Suggested gas pri
     --help, -h                                                          show help
```

Note that the default for datadir is platform-specific. See backup & restore for more information.

Examples

Accounts

See Account management

Import ether presale wallet into your node (prompts for password):

```
geth wallet import /path/to/my/etherwallet.json
```

Import an EC privatekey into an ethereum account (prompts for password):

```
geth account import /path/to/key.prv
```

Geth JavaScript Runtime Environment

See Geth javascript console

Bring up the geth javascript console:

```
geth --verbosity 5 --jspath /mydapp/js console 2>> /path/to/logfile
```

Execute `test.js` javascript using js API and log Debug-level messages to `/path/to/logfile`:

```
geth --verbosity 6 js test.js  2>> /path/to/logfile
```

Import/export chains and dump blocks

Import a blockchain from file:

```
geth import blockchain.bin
```

Upgrade chainblock database

When the consensus algorithm is changed blocks in the blockchain must be reimported with the new algorithm. Geth will inform the user with instructions when and how to do this when it's necessary.

```
geth upgradedb
```

Mining and networking

Start two mining nodes using different data directories listening on ports 30303 and 30304, respectively:

```
geth --mine --minerthreads 4 --datadir /usr/local/share/ethereum/30303 --port 30303
geth --mine --minerthreads 4 --datadir /usr/local/share/ethereum/30304 --port 30304
```

Start an rpc client on port 8000:

```
geth --rpc=true --rpcport 8000 --rpccorsdomain '"*"'
```

Launch the client without network:

```
geth --maxpeers 0 --nodiscover --networdid 3301 js justwannarunthis.js
```

Resetting the blockchain

In the datadir, delete the blockchain directory. For an example above:

```
rm -rf /usr/local/share/ethereum/30303/blockchain
```

Sample usage in testing environment

The lines below are meant only for test network and safe environments for non-interactive scripted use.

```
geth --datadir /tmp/eth/42 --password <(echo -n notsosecret) account new 2>> /tmp/eth/42.
geth --datadir /tmp/eth/42 --port 30342  js <(echo 'console.log(admin.nodeInfo().NodeUrl)
geth --datadir /tmp/eth/42 --port 30342 --password <(echo -n notsosecret) --unlock primar
```

Attach

Attach a console to a running geth instance. By default this happens over IPC over the default IPC endpoint but when necessary a custom endpoint could be specified:

```
geth attach ipc:/some/path
geth attach rpc:http://host:8545
```

Alternative ways to set flags

WARNING: This is not available for the latest frontier.

The same flags can be set via config file (by default `<datadir>/conf.ini`) as well as environment variables.

Precedence: default < config file < environment variables < command line

JSON RPC API

JSON is a lightweight data-interchange format. It can represent numbers, strings, ordered sequences of values, and collections of name/value pairs.

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. Primarily this specification defines several data structures and the rules around their processing. It is transport agnostic in that the concepts can be used within the same process, over sockets, over HTTP, or in many various message passing environments. It uses JSON (RFC 4627) as data format.

JavaScript API

To talk to an ethereum node from inside a JavaScript application use the web3.js library, which gives an convenient interface for the RPC methods. See the JavaScript API for more.

JSON-RPC Endpoint

Default JSON-RPC endpoints:

```
C++: http://localhost:8545
Go: http://localhost:8545
Py: http://localhost:4000
```

Go

You can start the HTTP JSON-RPC with the `--rpc` flag

```
geth --rpc
```

change the default port (8545) and listing address (localhost) with:

```
geth --rpc --rpcaddr <ip> --rpcport <portnumber>
```

If accessing the RPC from a browser, CORS will need to be enabled with the appropriate domain set. Otherwise, JavaScript calls are limit by the same-origin policy and requests will fail:

```
geth --rpc --rpccorsdomain "http://localhost:3000"
```

The JSON RPC can also be started from the geth console using the `admin.startRPC(addr, port)` command.

C++

You can start it by running `eth` application with `-j` option:

```
./eth -j
```

You can also specify JSON-RPC port (default is 8545):

```
./eth -j --json-rpc-port 8079
```

Python

In python the JSONRPC server is currently started by default and listens on `127.0.0.1:4000`

You can change the port and listen address by giving a config option.

`pyethapp -c jsonrpc.listen_port=4002 -c jsonrpc.listen_host=127.0.0.2 run`

JSON-RPC support

	cpp-ethereum	go-ethereum	py-ethereum
JSON-RPC 1.0	✓		
JSON-RPC 2.0	✓	✓	✓
Batch requests	✓	✓	✓
HTTP	✓	✓	✓

Output HEX values

At present there are two key datatypes that are passed over JSON: unformatted byte arrays and quantities. Both are passed with a hex encoding, however with different requirements to formatting:

When encoding **QUANTITIES** (integers, numbers): encode as hex, prefix with "0x", the most compact representation (slight exception: zero should be represented as "0x0"). Examples:

- 0x41 (65 in decimal)
- 0x400 (1024 in decimal)
- WRONG: 0x (should always have at least one digit - zero is "0x0")
- WRONG: 0x0400 (no leading zeroes allowed)
- WRONG: ff (must be prefixed 0x)

When encoding **UNFORMATTED DATA** (byte arrays, account addresses, hashes, bytecode arrays): encode as hex, prefix with "0x", two hex digits per byte. Examples:

- 0x41 (size 1, "A")
- 0x004200 (size 3, "\0B\0")
- 0x (size 0, "")
- WRONG: 0xf0f0f (must be even number of digits)
- WRONG: 004200 (must be prefixed 0x)

Currently cpp-ethereum and go-ethereum provides JSON-RPC communication only over http.

The default block parameter

The following methods have a extra default block parameter:

- eth_getBalance
- eth_getCode
- eth_getTransactionCount
- eth_getStorageAt
- eth_call

When requests are made that act on the state of ethereum, the last default block parameter determines the height of the block.

The following options are possible for the defaultBlock parameter:

- `HEX String` - an integer block number
- `String "earliest"` for the earliest/genesis block
- `String "latest"` - for the latest mined block
- `String "pending"` - for the pending state/transactions

JSON-RPC methods

- web3_clientVersion

JSON RPC API Reference

web3_clientVersion

Returns the current client version.

Parameters

none

Returns

`String` - The current client version

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc":"2.0",
  "result": "Mist/v0.9.3/darwin/go1.4.1"
}
```

web3_sha3

Returns Keccak-256 (*not* the standardized SHA3-256) of the given data.

Parameters

1. `String` - the data to convert into a SHA3 hash

```
params: [
  '0x68656c6c6f20776f726c64'
]
```

Returns

`DATA` - The SHA3 result of the given string.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"web3_sha3","params":["0x68656c6c6f20776f7

// Result
{
  "id":64,
  "jsonrpc": "2.0",
  "result": "0x47173285a8d7341e5e972fc677286384f802f8ef42a5ec5f03bbfa254cb01fad"
}
```

net_version

Returns the current network protocol version.

Parameters

none

Returns

`String` - The current network protocol version

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"net_version","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc": "2.0",
  "result": "59"
}
```

net_listening

Returns `true` if client is actively listening for network connections.

Parameters

none

Returns

`Boolean` - `true` when listening, otherwise `false` .

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"net_listening","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc":"2.0",
  "result":true
}
```

net_peerCount

Returns number of peers currenly connected to the client.

Parameters

none

Returns

`QUANTITY` - integer of the number of connected peers.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"net_peerCount","params":[],"id":74}'

// Result
{
  "id":74,
  "jsonrpc": "2.0",
  "result": "0x2" // 2
}
```

---

eth_protocolVersion

Returns the current ethereum protocol version.

Parameters

none

Returns

`String` - The current ethereum protocol version

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_protocolVersion","params":[],"id":67}

// Result
{
  "id":67,
  "jsonrpc": "2.0",
  "result": "54"
}
```

eth_syncing

Returns an object object with data about the sync status or FALSE.

Parameters

none

Returns

`Object|Boolean` , An object with sync status data or `FALSE` , when not syncing:

- `startingBlock` : `QUANTITY` - The block at which the import started (will only be reset, after the sync reached his head)
- `currentBlock` : `QUANTITY` - The current block, same as eth_blockNumber
- `highestBlock` : `QUANTITY` - The estimated highest block

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_isSyncing","params":[],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": {
    startingBlock: '0x384',
    currentBlock: '0x386',
    highestBlock: '0x454'
  }
}
// Or when not syncing
{
  "id":1,
  "jsonrpc": "2.0",
  "result": false
}
```

eth_coinbase

Returns the client coinbase address.

Parameters

none

Returns

`DATA` , 20 bytes - the current coinbase address.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_coinbase","params":[],"id":64}'

// Result
{
  "id":64,
  "jsonrpc": "2.0",
  "result": "0x407d73d8a49eeb85d32cf465507dd71d507100c1"
}
```

eth_mining

Returns `true` if client is actively mining new blocks.

Parameters

none

Returns

`Boolean` - returns `true` of the client is mining, otherwise `false` .

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_mining","params":[],"id":71}'

// Result
{
  "id":71,
  "jsonrpc": "2.0",
  "result": true
}
```

---

eth_hashrate

Returns the number of hashes per second that the node is mining with.

Parameters

none

Returns

`QUANTITY` - number of hashes per second.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_hashrate","params":[],"id":71}'

// Result
{
  "id":71,
  "jsonrpc": "2.0",
  "result": "0x38a"
}
```

eth_gasPrice

Returns the current price per gas in wei.

Parameters

none

Returns

`QUANTITY` - integer of the current gas price in wei.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":73}'

// Result
{
  "id":73,
  "jsonrpc": "2.0",
  "result": "0x09184e72a000" // 10000000000000
}
```

eth_accounts

Returns a list of addresses owned by client.

Parameters

none

Returns

`Array of DATA` , 20 Bytes - addresses owned by the client.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_accounts","params":[],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": ["0x407d73d8a49eeb85d32cf465507dd71d507100c1"]
}
```

eth_blockNumber

Returns the number of most recent block.

Parameters

none

Returns

`QUANTITY` - integer of the current block number the client is on.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":83}'

// Result
{
  "id":83,
  "jsonrpc": "2.0",
  "result": "0x4b7" // 1207
}
```

eth_getBalance

Returns the balance of the account of given address.

Parameters

1. `DATA` , 20 Bytes - address to check for balance.
2. `QUANTITY|TAG` - integer block number, or the string `"latest"` , `"earliest"` or `"pending"` , see the default block parameter

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1',
   'latest'
]
```

Returns

`QUANTITY` - integer of the current balance in wei.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBalance","params":["0x407d73d8a49e

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x0234c8a3397aab58" // 158972490234375000
}
```

eth_getStorageAt

Returns the value from a storage position at a given address.

Parameters

1. `DATA` , 20 Bytes - address of the storage.
2. `QUANTITY` - integer of the position in the storage.
3. `QUANTITY|TAG` - integer block number, or the string `"latest"` , `"earliest"` or `"pending"` , see the default block parameter

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1',
   '0x0', // storage position at 0
   '0x2' // state at block number 2
]
```

Returns

`DATA` - the value at this storage position.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getStorageAt","params":["0x407d73d8a4

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x03"
}
```

eth_getTransactionCount

Returns the number of transactions *sent* from an address.

Parameters

1. `DATA` , 20 Bytes - address.
2. `QUANTITY|TAG` - integer block number, or the string `"latest"` , `"earliest"` or `"pending"` , see the default block parameter

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1',
   'latest' // state at the latest block
]
```

Returns

`QUANTITY` - integer of the number of transactions send from this address.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionCount","params":["0x407

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

eth_getBlockTransactionCountByHash

Returns the number of transactions in a block from a block matching the given block hash.

Parameters

1. `DATA` , 32 Bytes - hash of a block

```
params: [
   '0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238'
]
```

Returns

`QUANTITY` - integer of the number of transactions in this block.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockTransactionCountByHash","para

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xb" // 11
}
```

eth_getBlockTransactionCountByNumber

Returns the number of transactions in a block from a block matching the given block number.

Parameters

1. `QUANTITY|TAG` - integer of a block number, or the string `"earliest"` , `"latest"` or `"pending"` , as in the default block parameter.

```
params: [
   '0xe8', // 232
]
```

Returns

`QUANTITY` - integer of the number of transactions in this block.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockTransactionCountByNumber","pa

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xa" // 10
}
```

eth_getUncleCountByBlockHash

Returns the number of uncles in a block from a block matching the given block hash.

Parameters

1. `DATA` , 32 Bytes - hash of a block

```
params: [
   '0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238'
]
```

Returns

`QUANTITY` - integer of the number of uncles in this block.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleCountByBlockHash","params":["

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

eth_getUncleCountByBlockNumber

Returns the number of uncles in a block from a block matching the given block number.

Parameters

1. `QUANTITY` - integer of a block number, or the string "latest", "earliest" or "pending", see the default block parameter

```
params: [
   '0xe8', // 232
]
```

Returns

`QUANTITY` - integer of the number of uncles in this block.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleCountByBlockNumber","params":

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

eth_getCode

Returns code at a given address.

Parameters

1. `DATA` , 20 Bytes - address
2. `QUANTITY|TAG` - integer block number, or the string `"latest"` , `"earliest"` or `"pending"` , see the default block parameter

```
params: [
   '0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b',
   '0x2'  // 2
]
```

Returns

`DATA` - the code from the given address.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getCode","params":["0xa94f5374fce5edb

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x600160008035811a818181146012578301005b601b6001356025565b8060005260206000f2
}
```

eth_sign

Signs data with a given address.

Note the address to sign must be unlocked.

Parameters

1. `DATA` , 20 Bytes - address
2. `DATA` , Data to sign

Returns

`DATA` : Signed data

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_sign","params":["0xd1ade25ccd3d550a7e

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x2ac19db245478a06032e69cdbd2b54e648b78431d0a47bd1fbab18f79f820ba407466e37ad
}
```

eth_sendTransaction

Creates new message call transaction or a contract creation, if the data field contains code.

Parameters

1. `Object` - The transaction object
   - `from` : `DATA` , 20 Bytes - The address the transaction is send from.
   - `to` : `DATA` , 20 Bytes - (optional when creating new contract) The address the transaction is directed to.
   - `gas` : `QUANTITY` - (optional, default: 90000) Integer of the gas provided for the transaction execution. It will return unused gas.
   - `gasPrice` : `QUANTITY` - (optional, default: To-Be-Determined) Integer of the gasPrice used for each paid gas
   - `value` : `QUANTITY` - (optional) Integer of the value send with this transaction
   - `data` : `DATA` - (optional) The compiled code of a contract
   - `nonce` : `QUANTITY` - (optional) Integer of a nonce. This allows to overwrite your own pending transactions that use the same nonce.

```
params: [{
  "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
  "to": "0xd46e8dd67c5d32be8058bb8eb970870f072445675",
  "gas": "0x76c0", // 30400,
  "gasPrice": "0x9184e72a000", // 10000000000000
  "value": "0x9184e72a", // 2441406250
  "data": "0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072
}]
```

Returns

`DATA` , 32 Bytes - the transaction hash, or the zero hash if the transaction is not yet available.

Use eth_getTransactionReceipt to get the contract address, after the transaction was mined, when you created a contract.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_sendTransaction","params":[{see above

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
}
```

eth_sendRawTransaction

Creates new message call transaction or a contract creation for signed transactions.

Parameters

1. `Object` - The transaction object
    - `data` : `DATA` , The signed transaction data.

```
params: [{
  "data": "0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072
}]
```

Returns

`DATA` , 32 Bytes - the transaction hash, or the zero hash if the transaction is not yet available.

Use eth_getTransactionReceipt to get the contract address, after the transaction was mined, when you created a contract.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_sendRawTransaction","params":[{see ab

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
}
```

eth_call

Executes a new message call immediately without creating a transaction on the block chain.

Parameters

1. `Object` - The transaction call object
   - `from` : `DATA` , 20 Bytes - (optional) The address the transaction is send from.
   - `to` : `DATA` , 20 Bytes - The address the transaction is directed to.
   - `gas` : `QUANTITY` - (optional) Integer of the gas provided for the transaction execution. eth_call consumes zero gas, but this parameter may be needed by some executions.
   - `gasPrice` : `QUANTITY` - (optional) Integer of the gasPrice used for each paid gas
   - `value` : `QUANTITY` - (optional) Integer of the value send with this transaction
   - `data` : `DATA` - (optional) The compiled code of a contract
2. `QUANTITY|TAG` - integer block number, or the string `"latest"` , `"earliest"` or `"pending"` , see the default block parameter

Returns

`DATA` - the return value of executed contract.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_call","params":[{see above}],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x0"
}
```

eth_estimateGas

Makes a call or transaction, which won't be added to the blockchain and returns the used gas, which can be used for estimating the used gas.

Parameters

See eth_call parameters, expect that all properties are optional.

Returns

`QUANTITY` - the amount of gas used.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_estimateGas","params":[{see above}],"

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x5208" // 21000
}
```

eth_getBlockByHash

Returns information about a block by hash.

Parameters

1. `DATA` , 32 Bytes - Hash of a block.
2. `Boolean` - If `true` it returns the full transaction objects, if `false` only the hashes of the transactions.

```
params: [
   '0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331',
   true
]
```

Returns

`Object` - A block object, or `null` when no block was found:

- `number` : `QUANTITY` - the block number. `null` when its pending block.
- `hash` : `DATA` , 32 Bytes - hash of the block. `null` when its pending block.
- `parentHash` : `DATA` , 32 Bytes - hash of the parent block.
- `nonce` : `DATA` , 8 Bytes - hash of the generated proof-of-work. `null` when its pending block.
- `sha3Uncles` : `DATA` , 32 Bytes - SHA3 of the uncles data in the block.
- `logsBloom` : `DATA` , 256 Bytes - the bloom filter for the logs of the block. `null` when its

pending block.

- `transactionsRoot` : `DATA` , 32 Bytes - the root of the transaction trie of the block.
- `stateRoot` : `DATA` , 32 Bytes - the root of the final state trie of the block.
- `receiptsRoot` : `DATA` , 32 Bytes - the root of the receipts trie of the block.
- `miner` : `DATA` , 20 Bytes - the address of the beneficiary to whom the mining rewards were given.
- `difficulty` : `QUANTITY` - integer of the difficulty for this block.
- `totalDifficulty` : `QUANTITY` - integer of the total difficulty of the chain until this block.
- `extraData` : `DATA` - the "extra data" field of this block.
- `size` : `QUANTITY` - integer the size of this block in bytes.
- `gasLimit` : `QUANTITY` - the maximum gas allowed in this block.
- `gasUsed` : `QUANTITY` - the total used gas by all transactions in this block.
- `timestamp` : `QUANTITY` - the unix timestamp for when the block was collated.
- `transactions` : `Array` - Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter.
- `uncles` : `Array` - Array of uncle hashes.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockByHash","params":["0xe670ec64

// Result
{
"id":1,
"jsonrpc":"2.0",
"result": {
    "number": "0x1b4", // 436
    "hash": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331",
    "parentHash": "0x9646252be9520f6e71339a8df9c55e4d7619deeb018d2a3f2d21fc165dde5eb5",
    "nonce": "0xe04d296d2460cfb8472af2c5fd05b5a214109c25688d3704aed5484f9a7792f2",
    "sha3Uncles": "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
    "logsBloom": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331",
    "transactionsRoot": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b4
    "stateRoot": "0xd5855eb08b3387c0af375e9cdb6acfc05eb8f519e419b874b6ff2ffda7ed1dff",
    "miner": "0x4e65fda2159562a496f9f3522f89122a3088497a",
    "difficulty": "0x027f07", // 163591
    "totalDifficulty":  "0x027f07", // 163591
    "extraData": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "size":  "0x027f07", // 163591
    "gasLimit": "0x9f759", // 653145
    "minGasPrice": "0x9f759", // 653145
    "gasUsed": "0x9f759", // 653145
    "timestamp": "0x54e34e8e" // 1424182926
    "transactions": [{...},{ ... }]
    "uncles": ["0x1606e5...", "0xd5145a9..."]
  }
}
```

eth_getBlockByNumber

Returns information about a block by block number.

Parameters

1. `QUANTITY|TAG` - integer of a block number, or the string `"earliest"` , `"latest"` or `"pending"` , as in the default block parameter.
2. `Boolean` - If `true` it returns the full transaction objects, if `false` only the hashes of the transactions.

```
params: [
   '0x1b4', // 436
   true
]
```

Returns

See eth_getBlockByHash

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":["0x1b4",
```

Result see eth_getBlockByHash

eth_getTransactionByHash

Returns the information about a transaction requested by transaction hash.

Parameters

1. `DATA` , 32 Bytes - hash of a transaction

```
params: [
   "0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238"
]
```

Returns

`Object` - A transaction object, or `null` when no transaction was found:

- `hash` : `DATA` , 32 Bytes - hash of the transaction.
- `nonce` : `QUANTITY` - the number of transactions made by the sender prior to this one.
- `blockHash` : `DATA` , 32 Bytes - hash of the block where this transaction was in. `null` when its pending.
- `blockNumber` : `QUANTITY` - block number where this transaction was in. `null` when its pending.
- `transactionIndex` : `QUANTITY` - integer of the transactions index position in the block. `null` when its pending.
- `from` : `DATA` , 20 Bytes - address of the sender.
- `to` : `DATA` , 20 Bytes - address of the receiver. `null` when its a contract creation transaction.
- `value` : `QUANTITY` - value transferred in Wei.
- `gasPrice` : `QUANTITY` - gas price provided by the sender in Wei.
- `gas` : `QUANTITY` - gas provided by the sender.
- `input` : `DATA` - the data send along with the transaction.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionByHash","params":["0xb9

// Result
{
"id":1,
"jsonrpc":"2.0",
"result": {
    "hash":"0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b",
    "nonce":"0x",
    "blockHash": "0xbeab0aa2411b7ab17f30a99d3cb9c6ef2fc5426d6ad6fd9e2a26a6aed1d1055b",
    "blockNumber": "0x15df", // 5599
    "transactionIndex":  "0x1", // 1
    "from":"0x407d73d8a49eeb85d32cf465507dd71d507100c1",
    "to":"0x85h43d8a49eeb85d32cf465507dd71d507100c1",
    "value":"0x7f110" // 520464
    "gas": "0x7f110" // 520464
    "gasPrice":"0x09184e72a000",
    "input":"0x603880600c6000396000f300603880600c6000396000f3603880600c6000396000f360",
  }
}
```

eth_getTransactionByBlockHashAndIndex

Returns information about a transaction by block hash and transaction index position.

Parameters

1. `DATA` , 32 Bytes - hash of a block.
2. `QUANTITY` - integer of the transaction index position.

```
params: [
   '0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331',
   '0x0' // 0
]
```

Returns

See eth_getBlockByHash

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionByBlockHashAndIndex","p
```

Result see eth_getTransactionByHash

eth_getTransactionByBlockNumberAndIndex

Returns information about a transaction by block number and transaction index position.

Parameters

1. `QUANTITY|TAG` - a block number, or the string `"earliest"` , `"latest"` or `"pending"` , as in the default block parameter.
2. `QUANTITY` - the transaction index position.

```
params: [
   '0x29c', // 668
   '0x0' // 0
]
```

Returns

See eth_getBlockByHash

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionByBlockNumberAndIndex",
```

Result see eth_getTransactionByHash

eth_getTransactionReceipt

Returns the receipt of a transaction by transaction hash.

Note That the receipt is not available for pending transactions.

Parameters

1. `DATA` , 32 Bytes - hash of a transaction

```
params: [
   '0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238'
]
```

Returns

`Object` - A transaction receipt object, or `null` when no receipt was found:

- `transactionHash` : `DATA` , 32 Bytes - hash of the transaction.
- `transactionIndex` : `QUANTITY` - integer of the transactions index position in the block.
- `blockHash` : `DATA` , 32 Bytes - hash of the block where this transaction was in.
- `blockNumber` : `QUANTITY` - block number where this transaction was in.
- `cumulativeGasUsed` : `QUANTITY` - The total amount of gas used when this transaction was executed in the block.
- `gasUsed` : `QUANTITY` - The amount of gas used by this specific transaction alone.
- `contractAddress` : `DATA` , 20 Bytes - The contract address created, if the transaction was a contract creation, otherwise `null` .
- `logs` : `Array` - Array of log objects, which this transaction generated.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionReceipt","params":["0xb

// Result
{
"id":1,
"jsonrpc":"2.0",
"result": {
     transactionHash: '0xb903239f8543d04b5dc1ba6579132b143087c68db1b2168786408fcbce568238
     transactionIndex:  '0x1', // 1
     blockNumber: '0xb', // 11
     blockHash: '0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b',
     cumulativeGasUsed: '0x33bc', // 13244
     gasUsed: '0x4dc', // 1244
     contractAddress: '0xb60e8dd61c5d32be8058bb8eb970870f07233155' // or null, if none wa
     logs: [{
         // logs as returned by getFilterLogs, etc.
     }, ...]
  }
}
```

eth_getUncleByBlockHashAndIndex

Returns information about a uncle of a block by hash and uncle index position.

Parameters

1. `DATA` , 32 Bytes - hash a block.
2. `QUANTITY` - the uncle's index position.

```
params: [
   '0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b',
   '0x0' // 0
]
```

Returns

See eth_getBlockByHash

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleByBlockHashAndIndex","params"
```

Result see eth_getBlockByHash

Note: An uncle doesn't contain individual transactions.

eth_getUncleByBlockNumberAndIndex

Returns information about a uncle of a block by number and uncle index position.

Parameters

1. `QUANTITY|TAG` - a block number, or the string `"earliest"` , `"latest"` or `"pending"` , as in the default block parameter.
2. `QUANTITY` - the uncle's index position.

```
params: [
   '0x29c', // 668
   '0x0' // 0
]
```

Returns

See eth_getBlockByHash

Note: An uncle doesn't contain individual transactions.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleByBlockNumberAndIndex","param
```

Result see eth_getBlockByHash

eth_getCompilers

Returns a list of available compilers in the client.

Parameters

none

Returns

`Array` - Array of available compilers.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getCompilers","params":[],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": ["solidity", "lll", "serpent"]
}
```

eth_compileSolidity

Returns compiled solidity code.

Parameters

1. `String` - The source code.

```
params: [
   "contract test { function multiply(uint a) returns(uint d) {   return a * 7;   } }",
]
```

Returns

`DATA` - The compiled source code.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_compileSolidity","params":["contract

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": {
      "code": "0x605880600c6000396000f3006000357c0100000000000000000000000000000000000
      "info": {
        "source": "contract test {\n   function multiply(uint a) constant returns(uint d)
        "language": "Solidity",
        "languageVersion": "0",
        "compilerVersion": "0.9.19",
        "abiDefinition": [
          {
            "constant": true,
            "inputs": [
              {
                "name": "a",
                "type": "uint256"
              }
            ],
            "name": "multiply",
            "outputs": [
              {
                "name": "d",
                "type": "uint256"
              }
            ],
            "type": "function"
          }
        ],
        "userDoc": {
          "methods": {}
        },
        "developerDoc": {
          "methods": {}
        }
      }

}
```

eth_compileLLL

Returns compiled LLL code.

Parameters

1. `String` - The source code.

```
params: [
   "(returnlll (suicide (caller)))",
]
```

Returns

`DATA` - The compiled source code.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_compileSolidity","params":["(returnll

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114601857005b
}
```

eth_compileSerpent

Returns compiled serpent code.

Parameters

1. `String` - The source code.

```
params: [
   "/* some serpent */",
]
```

Returns

`DATA` - The compiled source code.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_compileSerpent","params":["/* some se

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114601857005b
}
```

eth_newFilter

Creates a filter object, based on filter options, to notify when the state changes (logs). To check if the state has changed, call eth_getFilterChanges.

Parameters

1. `Object` - The filter options:
   - `fromBlock` : `QUANTITY|TAG` - (optional, default: `"latest"`) Integer block number, or `"latest"` for the last mined block or `"pending"`, `"earliest"` for not yet mined transactions.
   - `toBlock` : `QUANTITY|TAG` - (optional, default: `"latest"`) Integer block number, or `"latest"` for the last mined block or `"pending"`, `"earliest"` for not yet mined transactions.
   - `address` : `DATA|Array`, 20 Bytes - (optional) Contract address or a list of addresses from which logs should originate.
   - `topics` : `Array of DATA`, - (optional) Array of 32 Bytes `DATA` topics.

```
params: [{
  "fromBlock": "0x1",
  "toBlock": "0x2",
  "address": "0x8888f1f195afa192cfee860698584c030f4c9db1",
  "topics": ["0x000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b"]
}]
```

Returns

`QUANTITY` - A filter id.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_newFilter","params":[{"topics":["0x12

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

eth_newBlockFilter

Creates a filter in the node, to notify when a new block arrives. To check if the state has changed, call eth_getFilterChanges.

Parameters

None

Returns

`QUANTITY` - A filter id.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_newBlockFilter","params":[],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":  "2.0",
  "result": "0x1" // 1
}
```

eth_newPendingTransactionFilter

Creates a filter in the node, to notify when new pending transactions arrive. To check if the state has changed, call eth_getFilterChanges.

Parameters

None

Returns

`QUANTITY` - A filter id.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_newPendingTransactionFilter","params"

// Result
{
  "id":1,
  "jsonrpc":  "2.0",
  "result": "0x1" // 1
}
```

eth_uninstallFilter

Uninstalls a filter with given id. Should always be called when watch is no longer needed. Additonally Filters timeout when they aren't requested with eth_getFilterChanges for a period of time.

Parameters

1. `QUANTITY` - The filter id.

```
params: [
  "0xb" // 11
]
```

Returns

`Boolean` - `true` if the filter was successfully uninstalled, otherwise `false` .

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_uninstallFilter","params":["0xb"],"id

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": true
}
```

eth_getFilterChanges

Polling method for a filter, which returns an array of logs which occurred since last poll.

Parameters

1. `QUANTITY` - the filter id.

```
params: [
  "0x16" // 22
]
```

Returns

`Array` - Array of log objects, or an empty array if nothing has changed since last poll.

- For filters created with `eth_newBlockFilter` the return are block hashes (`DATA` , 32 Bytes), e.g. `["0x3454645634534..."]` .
- For filters created with `eth_newPendingTransactionFilter` the return are transaction hashes (`DATA` , 32 Bytes), e.g. `["0x6345343454645..."]` .
- For filters created with `eth_newFilter` logs are objects with following params:
    - `type` : `TAG` - `pending` when the log is pending. `mined` if log is already mined.
    - `logIndex` : `QUANTITY` - integer of the log index position in the block. `null` when its pending log.
    - `transactionIndex` : `QUANTITY` - integer of the transactions index position log was created from. `null` when its pending log.
    - `transactionHash` : `DATA` , 32 Bytes - hash of the transactions this log was created from. `null` when its pending log.
    - `blockHash` : `DATA` , 32 Bytes - hash of the block where this log was in. `null` when its pending. `null` when its pending log.
    - `blockNumber` : `QUANTITY` - the block number where this log was in. `null` when its pending. `null` when its pending log.

- `address` : `DATA` , 20 Bytes - address from which this log originated.
- `data` : `DATA` - contains one or more 32 Bytes non-indexed arguments of the log.
- `topics` : `Array of DATA` - Array of 0 to 4 32 Bytes `DATA` of indexed log arguments. (In *solidity*: The first topic is the *hash* of the signature of the event (e.g. `Deposit(address,bytes32,uint256)`), except you declared the event with the `anonymous` specifier.)

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getFilterChanges","params":["0x16"],"

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": [{
    "logIndex": "0x1", // 1
    "blockNumber":"0x1b4" // 436
    "blockHash": "0x8216c5785ac562ff41e2dcfdf5785ac562ff41e2dcfdf829c5a142f1fccd7d",
    "transactionHash":  "0xdf829c5a142f1fccd7d8216c5785ac562ff41e2dcfdf5785ac562ff41e2dcf
    "transactionIndex": "0x0", // 0
    "address": "0x16c5785ac562ff41e2dcfdf829c5a142f1fccd7d",
    "data":"0x0000000000000000000000000000000000000000000000000000000000000000",
    "topics": ["0x59ebeb90bc63057b6515673c3ecf9438e5058bca0f92585014eced636878c9a5"]
    },{
      ...
    }]
}
```

eth_getFilterLogs

Returns an array of all logs matching filter with given id.

Parameters

1. `QUANTITY` - The filter id.

```
params: [
  "0x16" // 22
]
```

Returns

See eth_getFilterChanges

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getFilterLogs","params":["0x16"],"id"
```

Result see eth_getFilterChanges

eth_getLogs

Returns an array of all logs matching a given filter object.

Parameters

1. `Object` - the filter object, see eth_newFilter parameters.

```
params: [{
  "topics": ["0x000000000000000000000000a94f5374fce5edbc8e2a8697c15331677e6ebf0b"]
}]
```

Returns

See eth_getFilterChanges

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getLogs","params":[{"topics":["0x0000
```

Result see eth_getFilterChanges

eth_getWork

Returns the hash of the current block, the seedHash, and the boundary condition to be met ("target").

Parameters

none

Returns

`Array` - Array with the following properties:

1. `DATA` , 32 Bytes - current block header pow-hash
2. `DATA` , 32 Bytes - the seed hash used for the DAG.
3. `DATA` , 32 Bytes - the boundary condition ("target"), 2^256 / difficulty.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getWork","params":[],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": [
      "0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef",
      "0x5EED00000000000000000000000000005EED0000000000000000000000000000",
      "0xd1ff1c01710000000000000000000000d1ff1c01710000000000000000000000"
    ]
}
```

eth_submitWork

Used for submitting a proof-of-work solution.

Parameters

1. `DATA` , 8 Bytes - The nonce found (64 bits)
2. `DATA` , 32 Bytes - The header's pow-hash (256 bits)
3. `DATA` , 32 Bytes - The mix digest (256 bits)

```
params: [
  "0x0000000000000001",
  "0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef",
  "0xD1FE5700000000000000000000000000D1FE5700000000000000000000000000"
]
```

Returns

`Boolean` - returns `true` if the provided solution is valid, otherwise `false` .

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0", "method":"eth_submitWork", "params":["0x0000000000

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

eth_submitHashrate

Used for submitting mining hashrate.

Parameters

1. `Hashrate` , a hexadecimal string representation (32 bytes) of the hash rate
2. `ID` , String - A random hexadecimal(32 bytes) ID identifying the client

```
params: [
  "0x0000000000000000000000000000000000000000000000000000000000500000",
  "0x59daa26581d0acd1fce254fb7e85952f4c09d0915afd33d3886cd914bc7d283c"
]
```

Returns

`Boolean` - returns `true` if submitting went through succesfully and `false` otherwise.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0", "method":"eth_submitHashrate", "params":["0x000000

// Result
{
  "id":73,
  "jsonrpc":"2.0",
  "result": true
}
```

db_putString

Stores a string in the local database.

Note this function is deprecated and will be removed in the future.

Parameters

1. `String` - Database name.
2. `String` - Key name.
3. `String` - String to store.

```
params: [
  "testDB",
  "myKey",
  "myString"
]
```

Returns

`Boolean` - returns `true` if the value was stored, otherwise `false` .

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"db_putString","params":["testDB","myKey",

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

db_getString

Returns string from the local database.

Note this function is deprecated and will be removed in the future.

Parameters

1. `String` - Database name.
2. `String` - Key name.

```
params: [
  "testDB",
  "myKey",
]
```

Returns

`String` - The previously stored string.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"db_getString","params":["testDB","myKey"]

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": "myString"
}
```

db_putHex

Stores binary data in the local database.

Note this function is deprecated and will be removed in the future.

Parameters

1. `String` - Database name.
2. `String` - Key name.
3. `DATA` - The data to store.

```
params: [
  "testDB",
  "myKey",
  "0x68656c6c6f20776f726c64"
]
```

Returns

`Boolean` - returns `true` if the value was stored, otherwise `false` .

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"db_putHex","params":["testDB","myKey","0x

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

db_getHex

Returns binary data from the local database.

Note this function is deprecated and will be removed in the future.

Parameters

1. `String` - Database name.
2. `String` - Key name.

```
params: [
  "testDB",
  "myKey",
]
```

Returns

`DATA` - The previously stored data.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"db_getHex","params":["testDB","myKey"],"i

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": "0x68656c6c6f20776f726c64"
}
```

shh_version

Returns the current whisper protocol version.

Parameters

none

Returns

`String` - The current whisper protocol version

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_version","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc": "2.0",
  "result": "2"
}
```

---

shh_post

Sends a whisper message.

Parameters

1. `Object` - The whisper post object:
    - `from` : `DATA` , 60 Bytes - (optional) The identity of the sender.
    - `to` : `DATA` , 60 Bytes - (optional) The identity of the receiver. When present whisper will encrypt the message so that only the receiver can decrypt it.
    - `topics` : `Array of DATA` - Array of `DATA` topics, for the receiver to identify messages.
    - `payload` : `DATA` - The payload of the message.
    - `priority` : `QUANTITY` - The integer of the priority in a rang from ... (?).
    - `ttl` : `QUANTITY` - integer of the time to live in seconds.

```
params: [{
  from: "0x04f96a5e25610293e42a73908e93ccc8c4d4dc0edcfa9fa872f50cb214e08ebf61a03e245533f9
  to: "0x3e245533f97284d442460f2998cd41858798ddf04f96a5e25610293e42a73908e93ccc8c4d4dc0ed
  topics: ["0x776869737065722d636861742d636c69656e74", "0x4d5a695276454c39425154466b61693
  payload: "0x7b2274797065223a226d6",
  priority: "0x64",
  ttl: "0x64",
}]
```

Returns

`Boolean` - returns `true` if the message was send, otherwise `false` .

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_post","params":[{"from":"0xc931d93e97

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

shh_newIdentity

Creates new whisper identity in the client.

Parameters

none

Returns

`DATA` , 60 Bytes - the address of the new identiy.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_newIdentity","params":[],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xc931d93e97ab07fe42d923478ba2465f283f440fd6cabea4dd7a2c807108f651b7135d1d6c
}
```

shh_hasIdentity

Checks if the client hold the private keys for a given identity.

Parameters

1. `DATA` , 60 Bytes - The identity address to check.

```
params: [
  "0x04f96a5e25610293e42a73908e93ccc8c4d4dc0edcfa9fa872f50cb214e08ebf61a03e245533f97284d4
]
```

Returns

`Boolean` - returns `true` if the client holds the privatekey for that identity, otherwise `false` .

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_hasIdentity","params":["0x04f96a5e256

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": true
}
```

shh_newGroup

(?)

Parameters

none

Returns

`DATA` , 60 Bytes - the address of the new group. (?)

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_newIdentity","params":[],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xc65f283f440fd6cabea4dd7a2c807108f651b7135d1d6ca90931d93e97ab07fe42d923478b
}
```

shh_addToGroup

(?)

Parameters

1. `DATA` , 60 Bytes - The identity address to add to a group (?).

```
params: [
  "0x04f96a5e25610293e42a73908e93ccc8c4d4dc0edcfa9fa872f50cb214e08ebf61a03e245533f97284d4
]
```

Returns

`Boolean` - returns `true` if the identity was successfully added to the group, otherwise `false` (?).

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_hasIdentity","params":["0x04f96a5e256

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": true
}
```

shh_newFilter

Creates filter to notify, when client receives whisper message matching the filter options.

Parameters

1. `Object` - The filter options:
   - `to` : `DATA` , 60 Bytes - (optional) Identity of the receiver. *When present it will try to decrypt any incoming message if the client holds the private key to this identity.*
   - `topics` : `Array of DATA` - Array of `DATA` topics which the incoming message's topics should match. You can use the following combinations:
     - `[A, B] = A && B`
     - `[A, [B, C]] = A && (B || C)`
     - `[null, A, B] = ANYTHING && A && B` `null` works as a wildcard

```
params: [{
   "topics": ['0x12341234bf4b564f'],
   "to": "0x04f96a5e25610293e42a73908e93ccc8c4d4dc0edcfa9fa872f50cb214e08ebf61a03e245533f
}]
```

Returns

`QUANTITY` - The newly created filter.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_newFilter","params":[{"topics": ['0x1

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": "0x7" // 7
}
```

shh_uninstallFilter

Uninstalls a filter with given id. Should always be called when watch is no longer needed. Additonally Filters timeout when they aren't requested with shh_getFilterChanges for a period of time.

Parameters

1. `QUANTITY` - The filter id.

```
params: [
  "0x7" // 7
]
```

Returns

`Boolean` - `true` if the filter was successfully uninstalled, otherwise `false` .

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_uninstallFilter","params":["0x7"],"id

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

shh_getFilterChanges

Polling method for whisper filters. Returns new messages since the last call of this method.

Note calling the shh_getMessages method, will reset the buffer for this method, so that you won't receive duplicate messages.

Parameters

1. `QUANTITY` - The filter id.

```
params: [
  "0x7" // 7
]
```

Returns

`Array` - Array of messages received since last poll:

- `hash` : `DATA` , 32 Bytes (?) - The hash of the message.
- `from` : `DATA` , 60 Bytes - The sender of the message, if a sender was specified.
- `to` : `DATA` , 60 Bytes - The receiver of the message, if a receiver was specified.
- `expiry` : `QUANTITY` - Integer of the time in seconds when this message should expire (?).
- `ttl` : `QUANTITY` - Integer of the time the message should float in the system in seconds (?).
- `sent` : `QUANTITY` - Integer of the unix timestamp when the message was sent.
- `topics` : `Array of DATA` - Array of `DATA` topics the message contained.
- `payload` : `DATA` - The payload of the message.
- `workProved` : `QUANTITY` - Integer of the work this message required before it was send (?).

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_getFilterChanges","params":["0x7"],"i

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": [{
    "hash": "0x33eb2da77bf3527e28f8bf493650b1879b08c4f2a362beae4ba2f71bafcd91f9",
    "from": "0x3ec052fc33..",
    "to": "0x87gdf76g8d7fgdfg...",
    "expiry": "0x54caa50a", // 1422566666
    "sent": "0x54ca9ea2", // 1422565026
    "ttl": "0x64" // 100
    "topics": ["0x6578616d"],
    "payload": "0x7b2274797065223a226d657373616765222c2263686...",
    "workProved": "0x0"
    }]
}
```

shh_getMessages

Get all messages matching a filter, which are still existing in the node's buffer.

Note calling this method, will also reset the buffer for the shh_getFilterChanges method, so that you won't receive duplicate messages.

Parameters

1. `QUANTITY` - The filter id.

```
params: [
  "0x7" // 7
]
```

Returns

See shh_getFilterChanges

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_getMessages","params":["0x7"],"id":73
```

Result see shh_getFilterChanges

Web3 JavaScript Đapp API

To make your Đapp work on Ethereum, you can use the `web3` object provided by the web3.js library. Under the hood it communicates to a local node through RPC calls. web3.js works with any Ethereum node, which exposes an RPC layer.

`web3` contains the `eth` object - `web3.eth` (for specifically Ethereum blockchain interactions) and the `shh` object - `web3.shh` (for Whisper interaction). Over time we'll introduce other objects for each of the other web3 protocols. Working examples can be found here.

If you want to look at some more sophisticated examples using web3.js check out these useful Đapp patterns.

Using callbacks

As this API is designed to work with a local RPC node and all its functions are by default use synchronous HTTP requests.con

If you want to make asynchronous request, you can pass an optional callback as the last parameter to most functions. All callbacks are using an error first callback style:

```
web3.eth.getBlock(48, function(error, result){
    if(!error)
        console.log(result)
    else
        console.error(error);
})
```

Batch requests

Batch requests allow queuing up requests and processing them at once.

```
var batch = web3.createBatch();
batch.add(web3.eth.getBalance.request('0x0000000000000000000000000000000000000000', 'late
batch.add(web3.eth.contract(abi).at(address).balance.request(address, callback2));
batch.execute();
```

A note on big numbers in web3.js

You will always get a BigNumber object for balance values as JavaScript is not able to handle big numbers correctly. Look at the following examples:

```
"101010100324325345346456456456456456"
// "101010100324325345346456456456456456"
101010100324325345346456456456456456
// 1.0101010032432535e+38
```

web3.js depends on the BigNumber Library and adds it automatically.

```
var balance = new BigNumber('131242344353464564564574574567456');
// or var balance = web3.eth.getBalance(someAddress);

balance.plus(21).toString(10); // toString(10) converts it to a number string
// "131242344353464564564574574567477"
```

The next example wouldn't work as we have more than 20 floating points, therefore it is recommended to keep you balance always in *wei* and only transform it to other units when presenting to the user:

```
var balance = new BigNumber('13124.234435346456466666457455567456');

balance.plus(21).toString(10); // toString(10) converts it to a number string, but can on
// "13145.23443534645646666646" // you number would be cut after the 20 floating point
```

Web3 Javascript Ðapp API Reference

- web3
  - version
    - api
    - client
    - network
    - ethereum
    - whisper
  - isConnected()
  - setProvider(provider)
  - currentProvider
  - reset()

- get(callback)

Usage

web3

The `web3` object provides all methods.

Example

```
var Web3 = require('web3');
// create an instance of web3 using the HTTP provider.
// NOTE in mist web3 is already available, so check first if its available before instant
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
```

web3.version.api

```
web3.version.api
// or async
web3.version.getApi(callback(error, result){ ... })
```

Returns

`String` - The ethereum js api version.

Example

```
var version = web3.version.api;
console.log(version); // "0.2.0"
```

web3.version.client

```
web3.version.client
// or async
web3.version.getClient(callback(error, result){ ... })
```

Returns

`String` - The client/node version.

Example

```
var version = web3.version.client;
console.log(version); // "Mist/v0.9.3/darwin/go1.4.1"
```

web3.version.network

```
web3.version.network
// or async
web3.version.getNetwork(callback(error, result){ ... })
```

Returns

`String` - The network protocol version.

Example

```
var version = web3.version.network;
console.log(version); // 54
```

web3.version.ethereum

```
web3.version.ethereum
// or async
web3.version.getEthereum(callback(error, result){ ... })
```

Returns

`String` - The ethereum protocol version.

Example

```
var version = web3.version.ethereum;
console.log(version); // 60
```

web3.version.whisper

```
web3.version.whisper
// or async
web3.version.getWhisper(callback(error, result){ ... })
```

Returns

`String` - The whisper protocol version.

Example

```
var version = web3.version.whisper;
console.log(version); // 20
```

web3.isConnected

```
web3.isConnected()
```

Should be called to check if a connection to a node exists

Parameters

none

Returns

`Boolean`

Example

```
if(!web3.isConnected()) {

   // show some dialog to ask the user to start a node

} else {

   // start web3 filters, calls, etc

}
```

web3.setProvider

```
web3.setProvider(provider)
```

Should be called to set provider.

Parameters

none

Returns

`undefined`

Example

```
web3.setProvider(new web3.providers.HttpProvider('http://localhost:8545')); // 8080 for c
```

---

web3.currentProvider

```
web3.currentProvider
```

Will contain the current provider, if one is set. This can be used to check if mist etc. set already a provider.

Returns

`Object` - The provider set or `null`;

Example

```
// Check if mist etc. already set a provider
if(!web3.currentProvider)
    web3.setProvider(new web3.providers.HttpProvider("http://localhost:8545"));
```

---

web3.reset

```
web3.reset(keepIsSyncing)
```

Should be called to reset state of web3. Resets everything except manager. Uninstalls all filters. Stops polling.

Parameters

1. `Boolean` - If `true` it will uninstall all filters, but will keep the web3.eth.isSyncing() polls

Returns

`undefined`

Example

```
web3.reset();
```

---

web3.sha3

```
web3.sha3(string [, callback])
```

Parameters

1. `String` - The string to hash using the SHA3 algorithm
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`String` - The SHA3 of the given data.

Example

```
var str = web3.sha3("Some ASCII string to be hashed");
console.log(str); // "0x536f6d6520415343494920737472696e6720746f206265206861736865642"

var hash = web3.sha3(str);
console.log(hash); // "0xb21dbc7a5eb6042d91f8f584af266f1a512ac89520f43562c6c1e37eab6eb0c4
```

---

web3.toHex

```
web3.toHex(mixed);
```

Converts any value into HEX.

Parameters

1. `String|Number|Object|Array|BigNumber` - The value to parse to HEX. If its an object or array it will be `JSON.stringify` first. If its a BigNumber it will make it the HEX value of a number.

Returns

`String` - The hex string of `mixed`.

Example

```
var str = web3.toHex({test: 'test'});
console.log(str); // '0x7b2274657374223a2274657374227d'
```

---

web3.toAscii

```
web3.toAscii(hexString);
```

Converts a HEX string into a ASCII string.

Parameters

1. `String` - A HEX string to be converted to ascii.

Returns

`String` - An ASCII string made from the given `hexString`.

Example

```
var str = web3.toAscii("0x657468657265756d00000000000000000000000000000000000000000000000
console.log(str); // "ethereum"
```

---

web3.fromAscii

```
web3.fromAscii(string [, padding]);
```

Converts any ASCII string to a HEX string.

Parameters

1. `String` - An ASCII string to be converted to HEX.
2. `Number` - The number of bytes the returned HEX string should have.

Returns

`String` - The converted HEX string.

Example

```
var str = web3.fromAscii('ethereum');
console.log(str); // "0x657468657265756d"

var str2 = web3.fromAscii('ethereum', 32);
console.log(str2); // "0x657468657265756d000000000000000000000000000000000000000000000000
```

web3.toDecimal

```
web3.toDecimal(hexString);
```

Converts a HEX string to its number representation.

Parameters

1. `String` - An HEX string to be converted to a number.

Returns

`Number` - The number representing the data `hexString` .

Example

```
var number = web3.toDecimal('0x15');
console.log(number); // 21
```

web3.fromDecimal

```
web3.fromDecimal(number);
```

Converts a number or number string to its HEX representation.

Parameters

1. `Number|String` - A number to be converted to a HEX string.

Returns

`String` - The HEX string representing of the given `number` .

Example

```
var value = web3.fromDecimal('21');
console.log(value); // "0x15"
```

---

web3.fromWei

```
web3.fromWei(number, unit)
```

Converts a number of wei into the following ethereum units:

- `kwei` / `ada`
- `mwei` / `babbage`
- `gwei` / `shannon`
- `szabo`
- `finney`
- `ether`
- `kether` / `grand` / `einstein`
- `mether`
- `gether`
- `tether`

Parameters

1. `Number|String|BigNumber` - A number or BigNumber instance.
2. `String` - One of the above ether units.

Returns

`String|BigNumber` - Either a number string, or a BigNumber instance, depending on the given `number` parameter.

Example

```
var value = web3.fromWei('21000000000000', 'finney');
console.log(value); // "0.021"
```

web3.toWei

```
web3.toWei(number, unit)
```

Converts an ethereum unit into wei. Possible units are:

- `kwei` / `ada`
- `mwei` / `babbage`
- `gwei` / `shannon`
- `szabo`
- `finney`
- `ether`
- `kether` / `grand` / `einstein`
- `mether`
- `gether`
- `tether`

Parameters

1. `Number|String|BigNumber` - A number or BigNumber instance.
2. `String` - One of the above ether units.

Returns

`String|BigNumber` - Either a number string, or a BigNumber instance, depending on the given `number` parameter.

Example

```
var value = web3.toWei('1', 'ether');
console.log(value); // "1000000000000000000"
```

web3.toBigNumber

```
web3.toBigNumber(numberOrHexString);
```

Converts a given number into a BigNumber instance.

See the note on BigNumber.

Parameters

1. `Number|String` - A number, number string or HEX string of a number.

Returns

`BigNumber` - A BigNumber instance representing the given value.

Example

```
var value = web3.toBigNumber('200000000000000000000001');
console.log(value); // instanceOf BigNumber
console.log(value.toNumber()); // 2.0000000000000002e+23
console.log(value.toString(10)); // '200000000000000000000001'
```

web3.net

web3.net.listening

```
web3.net.listening
// or async
web3.net.getListening(callback(error, result){ ... })
```

This property is read only and says whether the node is actively listening for network connections or not.

Returns

`Boolean` - `true` if the client is actively listening for network connections, otherwise `false` .

Example

```
var listening = web3.net.listening;
console.log(listening); // true of false
```

web3.net.peerCount

```
web3.net.peerCount
// or async
web3.net.getPeerCount(callback(error, result){ ... })
```

This property is read only and returns the number of connected peers.

Returns

`Number` - The number of peers currently connected to the client.

Example

```
var peerCount = web3.net.peerCount;
console.log(peerCount); // 4
```

web3.eth

Contains the ethereum blockchain related methods.

Example

```
var eth = web3.eth;
```

web3.eth.defaultAccount

```
web3.eth.defaultAccount
```

This default address is used for the following methods (optionally you can overwrite it by specifying the `from` property):

- web3.eth.sendTransaction()
- web3.eth.call()

Values

`String`, 20 Bytes - Any address you own, or where you have the private key for.

Default is `undefined`.

Returns

`String`, 20 Bytes - The currently set default address.

Example

```
var defaultAccount = web3.eth.defaultAccount;
console.log(defaultAccount); // ''

// set the default block
web3.eth.defaultAccount = '0x8888f1f195afa192cfee860698584c030f4c9db1';
```

---

web3.eth.defaultBlock

```
web3.eth.defaultBlock
```

This default block is used for the following methods (optionally you can overwrite the defaultBlock by passing it as the last parameter):

- web3.eth.getBalance()
- web3.eth.getCode()
- web3.eth.getTransactionCount()
- web3.eth.getStorageAt()
- web3.eth.call()

Values

Default block parameters can be one of the following:

- `Number` - a block number
- `String` - `"earliest"`, the genisis block
- `String` - `"latest"`, the latest block (current head of the blockchain)
- `String` - `"pending"`, the currently mined block (including pending transactions)

Default is `latest`

Returns

`Number|String` - The default block number to use when querying a state.

Example

```
var defaultBlock = web3.eth.defaultBlock;
console.log(defaultBlock); // 'latest'

// set the default block
web3.eth.defaultBlock = 231;
```

---

web3.eth.syncing

```
web3.eth.syncing
// or async
web3.eth.getSyncing(callback(error, result){ ... })
```

This property is read only and returns the either a sync object, when the node is syncing or `false` .

Returns

`Object|Boolean` - A sync object as follows, when the node is currently syncing or `false` :

- `startingBlock` : `Number` - The block number where the sync started.
- `currentBlock` : `Number` - The block number where at which block the node currently synced to already.
- `highestBlock` : `Number` - The estimated block number to sync to.

Example

```
var sync = web3.eth.syncing;
console.log(sync);
/*
{
   startingBlock: 300,
   currentBlock: 312,
   highestBlock: 512
}
*/
```

---

web3.eth.isSyncing

```
web3.eth.isSyncing(callback);
```

This convenience function calls the `callback` everytime a sync starts, updates and stops.

Returns

`Object` - a isSyncing object with the following methods:

- `syncing.addCallback()` : Adds another callback, which will be called when the node starts or stops syncing.
- `syncing.stopWatching()` : Stops the syncing callbacks.

Callback return value

- `Boolean` - The callback will be fired with `true` when the syncing starts and with `false` when it stopped.
- `Object` - While syncing it will return the syncing object:
    - `startingBlock` : `Number` - The block number where the sync started.
    - `currentBlock` : `Number` - The block number where at which block the node currently synced to already.
    - `highestBlock` : `Number` - The estimated block number to sync to.

Example

```
web3.eth.isSyncing(function(error, sync){
    if(!error) {
        // stop all app activity
        if(sync === true) {
           // we use `true`, so it stops all filters, but not the web3.eth.syncing pollin
           web3.reset(true);

        // show sync info
        } else if(sync) {
           console.log(sync.currentBlock);

        // re-gain app operation
        } else {
            // run your app init function...
        }
    }
});
```

web3.eth.coinbase

```
web3.eth.coinbase
// or async
web3.eth.getCoinbase(callback(error, result){ ... })
```

This property is read only and returns the coinbase address were the mining rewards go to.

Returns

`String` - The coinbase address of the client.

Example

```
var coinbase = web3.eth.coinbase;
console.log(coinbase); // "0x407d73d8a49eeb85d32cf465507dd71d507100c1"
```

web3.eth.mining

```
web3.eth.mining
// or async
web3.eth.getMining(callback(error, result){ ... })
```

This property is read only and says whether the node is mining or not.

Returns

`Boolean` - `true` if the client is mining, otherwise `false` .

Example

```
var mining = web3.eth.mining;
console.log(mining); // true or false
```

web3.eth.hashrate

```
web3.eth.hashrate
// or async
web3.eth.getHashrate(callback(error, result){ ... })
```

This property is read only and returns the number of hashes per second that the node is mining with.

Returns

`Number` - number of hashes per second.

Example

```
var hashrate = web3.eth.hashrate;
console.log(hashrate); // 493736
```

---

web3.eth.gasPrice

```
web3.eth.gasPrice
// or async
web3.eth.getGasPrice(callback(error, result){ ... })
```

This property is read only and returns the current gas price. The gas price is determined by the x latest blocks median gas price.

Returns

`BigNumber` - A BigNumber instance of the current gas price in wei.

See the note on BigNumber.

Example

```
var gasPrice = web3.eth.gasPrice;
console.log(gasPrice.toString(10)); // "10000000000000"
```

---

web3.eth.accounts

```
web3.eth.accounts
// or async
web3.eth.getAccounts(callback(error, result){ ... })
```

This property is read only and returns a list of accounts the node controls.

Returns

`Array` - An array of addresses controlled by client.

Example

```
var accounts = web3.eth.accounts;
console.log(accounts); // ["0x407d73d8a49eeb85d32cf465507dd71d507100c1"]
```

---

web3.eth.blockNumber

```
web3.eth.blockNumber
// or async
web3.eth.getBlockNumber(callback(error, result){ ... })
```

This property is read only and returns the current block number.

Returns

`Number` - The number of the most recent block.

Example

```
var number = web3.eth.blockNumber;
console.log(number); // 2744
```

---

web3.eth.register

```
web3.eth.register(addressHexString [, callback])
```

(Not Implemented yet) Registers the given address to be included in `web3.eth.accounts`. This allows non-private-key owned accounts to be associated as an owned account (e.g., contract wallets).

Parameters

1. `String` - The address to register
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

?

Example

```
web3.eth.register("0x407d73d8a49eeb85d32cf465507dd71d507100ca")
```

---

web3.eth.unRegister

```
web3.eth.unRegister(addressHexString [, callback])
```

(Not Implemented yet) Unregisters a given address.

Parameters

1. `String` - The address to unregister.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

?

Example

```
web3.eth.unregister("0x407d73d8a49eeb85d32cf465507dd71d507100ca")
```

---

web3.eth.getBalance

```
web3.eth.getBalance(addressHexString [, defaultBlock] [, callback])
```

Get the balance of an address at a given block.

Parameters

1. `String` - The address to get the balance of.
2. `Number|String` - (optional) If you pass this parameter it will not use the default block set with web3.eth.defaultBlock.

3. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`String` - A BigNumber instance of the current balance for the given address in wei.

See the note on BigNumber.

Example

```
var balance = web3.eth.getBalance("0x407d73d8a49eeb85d32cf465507dd71d507100c1");
console.log(balance); // instanceof BigNumber
console.log(balance.toString(10)); // '1000000000000'
console.log(balance.toNumber()); // 1000000000000
```

web3.eth.getStorageAt

```
web3.eth.getStorageAt(addressHexString, position [, defaultBlock] [, callback])
```

Get the storage at a specific position of an address.

Parameters

1. `String` - The address to get the storage from.
2. `Number` - The index position of the storage.
3. `Number|String` - (optional) If you pass this parameter it will not use the default block set with web3.eth.defaultBlock.
4. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`String` - The value in storage at the given position.

Example

```
var state = web3.eth.getStorageAt("0x407d73d8a49eeb85d32cf465507dd71d507100c1", 0);
console.log(state); // "0x03"
```

web3.eth.getCode

```
web3.eth.getCode(addressHexString [, defaultBlock] [, callback])
```

Get the code at a specific address.

Parameters

1. `String` - The address to get the code from.
2. `Number|String` - (optional) If you pass this parameter it will not use the default block set with web3.eth.defaultBlock.
3. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`String` - The data at given address `addressHexString`.

Example

```
var code = web3.eth.getCode("0xd5677cf67b5aa051bb40496e68ad359eb97cfbf8");
console.log(code); // "0x600160008035811a818181146012578301005b601b6001356025565b80600052
```

web3.eth.getBlock

```
web3.eth.getBlock(blockHashOrBlockNumber [, returnTransactionObjects] [, callback])
```

Returns a block matching the block number or block hash.

Parameters

1. `String|Number` - The block number or hash. Or the string `"earliest"`, `"latest"` or `"pending"` as in the default block parameter.
2. `Boolean` - (optional, default `false`) If `true`, the returned block will contain all transactions as objects, if `false` it will only contains the transaction hashes.
3. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Object` - The block object:

- `number` : `Number` - the block number. `null` when its pending block.
- `hash` : `String` , 32 Bytes - hash of the block. `null` when its pending block.
- `parentHash` : `String` , 32 Bytes - hash of the parent block.
- `nonce` : `String` , 8 Bytes - hash of the generated proof-of-work. `null` when its pending block.
- `sha3Uncles` : `String` , 32 Bytes - SHA3 of the uncles data in the block.
- `logsBloom` : `String` , 256 Bytes - the bloom filter for the logs of the block. `null` when its pending block.
- `transactionsRoot` : `String` , 32 Bytes - the root of the transaction trie of the block
- `stateRoot` : `String` , 32 Bytes - the root of the final state trie of the block.
- `miner` : `String` , 20 Bytes - the address of the beneficiary to whom the mining rewards were given.
- `difficulty` : `BigNumber` - integer of the difficulty for this block.
- `totalDifficulty` : `BigNumber` - integer of the total difficulty of the chain until this block.
- `extraData` : `String` - the "extra data" field of this block.
- `size` : `Number` - integer the size of this block in bytes.
- `gasLimit` : `Number` - the maximum gas allowed in this block.
- `gasUsed` : `Number` - the total used gas by all transactions in this block.
- `timestamp` : `Number` - the unix timestamp for when the block was collated.
- `transactions` : `Array` - Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter.
- `uncles` : `Array` - Array of uncle hashes.

Example

```
var info = web3.eth.block(3150);
console.log(info);
/*
{
  "number": 3,
  "hash": "0xef95f2f1ed3ca60b048b4bf67cde2195961e0bba6f70bcbea9a2c4e133e34b46",
  "parentHash": "0x2302e1c0b972d00932deb5dab9eb2982f570597d9d42504c05d9c2147eaf9c88",
  "nonce": "0xfb6e1a62d119228b",
  "sha3Uncles": "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
  "logsBloom": "0x00000000000000000000000000000000000000000000000000000000000000000000000
  "transactionsRoot": "0x3a1b03875115b79539e5bd33fb00d8f7b7cd61929d5a3c574f507b8acf415bee
  "stateRoot": "0xf1133199d44695dfa8fd1bcfe424d82854b5cebef75bddd7e40ea94cda515bcb",
  "miner": "0x8888f1f195afa192cfee860698584c030f4c9db1",
  "difficulty": BigNumber,
  "totalDifficulty": BigNumber,
  "size": 616,
  "extraData": "0x",
  "gasLimit": 3141592,
  "gasUsed": 21662,
  "timestamp": 1429287689,
  "transactions": [
    "0x9fc76417374aa880d4449a1f7f31ec597f00b1f6f3dd2d66f4c9c6c445836d8b"
  ],
  "uncles": []
}
*/
```

web3.eth.getBlockTransactionCount

```
web3.eth.getBlockTransactionCount(hashStringOrBlockNumber [, callback])
```

Returns the number of transaction in a given block.

Parameters

1. `String|Number` - The block number or hash. Or the string `"earliest"`, `"latest"` or `"pending"` as in the default block parameter.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Number` - The number of transactions in the given block.

Example

```
var number = web3.eth.getBlockTransactionCount("0x407d73d8a49eeb85d32cf465507dd71d507100c
console.log(number); // 1
```

web3.eth.getUncle

```
web3.eth.getUncle(blockHashStringOrNumber, uncleNumber [, returnTransactionObjects] [, ca
```

Returns a blocks uncle by a given uncle index position.

Parameters

1. `String|Number` - The block number or hash. Or the string `"earliest"` , `"latest"` or `"pending"` as in the default block parameter.
2. `Number` - The index position of the uncle.
3. `Boolean` - (optional, default `false`) If `true` , the returned block will contain all transactions as objects, if `false` it will only contains the transaction hashes.
4. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Object` - the returned uncle. For a return value see web3.eth.getBlock().

Note: An uncle doesn't contain individual transactions.

Example

```
var uncle = web3.eth.getUncle(500, 0);
console.log(uncle); // see web3.eth.getBlock
```

web3.eth.getTransaction

```
web3.eth.getTransaction(transactionHash [, callback])
```

Returns a transaction matching the given transaction hash.

Parameters

1. `String` - The transaction hash.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Object` - A transaction object its hash `transactionHash` :

- `hash` : `String` , 32 Bytes - hash of the transaction.
- `nonce` : `Number` - the number of transactions made by the sender prior to this one.
- `blockHash` : `String` , 32 Bytes - hash of the block where this transaction was in. `null` when its pending.
- `blockNumber` : `Number` - block number where this transaction was in. `null` when its pending.
- `transactionIndex` : `Number` - integer of the transactions index position in the block. `null` when its pending.
- `from` : `String` , 20 Bytes - address of the sender.
- `to` : `String` , 20 Bytes - address of the receiver. `null` when its a contract creation transaction.
- `value` : `BigNumber` - value transferred in Wei.
- `gasPrice` : `BigNumber` - gas price provided by the sender in Wei.
- `gas` : `Number` - gas provided by the sender.
- `input` : `String` - the data sent along with the transaction.

Example

```
var blockNumber = 668;
var indexOfTransaction = 0

var transaction = web3.eth.getTransaction(blockNumber, indexOfTransaction);
console.log(transaction);
/*
{
  "hash": "0x9fc76417374aa880d4449a1f7f31ec597f00b1f6f3dd2d66f4c9c6c445836d8b",
  "nonce": 2,
  "blockHash": "0xef95f2f1ed3ca60b048b4bf67cde2195961e0bba6f70bcbea9a2c4e133e34b46",
  "blockNumber": 3,
  "transactionIndex": 0,
  "from": "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b",
  "to": "0x6295ee1b4f6dd65047762f924ecd367c17eabf8f",
  "value": BigNumber,
  "gas": 314159,
  "gasPrice": BigNumber,
  "input": "0x57cb2fc4"
}
*/
```

web3.eth.getTransactionFromBlock

```
getTransactionFromBlock(hashStringOrNumber, indexNumber [, callback])
```

Returns a transaction based on a block hash or number and the transactions index position.

Parameters

1. `String` - A block number or hash. Or the string `"earliest"` , `"latest"` or `"pending"` as in the default block parameter.
2. `Number` - The transactions index position.
3. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Object` - A transaction object, see web3.eth.getTransaction:

Example

```
var transaction = web3.eth.getTransactionFromBlock('0x4534534534', 2);
console.log(transaction); // see web3.eth.getTransaction
```

web3.eth.getTransactionReceipt

```
web3.eth.getTransactionReceipt(hashString [, callback])
```

Returns the receipt of a transaction by transaction hash.

Note That the receipt is not available for pending transactions.

Parameters

1. `String` - The transaction hash.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Object` - A transaction receipt object, or `null` when no receipt was found:

- `blockHash` : `String` , 32 Bytes - hash of the block where this transaction was in.
- `blockNumber` : `Number` - block number where this transaction was in.
- `transactionHash` : `String` , 32 Bytes - hash of the transaction.
- `transactionIndex` : `Number` - integer of the transactions index position in the block.
- `from` : `String` , 20 Bytes - address of the sender.
- `to` : `String` , 20 Bytes - address of the receiver. `null` when its a contract creation transaction.
- `cumulativeGasUsed` : `Number` - The total amount of gas used when this transaction was executed in the block.
- `gasUsed` : `Number` - The amount of gas used by this specific transaction alone.
- `contractAddress` : `String` - 20 Bytes - The contract address created, if the transaction was a contract creation, otherwise `null` .
- `logs` : `Array` - Array of log objects, which this transaction generated.

Example

```
var receipt = web3.eth.getTransactionReceipt('0x9fc76417374aa880d4449a1f7f31ec597f00b1f6f
console.log(receipt);
{
  "transactionHash": "0x9fc76417374aa880d4449a1f7f31ec597f00b1f6f3dd2d66f4c9c6c445836d8b"
  "transactionIndex": 0,
  "blockHash": "0xef95f2f1ed3ca60b048b4bf67cde2195961e0bba6f70bcbea9a2c4e133e34b46",
  "blockNumber": 3,
  "contractAddress": "0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b",
  "cumulativeGasUsed": 314159,
  "gasUsed": 30234,
  "logs": [{
         // logs as returned by getFilterLogs, etc.
     }, ...]
}
```

web3.eth.getTransactionCount

```
web3.eth.getTransactionCount(addressHexString [, defaultBlock] [, callback])
```

Get the numbers of transactions sent from this address.

Parameters

1. `String` - The address to get the numbers of transactions from.
2. `Number|String` - (optional) If you pass this parameter it will not use the default block set

with web3.eth.defaultBlock.

3. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Number` - The number of transactions sent from the given address.

Example

```
var number = web3.eth.getTransactionCount("0x407d73d8a49eeb85d32cf465507dd71d507100c1");
console.log(number); // 1
```

web3.eth.sendTransaction

```
web3.eth.sendTransaction(transactionObject [, callback])
```

Sends a transaction to the network.

Parameters

1. `Object` - The transaction object to send:
   - `from` : `String` - The address for the sending account. Uses the web3.eth.defaultAccount property, if not specified.
   - `to` : `String` - (optional) The destination address of the message, left undefined for a contract-creation transaction.
   - `value` : `Number|String|BigNumber` - (optional) The value transferred for the transaction in Wei, also the endowment if it's a contract-creation transaction.
   - `gas` : `Number|String|BigNumber` - (optional, default: To-Be-Determined) The amount of gas to use for the transaction (unused gas is refunded).
   - `gasPrice` : `Number|String|BigNumber` - (optional, default: To-Be-Determined) The price of gas for this transaction in wei, defaults to the mean network gas price.
   - `data` : `String` - (optional) Either a byte string containing the associated data of the message, or in the case of a contract-creation transaction, the initialisation code.
   - `nonce` : `Number` - (optional) Integer of a nonce. This allows to overwrite your own pending transactions that use the same nonce.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`String` - The 32 Bytes transaction hash as HEX string.

If the transaction was a contract creation use web3.eth.getTransactionReceipt() to get the contract address, after the transaction was mined.

Example

```
// compiled solidity source code using https://chriseth.github.io/cpp-ethereum/
var code = "603d80600c6000396000f3007c010000000000000000000000000000000000000000000000000
8060005260206000f3";

web3.eth.sendTransaction({data: code}, function(err, address) {
  if (!err)
    console.log(address); // "0x7f9fade1c0d57a7af66ab4ead7c2eb7b11a91385"
});
```

web3.eth.call

```
web3.eth.call(callObject [, defaultBlock] [, callback])
```

Executes a message call transaction, which is directly executed in the VM of the node, but never mined into the blockchain.

Parameters

1. `Object` - A transaction object see web3.eth.sendTransaction, with the difference that for calls the `from` property is optional as well.
2. `Number|String` - (optional) If you pass this parameter it will not use the default block set with web3.eth.defaultBlock.
3. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`String` - The returned data of the call, e.g. a codes functions return value.

Example

```
var result = web3.eth.call({
    to: "0xc4abd0339eb8d57087278718986382264244252f",
    data: "0xc6888fa10000000000000000000000000000000000000000000000000000000000000003"
});
console.log(result); // "0x00000000000000000000000000000000000000000000000000000000000000
```

web3.eth.estimateGas

```
web3.eth.estimateGas(callObject [, defaultBlock] [, callback])
```

Executes a message call or transaction, which is directly executed in the VM of the node, but never mined into the blockchain and returns the amount of the gas used.

Parameters

See web3.eth.sendTransaction, expect that all properties are optional.

Returns

`Number` - the used gas for the simulated call/transaction.

Example

```
var result = web3.eth.estimateGas({
    to: "0xc4abd0339eb8d57087278718986382264244252f",
    data: "0xc6888fa10000000000000000000000000000000000000000000000000000000000000003"
});
console.log(result); // "0x00000000000000000000000000000000000000000000000000000000000000
```

web3.eth.filter

```
// can be 'latest' or 'pending'
var filter = web3.eth.filter(filterString);
// OR object are log filter options
var filter = web3.eth.filter(options);

// watch for changes
filter.watch(function(error, result){
  if (!error)
    console.log(result);
});

// Additionally you can start watching right away, by passing a callback:
web3.eth.filter(options, function(error, result){
  if (!error)
    console.log(result);
});
```

Parameters

1. `String|Object` - The string `"latest"` or `"pending"` to watch for changes in the latest block or pending transactions respectively. Or a filter options object as follows:
    - `fromBlock` : `Number|String` - The number of the earliest block (`latest` may be given to mean the most recent and `pending` currently mining, block). By default `latest` .
    - `toBlock` : `Number|String` - The number of the latest block (`latest` may be given to mean the most recent and `pending` currently mining, block). By default `latest` .
    - `address` : `String` - An address or a list of addresses to only get logs from particular account(s).
    - `topics` : `Array of Strings` - An array of values which must each appear in the log entries. The order is important, if you want to leave topics out use `null` , e.g. `[null, '0x00...']` .

Returns

`Object` - A filter object with the following methods:

- `filter.get(callback)` : Returns all of the log entries that fit the filter.
- `filter.watch(callback)` : Watches for state changes that fit the filter and calls the callback. See this note for details.
- `filter.stopWatching()` : Stops the watch and uninstalls the filter in the node. Should always be called once it is done.

Watch callback return value

- `String` - When using the `"latest"` parameter, it returns the block hash of the last incoming block.

- `String` - When using the `"pending"` parameter, it returns a transaction hash of the last add pending transaction.
- `Object` - When using manual filter options, it returns a log object as follows:
    - `logIndex` : `Number` - integer of the log index position in the block. `null` when its pending log.
    - `transactionIndex` : `Number` - integer of the transactions index position log was created from. `null` when its pending log.
    - `transactionHash` : `String` , 32 Bytes - hash of the transactions this log was created from. `null` when its pending log.
    - `blockHash` : `String` , 32 Bytes - hash of the block where this log was in. `null` when its pending. `null` when its pending log.
    - `blockNumber` : `Number` - the block number where this log was in. `null` when its pending. `null` when its pending log.
    - `address` : `String` , 32 Bytes - address from which this log originated.
    - `data` : `String` - contains one or more 32 Bytes non-indexed arguments of the log.
    - `topics` : `Array of Strings` - Array of 0 to 4 32 Bytes `DATA` of indexed log arguments. (In *solidity*: The first topic is the *hash* of the signature of the event (e.g. `Deposit(address,bytes32,uint256)`), except you declared the event with the `anonymous` specifier.)

Note For event filter return values see Contract Events

Example

```
var filter = web3.eth.filter('pending');

filter.watch(function (error, log) {
  console.log(log); //  {"address":"0x0000000000000000000000000000000000000000", "data":"
});

// get all past logs again.
var myResults = filter.get(function(error, logs){ ... });

...

// stops and uninstalls the filter
filter.stopWatching();
```

web3.eth.contract

```
web3.eth.contract(abiArray)
```

Creates a contract object for a solidity contract, which can be used to initiate contracts on an address. You can read more about events here.

Parameters

1. `Array` - ABI array with descriptions of functions and events of the contract.

Returns

`Object` - A contract object, which can be initiated as follows:

```
var MyContract = web3.eth.contract(abiArray);
```

And then you can either initiate an existing contract on an address, or deploy the contract using the compiled byte code:

```
// Instantiate from an existing address:
var myContractInstance = MyContract.at(myContractAddress);


// Or deploy a new contract:

// Deploy the contract asyncronous:
var myContractReturned = MyContract.new(param1, param2, {
   data: myContractCode,
   gas: 300000,
   from: mySenderAddress}, function(err, myContract){
    if(!err) {
       // NOTE: The callback will fire twice!
       // Once the contract has the transactionHash property set and once its deployed on

       // e.g. check tx hash on the first call (transaction send)
       if(!myContract.address) {
           console.log(myContract.transactionHash) // The hash of the transaction, which

       // check address on the second call (contract deployed)
       } else {
           console.log(myContract.address) // the contract address
       }

       // Note that the returned "myContractReturned" === "myContract",
       // so the returned "myContractReturned" object will also get the address set.
    }
  });

// Deploy contract syncronous: The address will be added as soon as the contract is mined
// Additionally you can watch the transaction by using the "transactionHash" property
var myContractInstance = MyContract.new(param1, param2, {data: myContractCode, gas: 30000
myContractInstance.transactionHash // The hash of the transaction, which created the cont
myContractInstance.address // undefined at start, but will be auto-filled later
```

Note When you deploy a new contract, you should check for the next 12 blocks or so if the contract code is still at the address (using web3.eth.getCode()), to make sure a fork didn't change that.

Example

```
// contract abi
var abi = [{
     name: 'myConstantMethod',
     type: 'function',
     constant: true,
     inputs: [{ name: 'a', type: 'string' }],
     outputs: [{name: 'd', type: 'string' }]
}, {
     name: 'myStateChangingMethod',
     type: 'function',
     constant: false,
     inputs: [{ name: 'a', type: 'string' }, { name: 'b', type: 'int' }],
     outputs: []
}, {
     name: 'myEvent',
     type: 'event',
     inputs: [{name: 'a', type: 'int', indexed: true},{name: 'b', type: 'bool', indexed:
}];

// creation of contract object
var MyContract = web3.eth.contract(abi);

// initiate contract for an address
var myContractInstance = MyContract.at('0xc4abd0339eb8d57087278718986382264244252f');

// call constant function
var result = myContractInstance.myConstantMethod('myParam');
console.log(result) // '0x25434534534'

// send a transaction to a function
myContractInstance.myStateChangingMethod('someParam1', 23, {value: 200, gas: 2000});

// short hand style
web3.eth.contract(abi).at(address).myAwesomeMethod(...);

// create filter
var filter = myContractInstance.myEvent({a: 5}, function (error, result) {
  if (!error)
    console.log(result);
    /*
    {
        address: '0x8718986382264244252fc4abd0339eb8d5708727',
        topics: "0x12345678901234567890123456789012", "0x0000000000000000000000000000000
        data: "0x0000000000000000000000000000000000000000000000000000000000000001",
        ...
    }
    */
});
```

Contract Methods

```
// Automatically determines the use of call or sendTransaction based on the method type
myContractInstance.myMethod(param1 [, param2, ...] [, transactionObject] [, callback]);

// Explicitly calling this method
myContractInstance.myMethod.call(param1 [, param2, ...] [, transactionObject] [, callback

// Explicitly sending a transaction to this method
myContractInstance.myMethod.sendTransaction(param1 [, param2, ...] [, transactionObject]
```

The contract object exposes the contracts methods, which can be called using parameters and a transaction object.

Parameters

- `String|Number` - (optional) Zero or more parameters of the function.
- `Object` - (optional) The (previous) last parameter can be a transaction object, see web3.eth.sendTransaction parameter 1 for more.
- `Function` - (optional) If you pass a callback as the last parameter the HTTP request is made asynchronous. See this note for details.

Returns

`String` - If its a call the result data, if its a send transaction a created contract address, or the transaction hash, see web3.eth.sendTransaction for details.

Example

```
// creation of contract object
var MyContract = web3.eth.contract(abi);

// initiate contract for an address
var myContractInstance = MyContract.at('0x78e97bcc5b5dd9ed228fed7a4887c0d7287344a9');

var result = myContractInstance.myConstantMethod('myParam');
console.log(result) // '0x25434534534'

myContractInstance.myStateChangingMethod('someParam1', 23, {value: 200, gas: 2000}, funct
```

Contract Events

```
var event = myContractInstance.MyEvent({valueA: 23} [, additionalFilterObject])

// watch for changes
event.watch(function(error, result){
  if (!error)
    console.log(result);
});

// Or pass a callback to start watching immediately
var event = myContractInstance.MyEvent([{valueA: 23}] [, additionalFilterObject] , functi
  if (!error)
    console.log(result);
});
```

You can use events like filters and they have the same methods, but you pass different objects to create the event filter.

Parameters

1. `Object` - Indexed return values you want to filter the logs by, e.g. `{'valueA': 1, 'valueB': [myFirstAddress, mySecondAddress]}` . By default all filter values are set to `null` . It means, that they will match any event of given type sent from this contract.
2. `Object` - Additional filter options, see filters parameter 1 for more. By default filterObject has field 'address' set to address of the contract. Also first topic is the signature of event.
3. `Function` - (optional) If you pass a callback as the last parameter it will immediately start watching and you don't need to call `myEvent.watch(function(){})` . See this note for details.

Callback return

`Object` - An event object as follows:

- `args` : `Object` - The arguments coming from the event.
- `event` : `String` - The event name.
- `logIndex` : `Number` - integer of the log index position in the block.
- `transactionIndex` : `Number` - integer of the transactions index position log was created from.
- `transactionHash` : `String` , 32 Bytes - hash of the transactions this log was created from.
- `address` : `String` , 32 Bytes - address from which this log originated.
- `blockHash` : `String` , 32 Bytes - hash of the block where this log was in. `null` when its pending.
- `blockNumber` : `Number` - the block number where this log was in. `null` when its

pending.

Example

```
var MyContract = web3.eth.contract(abi);
var myContractInstance = MyContract.at('0x78e97bcc5b5dd9ed228fed7a4887c0d7287344a9');

// watch for an event with {some: 'args'}
var myEvent = myContractInstance.MyEvent({some: 'args'}, {fromBlock: 0, toBlock: 'latest'
myEvent.watch(function(error, result){
   ...
});

// would get all past logs again.
var myResults = myEvent.get(function(error, logs){ ... });

...

// would stop and uninstall the filter
myEvent.stopWatching();
```

Contract allEvents

```
var events = myContractInstance.allEvents([additionalFilterObject]);

// watch for changes
events.watch(function(error, event){
  if (!error)
    console.log(event);
});

// Or pass a callback to start watching immediately
var events = myContractInstance.allEvents([additionalFilterObject,] function(error, log){
  if (!error)
    console.log(log);
});
```

Will call the callback for all events which are created by this contract.

Parameters

1. `Object` - Additional filter options, see filters parameter 1 for more. By default filterObject has field 'address' set to address of the contract. Also first topic is the signature of event.

2. `Function` - (optional) If you pass a callback as the last parameter it will immediately start watching and you don't need to call `myEvent.watch(function(){})`. See this note for details.

Callback return

`Object` - See Contract Events for more.

Example

```
var MyContract = web3.eth.contract(abi);
var myContractInstance = MyContract.at('0x78e97bcc5b5dd9ed228fed7a4887c0d7287344a9');

// watch for an event with {some: 'args'}
var events = myContractInstance.allEvents({fromBlock: 0, toBlock: 'latest'});
events.watch(function(error, result){
   ...
});

// would get all past logs again.
events.get(function(error, logs){ ... });

...

// would stop and uninstall the filter
myEvent.stopWatching();
```

web3.eth.getCompilers

```
web3.eth.getCompilers([callback])
```

Gets a list of available compilers.

Parameters

1. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Array` - An array of strings of available compilers.

Example

```
var number = web3.eth.getCompilers();
console.log(number); // ["lll", "solidity", "serpent"]
```

web3.eth.compile.solidity

```
web3.eth.compile.solidity(sourceString [, callback])
```

Compiles solidity source code.

Parameters

1. `String` - The solidity source code.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Object` - Contract and compiler info.

Example

```
var source = "" +
    "contract test {\n" +
    "   function multiply(uint a) returns(uint d) {\n" +
    "       return a * 7;\n" +
    "   }\n" +
    "}\n";
var compiled = web3.eth.compile.solidity(source);
console.log(compiled);
// {
  "test": {
    "code": "0x605280600c6000396000f3006000357c0100000000000000000000000000000000000000
    "info": {
      "source": "contract test {\n\tfunction multiply(uint a) returns(uint d) {\n\t\tretu
      "language": "Solidity",
      "languageVersion": "0",
      "compilerVersion": "0.8.2",
      "abiDefinition": [
        {
          "constant": false,
          "inputs": [
            {
              "name": "a",
              "type": "uint256"
            }
          ],
          "name": "multiply",
          "outputs": [
            {
              "name": "d",
              "type": "uint256"
            }
          ],
          "type": "function"
        }
      ],
      "userDoc": {
        "methods": {}
      },
      "developerDoc": {
        "methods": {}
      }
    }
  }
}
```

web3.eth.compile.lll

```
web3. eth.compile.lll(sourceString [, callback])
```

Compiles LLL source code.

Parameters

1. `String` - The LLL source code.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`String` - The compiled LLL code as HEX string.

Example

```
var source = "...";

var code = web3.eth.compile.lll(source);
console.log(code); // "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114
```

---

web3.eth.compile.serpent

```
web3.eth.compile.serpent(sourceString [, callback])
```

Compiles serpent source code.

Parameters

1. `String` - The serpent source code.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`String` - The compiled serpent code as HEX string.

```
var source = "...";

var code = web3.eth.compile.serpent(source);
console.log(code); // "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114
```

web3.eth.namereg

```
web3.eth.namereg
```

Returns GlobalRegistrar object.

Usage

see namereg example

web3.db

web3.db.putString

```
web3.db.putString(db, key, value)
```

This method should be called, when we want to store a string in the local leveldb database.

Parameters

1. `String` - The database to store to.
2. `String` - The name of the store.
3. `String` - The string value to store.

Returns

`Boolean` - `true` if successfull, otherwise `false` .

Example

param is db name, second is the key, and third is the string value.

```
web3.db.putString('testDB', 'key', 'myString') // true
```

web3.db.getString

```
web3.db.getString(db, key)
```

This method should be called, when we want to get string from the local leveldb database.

Parameters

1. `String` - The database string name to retrieve from.
2. `String` - The name of the store.

Returns

`String` - The stored value.

Example

param is db name and second is the key of string value.

```
var value = web3.db.getString('testDB', 'key');
console.log(value); // "myString"
```

web3.db.putHex

```
web3.db.putHex(db, key, value)
```

This method should be called, when we want to store binary data in HEX form in the local leveldb database.

Parameters

1. `String` - The database to store to.
2. `String` - The name of the store.
3. `String` - The HEX string to store.

Returns

`Boolean` - `true` if successfull, otherwise `false`.

Example

```
web3.db.putHex('testDB', 'key', '0x4f554b443'); // true
```

web3.db.getHex

```
web3.db.getHex(db, key)
```

This method should be called, when we want to get a binary data in HEX form from the local leveldb database.

Parameters

1. `String` - The database to store to.
2. `String` - The name of the store.

Returns

`String` - The stored HEX value.

Example

param is db name and second is the key of value.

```
var value = web3.db.getHex('testDB', 'key');
console.log(value); // "0x4f554b443"
```

web3.shh

Whisper Overview

Example

```
var shh = web3.shh;
```

web3.shh.post

web3.shh.post(object [, callback])

This method should be called, when we want to post whisper message to the network.

Parameters

1. `Object` - The post object:
   - `from` : `String` , 60 Bytes HEX - (optional) The identity of the sender.
   - `to` : `String` , 60 Bytes HEX - (optional) The identity of the receiver. When present whisper will encrypt the message so that only the receiver can decrypt it.
   - `topics` : `Array of Strings` - Array of topics `Strings` , for the receiver to identify messages.
   - `payload` : `String|Number|Object` - The payload of the message. Will be autoconverted to a HEX string before.
   - `priority` : `Number` - The integer of the priority in a rang from ... (?).
   - `ttl` : `Number` - integer of the time to live in seconds.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Boolean` - returns `true` if the message was send, otherwise `false` .

Example

```
var identity = web3.shh.newIdentity();
var topic = 'example';
var payload = 'hello whisper world!';

var message = {
  from: identity,
  topics: [topic],
  payload: payload,
  ttl: 100,
  workToProve: 100 // or priority TODO
};

web3.shh.post(message);
```

---

web3.shh.newIdentity

```
web3.shh.newIdentity([callback])
```

Should be called to create new identity.

Parameters

1. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`String` - A new identity HEX string.

Example

```
var identity = web3.shh.newIdentity();
console.log(identity); // "0xc931d93e97ab07fe42d923478ba2465f283f440fd6cabea4dd7a2c807108
```

web3.shh.hasIdentity

```
web3.shh.hasIdentity(identity, [callback])
```

Should be called, if we want to check if user has given identity.

Parameters

1. `String` - The identity to check.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Returns

`Boolean` - returns `true` if the identity exists, otherwise `false` .

Example

```
var identity = web3.shh.newIdentity();
var result = web3.shh.hasIdentity(identity);
console.log(result); // true

var result2 = web3.shh.hasIdentity(identity + "0");
console.log(result2); // false
```

web3.shh.newGroup

Example

```
// TODO: not implemented yet
```

web3.shh.addToGroup

Example

```
// TODO: not implemented yet
```

web3.shh.filter

```
var filter = web3.shh.filter(options)

// watch for changes
filter.watch(function(error, result){
  if (!error)
    console.log(result);
});
```

Watch for incoming whisper messages.

Parameters

1. `Object` - The filter options:
    - `topics` : `Array of Strings` - Filters messages by this topic(s). You can use the following combinations:
        - `['topic1', 'topic2'] == 'topic1' && 'topic2'`
        - `['topic1', ['topic2', 'topic3']] == 'topic1' && ('topic2' || 'topic3')`
        - `[null, 'topic1', 'topic2'] == ANYTHING && 'topic1' && 'topic2'` -> `null` works as a wildcard
    - `to` : Filter by identity of receiver of the message. If provided and the node has this identity, it will decrypt incoming encrypted messages.
2. `Function` - (optional) If you pass a callback the HTTP request is made asynchronous. See this note for details.

Callback return

`Object` - The incoming message:

- `from` : `String` , 60 Bytes - The sender of the message, if a sender was specified.

- `to`: `String`, 60 Bytes - The receiver of the message, if a receiver was specified.
- `expiry`: `Number` - Integer of the time in seconds when this message should expire (?).
- `ttl`: `Number` - Integer of the time the message should float in the system in seconds (?).
- `sent`: `Number` - Integer of the unix timestamp when the message was sent.
- `topics`: `Array of String` - Array of `String` topics the message contained.
- `payload`: `String` - The payload of the message.
- `workProved`: `Number` - Integer of the work this message required before it was send (?).

web3.eth.sendIBANTransaction

```
var txHash = web3.eth.sendIBANTransaction('0x00c5496aee77c1ba1f0854206a26dda82a81d6d8', '
```

Sends IBAN transaction from user account to destination IBAN address.

Parameters

- `string` - address from which we want to send transaction
- `string` - IBAN address to which we want to send transaction
- `value` - value that we want to send in IBAN transaction

web3.eth.iban

```
var i = new web3.eth.iban("XE81ETHXREGGAVOFYORK");
```

web3.eth.iban.fromAddress

```
var i = web3.eth.iban.fromAddress('0x00c5496aee77c1ba1f0854206a26dda82a81d6d8');
console.log(i.toString()); // 'XE7338O073KYGTWWZN0F2WZ0R8PX5ZPPZS
```

web3.eth.iban.fromBban

```
var i = web3.eth.iban.fromBban('ETHXREGGAVOFYORK');
console.log(i.toString()); // "XE81ETHXREGGAVOFYORK"
```

web3.eth.iban.createIndirect

```
var i = web3.eth.iban.createIndirect({
  institution: "XREG",
  identifier: "GAVOFYORK"
});
console.log(i.toString()); // "XE81ETHXREGGAVOFYORK"
```

web3.eth.iban.isValid

```
var valid = web3.eth.iban.isValid("XE81ETHXREGGAVOFYORK");
console.log(valid); // true

var valid2 = web3.eth.iban.isValid("XE82ETHXREGGAVOFYORK");
console.log(valid2); // false, cause checksum is incorrect

var i = new web3.eth.iban("XE81ETHXREGGAVOFYORK");
var valid3 = i.isValid();
console.log(valid3); // true
```

web3.eth.iban.isDirect

```
var i = new web3.eth.iban("XE81ETHXREGGAVOFYORK");
var direct = i.isDirect();
console.log(direct); // false
```

web3.eth.iban.isIndirect

```
var i = new web3.eth.iban("XE81ETHXREGGAVOFYORK");
var indirect = i.isIndirect();
console.log(indirect); // true
```

web3.eth.iban.checksum

```
var i = new web3.eth.iban("XE81ETHXREGGAVOFYORK");
var checksum = i.checksum();
console.log(checksum); // "81"
```

web3.eth.iban.institution

```
var i = new web3.eth.iban("XE81ETHXREGGAVOFYORK");
var institution = i.institution();
console.log(institution); // 'XREG'
```

web3.eth.iban.client

```
var i = new web3.eth.iban("XE81ETHXREGGAVOFYORK");
var client = i.client();
console.log(client); // 'GAVOFYORK'
```

web3.eth.iban.address

```
var i = new web3.eth.iban('XE7338O073KYGTWWZN0F2WZ0R8PX5ZPPZS');
var address = i.address();
console.log(address); // '00c5496aee77c1ba1f0854206a26dda82a81d6d8'
```

web3.eth.iban.toString

```
var i = new web3.eth.iban('XE7338O073KYGTWWZN0F2WZ0R8PX5ZPPZS');
console.log(i.toString()); // 'XE7338O073KYGTWWZN0F2WZ0R8PX5ZPPZS'
```

JavaScript Runtime Environment

Ethereum implements a **javascript runtime environment** (JSRE) that can be used in either interactive (console) or non-interactive (script) mode.

Ethereum's Javascript console exposes the full web3 JavaScript Dapp API and the admin API.

Interactive use: the JSRE REPL Console

The `ethereum CLI` executable `geth` has a JavaScript console (a **Read, Evaluate & Print Loop** = REPL exposing the JSRE), which can be started with the `console` or `attach` subcommand. The `console` subcommands starts the geth node and then opens the console. The `attach` subcommand will not start the geth node but instead tries to open the console on a running geth instance.

```
$ geth console
$ geth attach
```

The attach node accepts an endpoint in case the geth node is running with a non default ipc endpoint or you would like to connect over the rpc interface.

```
$ geth attach ipc:/some/custom/path
$ geth attach rpc:http://191.168.1.1:8545
```

Note that by default the geth node doesn't start the rpc service and not all functionality is provided over this interface due to security reasons. These defaults can be overridden when the `--rpcapi` argument when the geth node is started, or with admin.startRPC.

If you need log information, start with:

```
$ geth --verbosity 5 console 2>> /tmp/eth.log
```

Otherwise mute your logs, so that it does not pollute your console:

```
$ geth console 2>> /dev/null
```

or

```
$ geth --verbosity 0 console
```

Note: Since the database can only be accessed by one process, this means you cannot run `geth console` if you have an instance of geth already running.

Non-interactive use: JSRE script mode

It's also possible to execute files to the JavaScript intepreter. The `console` and `attach` subcommand accept the `--exec` argument which is a javascript statement.

```
$ geth --exec "eth.blockNumber" attach
```

This prints the current block number of a running geth instance.

Or execute a script with more complex statements with:

```
$ geth --exec 'loadScript("/tmp/checkbalances.js")' attach
$ geth --jspath "/tmp" --exec 'loadScript("checkbalances.js")' attach
```

Find an example script here

Use the `--jspath <path/to/my/js/root>` to set a libdir for your js scripts. Parameters to `loadScript()` with no absolute path will be understood relative to this directory.

You can exit the console cleanly by typing `exit` or simply with `CTRL-C`.

Caveat

The go-ethereum JSRE uses the Otto JS VM which has some limitations:

- "use strict" will parse, but does nothing.
- The regular expression engine (re2/regexp) is not fully compatible with the ECMA5 specification.

Note that the other known limitation of Otto (namely the lack of timers) is taken care of. Ethereum JSRE implements both `setTimeout` and `setInterval`. In addition to this, the console provides `admin.sleep(seconds)` as well as a "blocktime sleep" method `admin.sleepBlocks(number)`.

Since `ethereum.js` uses the `bignumer.js` library (MIT Expat Licence), it is also autoloded.

Timers

In addition to the full functionality of JS (as per ECMA5), the ethereum JSRE is augmented with various timers. It implements `setInterval` , `clearInterval` , `setTimeout` , `clearTimeout` you may be used to using in browser windows. It also provides implementation for `admin.sleep(seconds)` and a block based timer, `admin.sleepBlocks(n)` which sleeps till the number of new blocks added is equal to or greater than `n` , think "wait for n confirmations".

Management APIs

Beside the official DApp API interface the go ethereum node has support for additional management API's. These API's are offered using JSON-RPC and follow the same conventions as used in the DApp API. The go ethereum package comes with a console client which has support for all additional API's.

How to

It is possible to specify the set of API's which are offered over an interface with the `--${interface}api` command line argument for the go ethereum daemon. Where `${interface}` can be `rpc` for the `http` interface or `ipc` for an unix socket on unix or named pipe on Windows.

For example, `geth --ipcapi "admin,eth,miner" --rpcapi "eth,web3"` will

- enable the admin, official DApp and miner API over the IPC interface
- enable the eth and web3 API over the RPC interface

Please note that offering an API over the `rpc` interface will give everyone access to the API who can access this interface (e.g. DApp's). So be careful which API's you enable. By default geth enables all API's over the `ipc` interface and only the db,eth,net and web3 API's over the `rpc` interface.

To determine which API's an interface provides the `modules` transaction can be used, e.g. over an `ipc` interface on unix systems:

```
echo '{"jsonrpc":"2.0","method":"modules","params":[],"id":1}' | nc -U $datadir/geth.ipc
```

will give all enabled modules including the version number:

```
{
   "id":1,
   "jsonrpc":"2.0",
   "result":{
      "admin":"1.0",
      "db":"1.0",
      "debug":"1.0",
      "eth":"1.0",
      "miner":"1.0",
      "net":"1.0",
      "personal":"1.0",
      "shh":"1.0",
      "txpool":"1.0",
      "web3":"1.0"
   }
}
```

Integration

These additional API's follow the same conventions as the official DApp API. Web3 can be extended and used to consume these additional API's.

The different functions are split into multiple smaller logically grouped API's. Examples are given for the Javascript console but can easily be converted to a rpc request.

2 examples:

- Console: `miner.start()`
- IPC: `echo '{"jsonrpc":"2.0","method":"miner_start","params":[],"id":1}' | nc -U $datadir/geth.ipc`
- RPC: `curl -X POST --data '{"jsonrpc":"2.0","method":"miner_start","params":[],"id":74}' localhost:8545`

With the number of THREADS as an arguments:

- Console: `miner.start(4)`
- IPC: `echo '{"jsonrpc":"2.0","method":"miner_start","params":[4],"id":1}' | nc -U $datadir/geth.ipc`
- RPC: `curl -X POST --data '{"jsonrpc":"2.0","method":"miner_start","params":[4],"id":74}' localhost:8545`

Management API Reference

- eth
    - sign
    - pendingTransactions
    - resend
- admin
    - addPeer
    - peers
    - nodeInfo
    - datadir
    - importChain
    - exportChain
    - chainSyncStatus
    - startRPC
    - stopRPC
    - verbosity
    - setSolc
    - sleepBlocks
    - startNatSpec
    - stopNatSpec
    - getContractInfo
    - register
    - registerUrl
- miner
    - start
    - stop
    - startAutoDAG
    - stopAutoDAG
    - makeDAG
    - hashrate
    - setExtra
    - [setGasPrice] (#minersetgasprice)
    - [setEtherbase] (#minersetetherbase)
- personal
    - newAccount
    - listAccounts
    - deleteAccount
    - unlockAccount
- txpool
    - status
- debug

Personal

The `personal` api exposes method for personal the methods to manage, control or monitor your node. It allows for limited file system access.

---

personal.listAccounts

```
personal.listAccounts
```

List all accounts

Return

collection with accounts

Example

`personal.listAccounts`

personal.newAccount

```
personal.newAccount(passwd)
```

Create a new password protected account

Return

`string` address of the new account

Example

`personal.newAccount("mypasswd")`

personal.deleteAccount

```
personal.deleteAccount(addr, passwd)
```

Delete the account with the given address and password

Return

indication if the account was deleted

Example

`personal.deleteAccount(eth.coinbase, "mypasswd")`

personal.unlockAccount

```
personal.unlockAccount(addr, passwd, duration)
```

Unlock the account with the given address, password and an optional duration (in seconds)

Return

`boolean` indication if the account was unlocked

Example

`personal.unlockAccount(eth.coinbase, "mypasswd", 300)`

TxPool

txpool.status

```
txpool.status
```

Number of pending/queued transactions

Return

`pending` all processable transactions

`queued` all non-processable transactions

Example

`txpool.status`

---

admin

The `admin` exposes the methods to manage, control or monitor your node. It allows for limited file system access.

---

admin.chainSyncStatus

```
admin.chainSyncStatus
```

Prints info on blockchain synching.

return

`blocksAvailable` , blocks which have not been downloaded

`blocksWaitingForImport` , downloaded blocks waiting before import

`estimate` , a (very rough) estimate before the node has imported all blocks

`importing` , blocks currently importing

admin.verbosity

```
admin.verbosity(level)
```

Sets logger verbosity level to *level*. 1-6: silent, error, warn, info, debug, detail

Example

```
> admin.verbosity(6)
```

admin.nodeInfo

```
admin.nodeInfo
```

Returns

information on the node.

Example

```
> admin.nodeInfo
{
   Name: 'Ethereum(G)/v0.9.36/darwin/go1.4.1',
   NodeUrl: 'enode://c32e13952965e5f7ebc85b02a2eb54b09d55f553161c6729695ea34482af933d0a4b
   NodeID: '0xc32e13952965e5f7ebc85b02a2eb54b09d55f553161c6729695ea34482af933d0a4b035efb5
   IP: '89.42.0.12',
   DiscPort: 30303,
   TCPPort: 30303,
   Td: '0',
   ListenAddr: '[::]:30303'
}
```

To connect to a node, use the enode-format nodeUrl as an argument to addPeer or with CLI param `bootnodes`.

---

admin.addPeer

```
admin.addPeer(nodeURL)
```

Pass a `nodeURL` to connect a to a peer on the network. The `nodeURL` needs to be in enode URL format. geth will maintain the connection until it shuts down and attempt to reconnect if the connection drops intermittently.

You can find out your own node URL by using nodeInfo or looking at the logs when the node boots up e.g.:

```
[P2P Discovery] Listening, enode://6f8a80d14311c39f35f516fa664deaaaa13e85b2f7493f37f6144d
```

Returns

`true` on success.

Example

```
> admin.addPeer('enode://6f8a80d14311c39f35f516fa664deaaaa13e85b2f7493f37f6144d86991ec012
```

---

admin.peers

```
admin.peers
```

Returns

an array of objects with information about connected peers.

Example

```
> admin.peers
[ { ID: '0x6cdd090303f394a1cac34ecc9f7cda18127eafa2a3a06de39f6d920b0e583e062a7362097c7c65
```

admin.importChain

```
admin.importChain(file)
```

Imports the blockchain from a marshalled binary format. **Note** that the blockchain is reset (to genesis) before the imported blocks are inserted to the chain.

Returns

`true` on success, otherwise `false` .

Example

```
admin.importChain('path/to/file')
// true
```

admin.exportChain

```
admin.exportChain(file)
```

Exports the blockchain to the given file in binary format.

Returns

`true` on success, otherwise `false` .

Example

```
admin.exportChain('path/to/file')
```

admin.startRPC

```
 admin.startRPC(host, portNumber, corsheader, modules)
```

Starts the HTTP server for the JSON-RPC.

Returns

`true` on success, otherwise `false` .

Example

```
admin.startRPC("127.0.0.1", 8545, "*", "web3,db,net,eth")
// true
```

admin.stopRPC

```
admin.stopRPC()
```

Stops the HTTP server for the JSON-RPC.

Returns

`true` on success, otherwise `false` .

Example

```
admin.stopRPC()
// true
```

admin.sleepBlocks

```
admin.sleepBlocks(n)
```

Sleeps for n blocks.

admin.datadir

```
admin.datadir
```

the directory this nodes stores its data

Returns

directory on success

Example

```
admin.datadir
'/Users/username/Library/Ethereum'
```

admin.setSolc

```
admin.setSolc(path2solc)
```

Set the solidity compiler

Returns

a string describing the compiler version when path was valid, otherwise an error

Example

```
admin.setSolc('/some/path/solc')
'solc v0.9.29
Solidity Compiler: /some/path/solc
'
```

admin.startNatSpec

```
 admin.startNatSpec()
```

activate NatSpec: when sending a transaction to a contract, Registry lookup and url fetching is used to retrieve authentic contract Info for it. It allows for prompting a user with authentic contract-specific confirmation messages.

admin.stopNatSpec

```
 admin.stopNatSpec()
```

deactivate NatSpec: when sending a transaction, the user will be prompted with a generic confirmation message, no contract info is fetched

admin.getContractInfo

```
admin.getContractInfo(address)
```

this will retrieve the contract info json for a contract on the address

Returns

returns the contract info object

Examples

```
> info = admin.getContractInfo(contractaddress)
> source = info.source
> abi = info.abiDefinition
```

admin.saveInfo

```
admin.saveInfo(contract.info, filename);
```

will write contract info json into the target file, calculates its content hash. This content hash then can used to associate a public url with where the contract info is publicly available and verifiable. If you register the codehash (hash of the code of the contract on contractaddress).

Returns

`contenthash` on success, otherwise `undefined` .

Examples

```
source = "contract test {\n" +
"   /// @notice will multiply `a` by 7.\n" +
"   function multiply(uint a) returns(uint d) {\n" +
"      return a * 7;\n" +
"   }\n" +
"} ";
contract = eth.compile.solidity(source).test;
txhash = eth.sendTransaction({from: primary, data: contract.code });
// after it is uncluded
contractaddress = eth.getTransactionReceipt(txhash);
filename = "/tmp/info.json";
contenthash = admin.saveInfo(contract.info, filename);
```

admin.register

```
admin.register(address, contractaddress, contenthash);
```

will register content hash to the codehash (hash of the code of the contract on contractaddress). The register transaction is sent from the address in the first parameter. The transaction needs to be processed and confirmed on the canonical chain for the registration to take effect.

Returns

`true` on success, otherwise `false` .

Examples

```
source = "contract test {\n" +
"   /// @notice will multiply `a` by 7.\n" +
"   function multiply(uint a) returns(uint d) {\n" +
"      return a * 7;\n" +
"   }\n" +
"} ";
contract = eth.compile.solidity(source).test;
txhash = eth.sendTransaction({from: primary, data: contract.code });
// after it is uncluded
contractaddress = eth.getTransactionReceipt(txhash);
filename = "/tmp/info.json";
contenthash = admin.saveInfo(contract.info, filename);
admin.register(primary, contractaddress, contenthash);
```

admin.registerUrl

```
admin.registerUrl(address, codehash, contenthash);
```

this will register a contant hash to the contract' codehash. This will be used to locate contract info json files. Address in the first parameter will be used to send the transaction.

Returns

`true` on success, otherwise `false` .

Examples

```
source = "contract test {\n" +
"   /// @notice will multiply `a` by 7.\n" +
"   function multiply(uint a) returns(uint d) {\n" +
"      return a * 7;\n" +
"   }\n" +
"} ";
contract = eth.compile.solidity(source).test;
txhash = eth.sendTransaction({from: primary, data: contract.code });
// after it is uncluded
contractaddress = eth.getTransactionReceipt(txhash);
filename = "/tmp/info.json";
contenthash = admin.saveInfo(contract.info, filename);
admin.register(primary, contractaddress, contenthash);
admin.registerUrl(primary, contenthash, "file://"+filename);
```

Miner

miner.start

```
miner.start(threadCount)
```

Starts mining on with the given `threadNumber` of parallel threads. This is an optional argument.

Returns

`true` on success, otherwise `false` .

Example

```
miner.start()
// true
```

---

miner.stop

```
miner.stop(threadCount)
```

Stops `threadCount` miners. This is an optional argument.

Returns

`true` on success, otherwise `false`.

Example

```
miner.stop()
// true
```

---

miner.startAutoDAG

```
miner.startAutoDAG()
```

Starts automatic pregeneration of the ethash DAG. This process make sure that the DAG for the subsequent epoch is available allowing mining right after the new epoch starts. If this is used by most network nodes, then blocktimes are expected to be normal at epoch transition. Auto DAG is switched on automatically when mining is started and switched off when the miner stops.

Returns

`true` on success, otherwise `false`.

---

miner.stopAutoDAG

```
miner.stopAutoDAG()
```

Stops automatic pregeneration of the ethash DAG. Auto DAG is switched off automatically when mining is stops.

Returns

`true` on success, otherwise `false`.

---

miner.makeDAG

```
miner.makeDAG(blockNumber, dir)
```

Generates the DAG for epoch `blockNumber/epochLength`. dir specifies a target directory, If `dir` is the empty string, then ethash will use the default directories `~/.ethash` on Linux and MacOS, and `~\AppData\Ethash` on Windows. The DAG file's name is `full-<revision-number>R-<seedhash>`

Returns

`true` on success, otherwise `false`.

---

miner.hashrate

```
miner.hashrate
```

Returns

Returns the current hash rate in H/s.

---

miner.setExtra

```
miner.setExtra("extra data")
```

Sets the extra data for the block when finding a block. Limited to 32 bytes.

---

miner.setGasPrice

```
miner.setGasPrice(gasPrice)
```

Sets the the gasprice for the miner

---

miner.setEtherbase

```
miner.setEtherbase(account)
```

Sets the the ether base, the address that will receive mining rewards.

Debug

debug.setHead

```
debug.setHead(blockNumber)
```

Sets the current head of the blockchain to the block referred to by *blockNumber*. See web3.eth.getBlock for more details on block fields and lookup by number or hash.

Returns

`true` on success, otherwise `false` .

Example

```
debug.setHead(eth.blockNumber-1000)
```

---

debug.seedHash

```
debug.seedHash(blockNumber)
```

Returns the hash for the epoch the given block is in.

Returns

hash in hex format

Example

```
> debug.seedHash(eth.blockNumber)
'0xf2e59013a0a379837166b59f871b20a8a0d101d1c355ea85d35329360e69c000'
```

debug.processBlock

```
debug.processBlock(blockNumber)
```

Processes the given block referred to by *blockNumber* with the VM in debug mode. See web3.eth.getBlock for more details on block fields and lookup by number or hash. In combination with `setHead`, this can be used to replay processing of a block to debug VM execution.

Returns

`true` on success, otherwise `false`.

Example

```
debug.processBlock(140101)
```

debug.getBlockRlp

```
debug.getBlockRlp(blockNumber)
```

Returns the hexadecimal representation of the RLP encoding of the block. See web3.eth.getBlock for more details on block fields and lookup by number or hash.

Returns

The hex representation of the RLP encoding of the block.

Example

```
> debug.getBlockRlp(131805)    'f90210f9020ba0ea4dcb53fe575e23742aa30266722a15429b7ba3d33
```

debug.printBlock

```
debug.printBlock(blockNumber)
```

Prints information about the block such as size, total difficulty, as well as header fields properly formatted.

See web3.eth.getBlock for more details on block fields and lookup by number or hash.

Returns

formatted string representation of the block

Example

```
> debug.printBlock(131805)
BLOCK(be465b020fdbedc4063756f0912b5a89bbb4735bd1d1df84363e05ade0195cb1): Size: 531.00 B T
NoNonce: ee48752c3a0bfe3d85339451a5f3f411c21c8170353e450985e1faab0a9ac4cc
Header:
[

        ParentHash:         ea4dcb53fe575e23742aa30266722a15429b7ba3d33ba8c87012881d7a77e
        UncleHash:          1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49
        Coinbase:           a4d8e9cae4d04b093aac82e6cd355b6b963fb7ff
        Root:               1f892bfd6f8fb2ec69f30c8799e371c24ebc5a9d55558640de1fb7ca8787d
        TxSha               56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b
        ReceiptSha:         56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b
        Bloom:              0000000000000000000000000000000000000000000000000000000000000
        Difficulty:         12292710
        Number:             131805
        GasLimit:           3141592
        GasUsed:            0
        Time:               1429487725
        Extra:              ΞTHΞЯSPHΞЯΞ
        MixDigest:          4cf6d2c4022dfab72af44e9a58d7ac9f7238ffce31d4da72ed6ec9eda60e1
        Nonce:              3f9e9ce6a261381c
]
Transactions:
[]
Uncles:
[]
}
```

debug.dumpBlock

```
debug.dumpBlock(blockNumber)
```

Returns

the raw dump of a block referred to by block number or block hash or undefined if the block is not found. see web3.eth.getBlock for more details on block fields and lookup by number or hash.

Example

```
> debug.dumpBlock(eth.blockNumber)
```

---

debug.metrics

```
debug.metrics(raw)
```

Returns

Collection of metrics, see for more information this wiki page.

Example

```
> metrics(true)
```

---

loadScript

```
 loadScript('/path/to/myfile.js');
```

Loads a JavaScript file and executes it. Relative paths are interpreted as relative to `jspath` which is specified as a command line flag, see Command Line Options.

sleep

```
sleep(s)
```

Sleeps for s seconds.

setInterval

```
setInterval(s, func() {})
```

clearInterval

setTimeout

clearTimeout

---

web3

The `web3` exposes all methods of the JavaScript API.

---

net

The `net` is a shortcut for web3.net.

---

eth

The `eth` is a shortcut for web3.eth. In addition to the `web3` and `eth` interfaces exposed by web3.js a few additional calls are exposed.

---

eth.sign

```
eth.sign(signer, data)
```

eth.pendingTransactions

```
eth.pendingTransactions
```

Returns pending transactions that belong to one of the users `eth.accounts`.

---

eth.resend

```
eth.resend(tx, <optional gas price>, <optional gas limit>)
```

Resends the given transaction returned by `pendingTransactions()` and allows you to overwrite the gas price and gas limit of the transaction.

Example

```
eth.sendTransaction({from: eth.accounts[0], to: "...", gasPrice: "1000"})
var tx = eth.pendingTransactions()[0]
eth.resend(tx, web3.toWei(10, "szabo"))
```

---

shh

The `shh` is a shortcut for web3.shh.

---

db

The `db` is a shortcut for web3.db.

---

inspect

The `inspect` method pretty prints the given value (supports colours)

Introduction

Ethereum is a platform that is intended to allow people to easily write decentralized applications (Đapps) using blockchain technology. A decentralized application is an application which serves some specific purpose to its users, but which has the important property that the application itself does not depend on any specific party existing. Rather than serving as a front-end for selling or providing a specific party's services, a Đapp is a tool for people and organizations on different sides of an interaction use to come together without any centralized intermediary.

Even necessary "intermediary" functions that are typically the domain of centralized providers, such as filtering, identity management, escrow and dispute resolution, are either handled directly by the network or left open for anyone to participate, using tools like internal token systems and reputation systems to ensure that users get access to high-quality services. Early examples of Đapps include BitTorrent for file sharing and Bitcoin for currency. Ethereum takes the primary developments used by BitTorrent and Bitcoin, the peer to peer network and the blockchain, and generalizes them in order to allow developers to use these technologies for any purpose.

The Ethereum blockchain can be alternately described as a blockchain with a built-in programming language, or as a consensus-based globally executed virtual machine. The part of the protocol that actually handles internal state and computation is referred to as the Ethereum Virtual Machine (EVM). From a practical standpoint, the EVM can be thought of as a large decentralized computer containing millions of objects, called "accounts", which have the ability to maintain an internal database, execute code and talk to each other.

There are two types of accounts:

1. **Externally owned account (EOAs)**: an account controlled by a private key, and if you own the private key associated with the EOA you have the ability to send ether and messages from it.
2. **Contract**: an account that has its own code, and is controlled by code.

By default, the Ethereum execution environment is lifeless; nothing happens and the state of every account remains the same. However, any user can trigger an action by sending a transaction from an externally owned account, setting Ethereum's wheels in motion. If the destination of the transaction is another EOA, then the transaction may transfer some ether but otherwise does nothing. However, if the destination is a contract, then the contract in turn activates, and automatically runs its code.

The code has the ability to read/write to its own internal storage (a database mapping 32-byte keys to 32-byte values), read the storage of the received message, and send messages to other contracts, triggering their execution in turn. Once execution stops, and all sub-executions triggered by a message sent by a contract stop (this all happens in a deterministic and synchronous order, ie. a sub-call completes fully before the parent call goes any further), the execution environment halts once again, until woken by the next transaction.

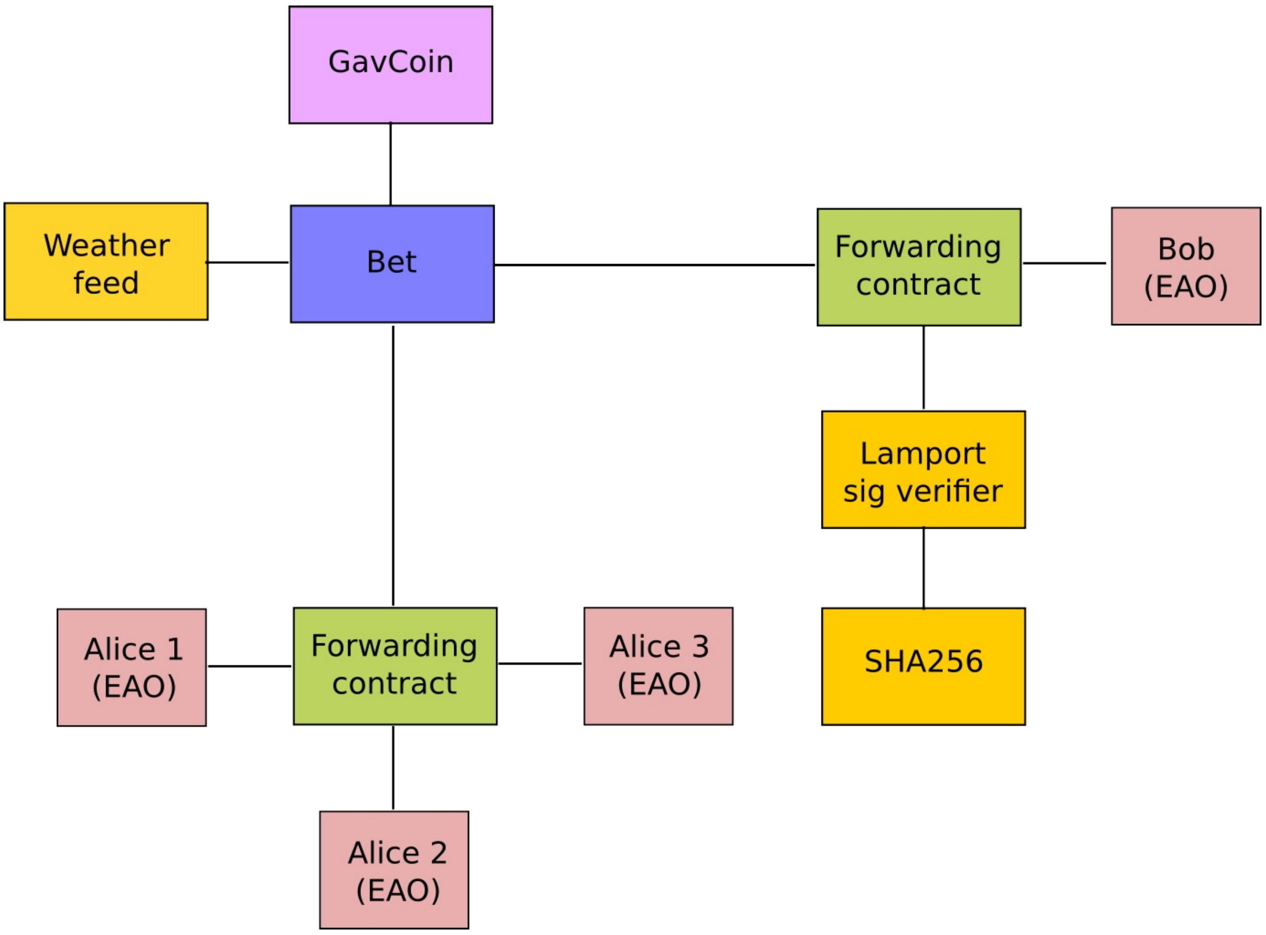
Contracts generally serve four purposes:

1. Maintain a data store representing something which is useful to either other contracts or to the outside world; one example of this is a contract that simulates a currency, and another is a contract that records membership in a particular organization.
2. Serve as a sort of externally owned account with a more complicated access policy; this is called a "forwarding contract" and typically involves simply resending incoming messages to some desired destination only if certain conditions are met; for example, one can have a forwarding contract that waits until two out of a given three private keys have confirmed a particular message before resending it (ie. multisig). More complex forwarding contracts have different conditions based on the nature of the message sent; the simplest use case for this functionality is a withdrawal limit that is overrideable via some more complicated access procedure.
3. Manage an ongoing contract or relationship between multiple users. Examples of this include a financial contract, an escrow with some particular set of mediators, or some kind of insurance. One can also have an open contract that one party leaves open for any other party to engage with at any time; one example of this is a contract that automatically pays a bounty to whoever submits a valid solution to some mathematical problem, or proves that it is providing some computational resource.
4. Provide functions to other contracts; essentially serving as a software library.

Contracts interact with each other through an activity that is alternately called either "calling" or "sending messages". A "message" is an object containing some quantity of ether (a special internal currency used in Ethereum with the primary purpose of paying transaction fees), a byte-array of data of any size, the addresses of a sender and a recipient. When a contract receives a message it has the option of returning some data, which the original sender of the message can then immediately use. In this way, sending a message is exactly like calling a function.

Because contracts can play such different roles, we expect that contracts will be interacting with each other. As an example, consider a situation where Alice and Bob are betting 100 GavCoin that the temperature in San Francisco will not exceed 35ºC at any point in the next year. However, Alice is very security-conscious, and as her primary account uses a forwarding contract which only sends messages with the approval of two out of three private keys. Bob is paranoid about quantum cryptography, so he uses a forwarding contract which passes along only messages that have been signed with Lamport signatures alongside traditional ECDSA (but because he's old fashioned, he prefers to use a version of Lamport sigs based on SHA256, which is not supported in Ethereum directly).

The betting contract itself needs to fetch data about the San Francisco weather from some contract, and it also needs to talk to the GavCoin contract when it wants to actually send the GavCoin to either Alice or Bob (or, more precisely, Alice or Bob's forwarding contract). We can show the relationships between the accounts thus:

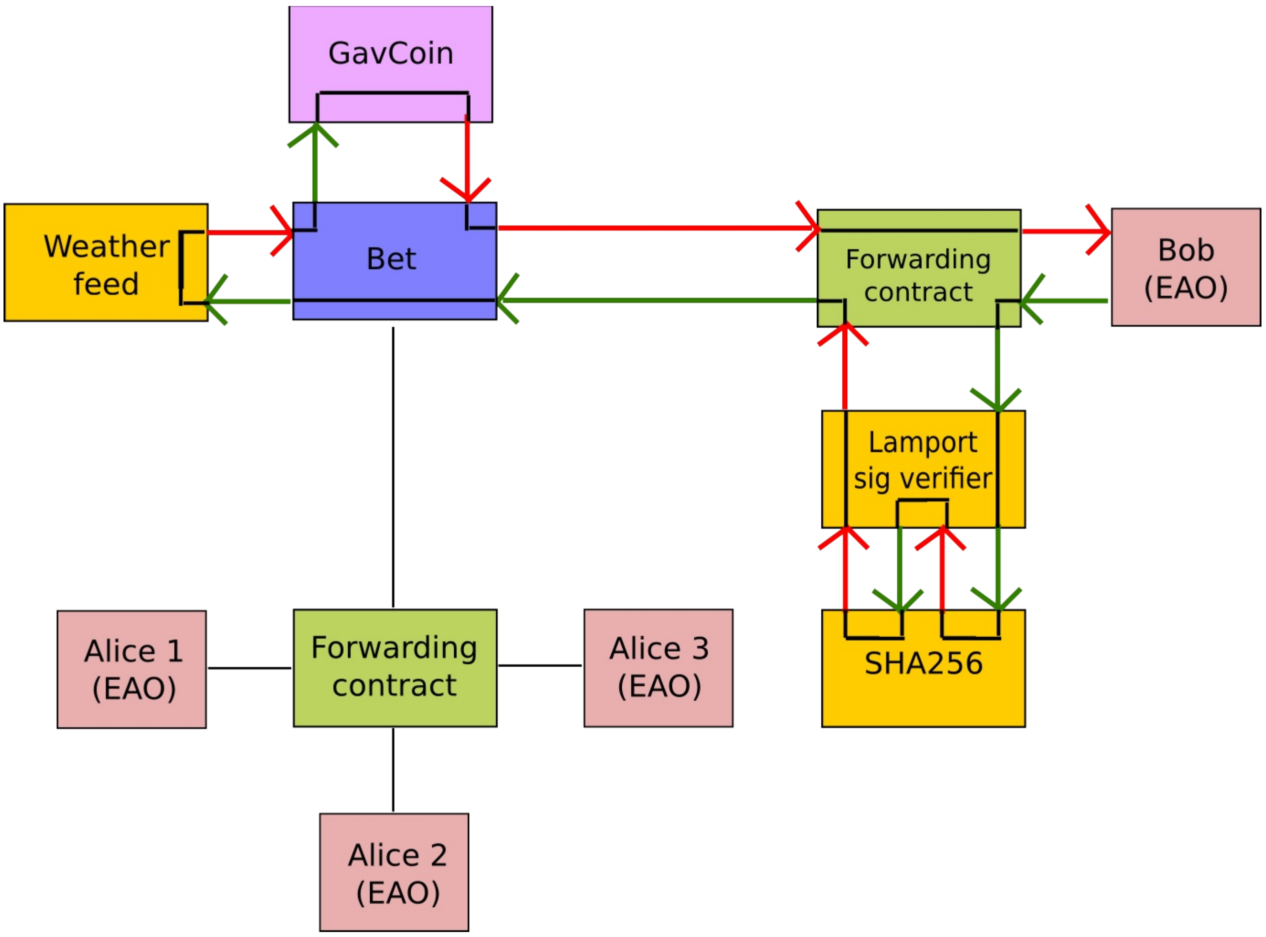


When Bob wants to finalize the bet, the following steps happen:

1. A transaction is sent, triggering a message from Bob's EOA to Bob's forwarding contract.
2. Bob's forwarding contract sends the hash of the message and the Lamport signature to a contract which functions as a Lamport signature verification library.
3. The Lamport signature verification library sees that Bob wants a SHA256-based Lamport sig, so it calls the SHA256 library many times as needed to verify the signature.
4. Once the Lamport signature verification library returns 1, signifying that the signature has been verified, it sends a message to the contract representing the bet.
5. The bet contract checks the contract providing the San Francisco temperature to see what the temperature is.

6. The bet contract sees that the response to the messages shows that the temperature is above 35ºC, so it sends a message to the GavCoin contract to move the GavCoin from its account to Bob's forwarding contract.

Note that the GavCoin is all "stored" as entries in the GavCoin contract's database; the word "account" in the context of step 6 simply means that there is a data entry in the GavCoin contract storage with a key for the bet contract's address and a value for its balance. After receiving this message, the GavCoin contract decreases this value by some amount and increases the value in the entry corresponding to Bob's forwarding contract's address. We can see these steps in the following diagram:



Account types and transactions

There are two types of accounts in Ethereum state:

- Normal or externally controlled accounts and
- contracts, i.e., snippets of code, think a class.

Both types of accounts have an ether balance.

Transactions can be fired from both types of accounts, though contracts only fire transactions in response to other transactions that they have received. Therefore, all action on the ethereum block chain is set in motion by transactions fired from externally-controlled accounts.

The simplest transactions are ether transfer transactions. But before we go into that you should read up on accounts and perhaps on mining.

Ether transfer

Assuming the account you are using as sender has sufficient funds, sending ether couldn't be easier. Which is also why you should probably be careful with this! You have been warned.

```
eth.sendTransaction({from: '0x036a03fc47084741f83938296a1c8ef67f6e34fa', to: '0xa8ade7fea
```

Note the unit conversion in the `value` field. Transaction values are expressed in weis, the most granular units of value. If you want to use some other unit (like `ether` in the example above), use the function `web3.toWei` for conversion.

Also, be advised that the amount debited from the source account will be slightly larger than that credited to the target account, which is what has been specified. The difference is a small transaction fee, discussed in more detail later.

Contracts can receive transfers just like externally controlled accounts, but they can also receive more complicated transactions that actually run (parts of) their code and update their state. In order to understand those transactions, a rudimentary understanding of contracts is required.

Writing a contract

Contracts live on the blockchain in an Ethereum-specific binary format (Ethereum Virtual Machine (=EVM) bytecode). However, contracts are typically written in some high level language such as solidity and then compiled into byte code to be uploaded on the blockchain.

Note that other languages also exist, notably serpent and LLL. Legacy Mutan (an early c-like language) is no longer officially maintained.

Language Resources

Solidity

Docs and tutorials

- Ethereum wiki tutorial
- Solidity FAQ - Ethereum forum
- The Solidity Programming Language · ethereum/wiki
- Ethereum ÐΞVcon-0: Solidity, Vision and Roadmap - YouTube Video
- Dapps for beginners
- Tutorial 1
- Tutorial 2
- Tutorial 3 (JavaScript API for Ethereum) (Outdated)
- Solidity tutorial 1 by Eris Industries
- Dapp tutorials by Andreas Olofsson (Eris Industries)
- Eris Solidity resources

Examples

- a dapp listing
- Solidity Contracts on Ethereum - Ether.Fund
- Ethereum dapp bin
- Solidity Standard Library
- Whisper chat Dapp written in meteor
- order statistic tree by Conrad Bars

Compilers

- Solidity realtime compiler

Serpent

- source on github
- serpent language spec

Contract/Dapp development environments and frameworks

- Mix standalone IDE by ETHDEV
- in-browser Cosmo that connects to `geth` via RPC. By Nick Dodson
- embark framework by Iuri Mathias
- truffle by Tim Coulter

Compiling a contract

Contracts live on the blockchain in an Ethereum-specific binary format (Ethereum Virtual Machine (=EVM) bytecode). However, contracts are typically written in some high level language such as solidity and then compiled into byte code to be uploaded on the blockchain.

For the frontier release, `geth` supports solidity compilation through system call to `solc`, the command line solidity compiler by Christian R. and Lefteris K. You can try Solidity realtime compiler (by Christian R) or Cosmo or Mix.

If you start up your `geth` node, you can check if the solidity compiler is available. This is what happens, if it is not:

```
> eth.compile.solidity("")
eth_compileSolidity method not available: solc (solidity compiler) not found
    at InvalidResponse (<anonymous>:-57465:-25)
    at send (<anonymous>:-115373:-25)
    at solidity (<anonymous>:-104109:-25)
    at <anonymous>:1:1
```

After you found a way to install `solc`, you make sure it's in the path. If `eth.getCompilers()` still does not find it (returns an empty array), you can set a custom path to the `solc` executable on the command line using th `solc` flag.

```
geth --datadir ~/frontier/00 --solc /usr/local/bin/solc --natspec
```

You can also set this option at runtime via the console:

```
> admin.setSolc("/usr/local/bin/solc")
solc v0.9.32
Solidity Compiler: /usr/local/bin/solc
Christian <c@ethdev.com> and Lefteris <lefteris@ethdev.com> (c) 2014-2015
true
```

Let us take this simple contract source:

```
> source = "contract test { function multiply(uint a) returns(uint d) { return a * 7; } }
```

This contract offers a unary method: called with a positive integer `a`, it returns `a * 7`.

You are ready to compile solidity code in the `geth` JS console using `eth.compile.solidity`:

```
> contract = eth.compile.solidity(source).test
{
  code: '605280600c6000396000f3006000357c0100000000000000000000000000000000000000000000
  info: {
    language: 'Solidity',
    languageVersion: '0',
    compilerVersion: '0.9.13',
    abiDefinition: [{
      constant: false,
      inputs: [{
        name: 'a',
        type: 'uint256'
      } ],
      name: 'multiply',
      outputs: [{
        name: 'd',
        type: 'uint256'
      } ],
      type: 'function'
    } ],
    userDoc: {
      methods: {
      }
    },
    developerDoc: {
      methods: {
      }
    },
    source: 'contract test { function multiply(uint a) returns(uint d) { return a * 7; }
  }
}
```

The compiler is also available via RPC and therefore via web3.js to any in-browser Đapp connecting to `geth` via RPC/IPC.

The following example shows how you interface `geth` via JSON-RPC to use the compiler.

```
./geth --datadir ~/eth/ --loglevel 6 --logtostderr=true --rpc --rpcport 8100 --rpccorsdom
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_compileSolidity","params":["contract
```

The compiler output for one source will give you contract objects each representing a single contract. The actual return value of `eth.compile.solidity` is a map of contract name -- contract object pairs. Since our contract's name is `test`, `eth.compile.solidity(source).test` will give you the contract object for the test contract containing the following fields:

- `code` : the compiled EVM code
- `info` : the rest of the metainfo the compiler outputs
  - `source` : the source code
  - `language` : contract language (Solidity, Serpent, LLL)
  - `languageVersion` : contract language version
  - `compilerVersion` : compiler version
  - `abiDefinition` : Application Binary Interface Definition
  - `userDoc` : NatSpec user Doc
  - `developerDoc` : NatSpec developer Doc

The immediate structuring of the compiler output (into `code` and `info`) reflects the two very different **paths of deployment**. The compiled EVM code is sent off to the blockchain with a contract creation transaction while the rest (info) will ideally live on the decentralised cloud as publicly verifiable metadata complementing the code on the blockchain.

If your source contains multiple contracts, the output will contain an entry for each contact, the corresponding contract info object can be retrieved with the name of the contract as attribute name. You can try this by inspecting the most current GlobalRegistrar code:

```
contracts = eth.compile.solidity(globalRegistrarSrc)
```

Creating and deploying a contract

Now that you got both an unlocked account as well as some funds, you can create a contract on the blockchain by sending a transaction to the empty address with the evm code as data. Simple, eh?

```
primaryAddress = eth.accounts[0]
MyContract = eth.contract(abi);
contact = MyContract.new(arg1, arg2, ...,{from: primaryAddress, data: evmCode})
```

All binary data is serialised in hexadecimal form. Hex strings always have a hex prefix `0x` .

Note that `arg1, arg2, ...` are the arguments for the contract constructor, in case it accepts any.

Also note that this step requires you to pay for execution. Your balance on the account (that you put as sender in the `from` field) will be reduced according to the gas rules of the VM once your transaction makes it into a block. More on that later. After some time, your transaction should appear included in a block confirming that the state it brought about is a consensus. Your contract now lives on the blockchain.

The asynchronous way of doing the same looks like this:

```
MyContract.new([arg1, arg2, ...,]{from: primaryAccount, data: evmCode}, function(err, con
  if (!err && contract.address)
    console.log(contract.address);
});
```

Gas and transaction costs

So how did you pay for all this? Under the hood, the transaction specified a gas limit and a gasprice, both of which could have been specified directly in the transaction object.

Gas limit is there to protect you from buggy code running until your funds are depleted. The product of `gasPrice` and `gas` represents the maximum amount of Wei that you are willing to pay for executing the transaction. What you specify as `gasPrice` is used by miners to rank transactions for inclusion in the blockchain. It is the price in Wei of one unit of gas, in which VM operations are priced.

The gas expenditure incurred by running your contract will be bought by the ether you have in your account at a price you specified in the transaction with `gasPrice`. If you do not have the ether to cover all the gas requirements to complete running your code, the processing aborts and all intermediate state changes roll back to the pre-transaction snapshot. The gas used up to the point where execution stopped were used after all, so the ether balance of your account will be reduced. These parameters can be adjusted on the transaction object fields `gas` and `gasPrice`. The `value` field is used the same as in ether transfer transactions between normal accounts. In other words transferring funds is available between any two accounts, either normal (i.e. externally controlled) or contract. If your contract runs out of funds, you should see an insufficient funds error.

For testing and playing with contracts you can use the test network or set up a private node (or cluster) potentially isolated from all the other nodes. If you then mine, you can make sure that your transaction will be included in the next block. You can see the pending transactions with:

```
eth.getBlock("pending", true).transactions
```

You can retrieve blocks by number (height) or by their hash:

```
genesis = eth.getBlock(0)
eth.getBlock(genesis.hash).hash == genesis.hash
true
```

Use `eth.blockNumber` to get the current blockchain height and the "latest" magic parameter to access the current head (newest block).

```
currentHeight = eth.blockNumber()
eth.getBlock("latest").hash == eth.getBlock(eth.blockNumber).hash
true
```

Interacting with contracts

`eth.contract` can be used to define a contract *class* that will comply with the contract interface as described in its ABI definition.

```
var Multiply7 = eth.contract(contract.info.abiDefinition);
var myMultiply7 = Multiply7.at(address);
```

Now all the function calls specified in the abi are made available on the contract instance. You can just call those methods on the contract instance and chain `sendTransaction(3, {from: address})` or `call(3)` to it. The difference between the two is that `call` performs a "dry run" locally, on your computer, while `sendTransaction` would actually submit your transaction for inclusion in the block chain and the results of its execution will eventually become part of the global consensus. In other words, use `call` , if you are interested only in the return value and use `sendTransaction` if you only care about "side effects" on the state of the contract.

In the example above, there are no side effects, therefore `sendTransaction` only burns gas and increases the entropy of the universe. All "useful" functionality is exposed by `call` :

```
myMultiply7.multiply.call(6)
42
```

Now suppose this contract is not yours, and you would like documentation or look at the source code. This is made possible by making available the contract info bundle and register it in the blockchain. The `admin` API provides convenience methods to fetch this bundle for any contract that chose to register. To see how it works, read about Contract Metadata or read the contract info deployment section of this document.

```
// get the contract info for contract address to do manual verification
var info = admin.getContractInfo(address) // lookup, fetch, decode
var source = info.source;
var abiDef = info.abiDefinition
```

Contract info (metadata)

In the previous sections we explained how you create a contract on the blockchain. Now we deal with the rest of the compiler output, the **contract metadata** or contract info. The idea is that

- contract info is uploaded somewhere identifiable by a `url` which is publicly accessible
- anyone can find out what the `url` is only knowing the contracts address

These requirements are achieved very simply by using a 2 step blockchain registry. The first step registers the contract code (hash) with a content hash in a contract called `HashReg`. The second step registers a url with the content hash in the `UrlHint` contract. These simple registry contracts will be part of the frontier proposition.

By using this scheme, it is sufficient to know a contract's address to look up the url and fetch the actual contract metadata info bundle. Read on to learn why this is good.

So if you are a conscientious contract creator, the steps are the following:

1. Deploy the contract itself to the blockchain
2. Get the contract info json file.
3. Deploy contract info json file to any url of your choice
4. Register codehash ->content hash -> url

The JS API makes this process very easy by providing helpers. Call `admin.register` to extract info from the contract, write out its json serialisation in the given file, calculates the content hash of the file and finally registers this content hash to the contract's code hash. Once you deployed that file to any url, you can use `admin.registerUrl` to register the url with your content hash on the blockchain as well. (Note that in case a fixed content addressed model is used as document store, the url-hint is no longer necessary.)

```
source = "contract test { function multiply(uint a) returns(uint d) { return a * 7; } }"
// compile with solc
contract = eth.compile.solidity(source).test
// create contract object
var MyContract = eth.contract(contract.info.abiDefinition)
// extracts info from contract, save the json serialisation in the given file,
contenthash = admin.saveInfo(contract.info, "~/dapps/shared/contracts/test/info.json")
// send off the contract to the blockchain
MyContract.new({from: primaryAccount, data: contract.code}, function(error, contract){
  if(!error && contract.address) {
    // calculates the content hash and registers it with the code hash in `HashReg`
    // it uses address to send the transaction.
    // returns the content hash that we use to register a url
    admin.register(primaryAccount, contract.address, contenthash)
    // here you deploy ~/dapps/shared/contracts/test/info.json to a url
    admin.registerUrl(primaryAccount, hash, url)
  }
});
```

NatSpec

This section will further elaborate what you can do with contracts and transactions building on a protocol NatSpec. Solidity implements smart comments doxigen style which then can be used to generate various facades meta documents of the code. One such use case is to generate custom messages for transaction confirmation that clients can prompt users with.

So we now extend the `multiply7` contract with a smart comment specifying a custom confirmation message (notice).

```
contract test {
   /// @notice Will multiply `a` by 7.
   function multiply(uint a) returns(uint d) {
       return a * 7;
   }
}
```

The comment has expressions in between backticks which are to be evaluated at the time the transaction confirmation message is presented to the user. The variables that refer to parameters of method calls then are instantiated in accordance with the actual transaction data sent by the user (or the user's dapp). NatSpec support for confirmation notices is fully implemented in `geth`. NatSpec relies on both the abi definition as well as the userDoc component to generate the proper confirmations. Therefore in order to access that, the contract needs to have registered its contract info as described above.

Let us see a full example. As a very conscientious smart contract dev, you first create your contract and deploy according to the recommended steps above:

```
source = "contract test {
   /// @notice Will multiply `a` by 7.
   function multiply(uint a) returns(uint d) {
       return a * 7;
   }
}"
contract = eth.compile.solidity(source).test
MyContract = eth.contract(contract.info.abiDefinition)
contenthash = admin.saveInfo(contract.info, "~/dapps/shared/contracts/test/info.json")
MyContract.new({from: primary, data: contract.code}, function(error, contract){
  if(!error && contract.address) {
    admin.register(primary, contract.address, contenthash)
    // put it up on your favourite oldworld site:
    admin.registerUrl(contentHash, "http://dapphub.com/test/info.json")
  }
});
```

Note that if we use content addressed storage system like swarm the second step is unnecessary, since the contenthash is (deterministically translates to) the unique address of the content itself.

For the purposes of a painless example just simply use the file url scheme (not exactly the cloud, but will show you how it works) without needing to deploy.

```
admin.registerUrl(contentHash, "file:///home/nirname/dapps/shared/contracts/test/info.jso
```

Now you are done as a dev, so swap seats as it were and pretend that you are a user who is sending a transaction to the infamous `multiply7` contract.

You need to start the client with the `--natspec` flag to enable smart confirmations and contractInfo fetching. You can also set it on the console with `admin.startNatSpec()` and `admin.stopNatSpec()` .

```
geth --natspec --unlock primary console 2>> /tmp/eth.log
```

Now at the console type:

```
// obtain the abi definition for your contract
var info = admin.getContractInfo(address)
var abiDef = info.abiDefinition
// instantiate a contract for transactions
var Multiply7 = eth.contract(abiDef);
var myMultiply7 = Multiply7.at(address);
```

And now try to send an actual transaction:

```
> myMultiply7.multiply.sendTransaction(6, {from: eth.accounts[0]})
NatSpec: Will multiply 6 by 7.
Confirm? [y/n] y
>
```

When this transaction gets included in a block, somewhere on a lucky miner's computer, 6 will get multiplied by 7, with the result ignored. Mission accomplished.

If the transaction is not picked up, we can see it with:

```
eth.pendingTransactions
```

This accumulates all the transactions sent, even the ones that were rejected and are not included in the current mined block (trans state). These latter can be shown by:

```
eth.getBlock("pending", true).transactions()
```

if you identify the index of your rejected transaction, you can resend it with modified gas limit and gas price (both optional parameters):

```
tx = eth.pendingTransactions[1]
eth.resend(tx, newGasPrice, newGasLimit)
```

Testing contracts and transactions

Often you need to resort to a low level strategy of testing and debugging contracts and transactions. This section introduces some debug tools and practices you can use. In order to test contracts and transactions without real-word consequences, you best test it on a private blockchain. This can be achieved with configuring an alternative network id (select a unique integer) and/or disable peers. It is recommended practice that for testing you use an alternative data directory and ports so that you never even accidentally clash with your live running node (assuming that runs using the defaults. Starting your `geth` with in VM debug mode with profiling and highest logging verbosity level is recommended:

```
geth --datadir ~/dapps/testing/00/ --port 30310 --rpcport 8110 --networkid 4567890 --nodi
```

Before you can submit any transactions, you need mine some ether on your private chain and for that you need an account. See the sections on Mining and Accounts

```
// create account. will prompt for password
personal.newAccount("mypassword");
// name your primary account, will often use it
primary = eth.accounts[0];
// check your balance (denominated in ether)
balance = web3.fromWei(eth.getBalance(primary), "ether");
```

```
// assume an existing unlocked primary account
primary = eth.accounts[0];

// mine 10 blocks to generate ether

// starting miner
miner.start(8);
// sleep for 10 blocks.
admin.sleepBlocks(10);
// then stop mining (just not to burn heat in vain)
miner.stop()  ;
balance = web3.fromWei(eth.getBalance(primary), "ether");
```

After you create transactions, you can force process them with the following lines:

```
miner.start(1);
admin.sleepBlocks(1);
miner.stop()  ;
```

you can check your pending transactions with

```
// shows transaction pool
txpool.status
// number of pending txs
eth.getBlockTransactionCount("pending");
// print all pending txs
eth.getBlock("pending", true).transactions
```

If you submitted contract creation transaction, you can check if the desired code actually got inserted in the current blockchain:

```
txhash = eth.sendTansaction({from:primary, data: code})
//... mining
contractaddress = eth.getTransactionReceipt(txhash);
eth.getCode(contractaddress)
```

Registrar services

The frontier chain comes with some very basic baselayer services, most of all the registrar. The registrar is composed of 3 components.

- GlobalRegistrar to associate names (strings) to accounts (addresses).
- HashReg to associate hashes to hashes (map any object to a 'content' hash.
- UrlHint to associate content hashes to a hint for the location of the content. This is needed only if content storage is not content addressed, otherwise content hash is already the content address. If it is used, content fetched from the url should hash to content hash. In order to check authenticity of content one can check if this verifies.

Create and deploy GlobalRegistrar, HashReg and UrlHint

If the registrar contracts are not hardcoded in the blockchain (they are not at the time of writing), the registrars need to be deployed at least once on every chain.

If you are on *the main live chain*, the address of the main global registrar is hardcoded in the latest clients and therefore *you do not need to do anything*. If you want to change this or you are on a private chain you need to deploy these contracts at least once:

```
primary = eth.accounts[0];

globalRegistrarAddr = admin.setGlobalRegistrar("", primary);
hashRegAddr = admin.setHashReg("", primary);
urlHintAddr = admin.setUrlHint("", primary);
```

You need to mine or wait till the txs are all picked up. Initialise the registrar on the new address and check if the other registrars' names resolve to the correct addresses:

```
registrar = GlobalRegistrar.at(globalRegistrarAddr);
primary == registrar.owner("HashReg");
primary == registrar.owner("UrlHint");
hashRegAddr == registrar.addr("HashReg");
urlHintAddr registrar.addr("UrlHint");
```

and the following ones return correct code:

```
eth.getCode(registrar.address);
eth.getCode(registrar.addr("HashReg"));
eth.getCode(registrar.addr("UrlHint"));
```

From the second time onwards on the same chain as well as on other nodes, you simply seed with the GlobalRegistrars address, the rest is handled through it.

```
primary = eth.accounts[0];
globalRegistrarAddr = "0x225178b4829bbe7c9f8a6d2e3d9d87b66ed57d4f"

// set the global registrar address
admin.setGlobalRegistrar(globalRegistrarAddr)
// set HashReg address via globalRegistrar
hashRegAddr = admin.setHashReg()
// set UrlHint address via globalRegistrar
urlHintAddr = admin.setUrlHint()

// (re)sets the registrar variable to a GlobalRegistrar contract instance
registrar = GlobalRegistrar.at(globalRegistrarAddr);
```

If this is successful, you should be able to check with the following commands if the registrar returns addresses:

```
registrar.owner("HashReg");
registrar.owner("UrlHint");
registrar.addr("HashReg");
registrar.addr("UrlHint");
```

and the following ones return correct code:

```
eth.getCode(registrar.address);
eth.getCode(registrar.addr("HashReg"));
eth.getCode(registrar.addr("UrlHint"));
```

Using the registrar services

Can provide useful interfaces between contracts and dapps.

Global registrar

To reserve a name register an account address with it, you need the following:

```
registrar.reserve.sendTransaction(name, {from:primary})
registrar.setAddress.sendTransaction (name, address, true, {from: primary})
```

You need to wait for the transactions to be picked up (or force mine them if you are on a private chain). To check you query the registrar:

```
registrar.owner(name)
registrar.addr(name)
```

HashReg and UrlHint

HashReg and UrlHint can be used with the following abis:

```
hashRegAbi = '[{"constant":false,"inputs":[],"name":"setowner","outputs":[],"type":"funct
urlHintAbi = '[{"constant":false,"inputs":[{"name":"_hash","type":"uint256"},{"name":"idx
```

setting up the contract instances:

```
hashReg = eth.contract(hashRegAbi).at(registrar.addr("HashReg")));
urlHint = eth.contract(UrlHintAbi).at(registrar.addr("UrlHint")));
```

Associate a content hash to a key hash:

```
hashReg.register.sendTransaction(keyhash, contenthash, {from:primary})
```

Associate a url to a content hash:

```
urlHint.register.sendTransaction(contenthash, url, {from:primary})
```

To check resolution:

```
contenthash = hashReg._hash(keyhash);
url = urlHint._url(contenthash);
```

Example script

The example script below demonstrates most features discussed in this tutorial. You can run it with the JSRE as `geth js script.js 2>>geth.log` . If you want to run this test on a local private chain, then start geth with:

```
geth --maxpeers 0 --networkid 123456 --nodiscover --unlock primary js script.js 2>> geth.
```

Note that `networkid` can be any arbitrary non-negative integer, 0 is always the live net.

```
personal.newAccount("")

primary = eth.accounts[0];
balance = web3.fromWei(eth.getBalance(primary), "ether");
personal.unlockAccount(primary, "00");
// miner.setEtherbase(primary)

miner.start(8); admin.sleepBlocks(10); miner.stop()  ;

// 0xc6d9d2cd449a754c494264e1809c50e34d64562b
primary = eth.accounts[0];
balance = web3.fromWei(eth.getBalance(primary), "ether");

globalRegistrarTxHash = admin.setGlobalRegistrar("0x0");
//'0x0'
globalRegistrarTxHash = admin.setGlobalRegistrar("", primary);
//'0xa69690d2b1a1dcda78bc7645732bb6eefcd6b188eaa37abc47a0ab0bd87a02e8'
miner.start(1); admin.sleepBlocks(1); miner.stop();
//true
globalRegistrarAddr = eth.getTransactionReceipt(globalRegistrarTxHash).contractAddress;
//'0x3d255836f5f8c9976ec861b1065f953b96908b07'
eth.getCode(globalRegistrarAddr);
//...
admin.setGlobalRegistrar(globalRegistrarAddr);
registrar = GlobalRegistrar.at(globalRegistrarAddr);

hashRegTxHash = admin.setHashReg("0x0");
hashRegTxHash = admin.setHashReg("", primary);
txpool.status
miner.start(1); admin.sleepBlocks(1); miner.stop();
hashRegAddr = eth.getTransactionReceipt(hashRegTxHash).contractAddress;
eth.getCode(hashRegAddr);

registrar.reserve.sendTransaction("HashReg", {from:primary});
registrar.setAddress.sendTransaction("HashReg",hashRegAddr,true, {from:primary});
miner.start(1); admin.sleepBlocks(1); miner.stop();
```

```
registrar.owner("HashReg");
registrar.addr("HashReg");

urlHintTxHash = admin.setUrlHint("", primary);
miner.start(1); admin.sleepBlocks(1); miner.stop();
urlHintAddr = eth.getTransactionReceipt(urlHintTxHash).contractAddress;
eth.getCode(urlHintAddr);

registrar.reserve.sendTransaction("UrlHint", {from:primary});
registrar.setAddress.sendTransaction("UrlHint",urlHintAddr,true, {from:primary});
miner.start(1); admin.sleepBlocks(1); miner.stop();
registrar.owner("UrlHint");
registrar.addr("UrlHint");

globalRegistrarAddr = "0xfd719187089030b33a1463609b7dfea0e5de25f0"
admin.setGlobalRegistrar(globalRegistrarAddr);
registrar = GlobalRegistrar.at(globalRegistrarAddr);
admin.setHashReg("");
admin.setUrlHint("");

///// ///////////////////////////////

admin.stopNatSpec();
primary = eth.accounts[0];
personal.unlockAccount(primary, "00")

globalRegistrarAddr = "0xfd719187089030b33a1463609b7dfea0e5de25f0";
admin.setGlobalRegistrar(globalRegistrarAddr);
registrar = GlobalRegistrar.at(globalRegistrarAddr);
admin.setHashReg("0x0");
admin.setHashReg("");
admin.setUrlHint("0x0");
admin.setUrlHint("");


registrar.owner("HashReg");
registrar.owner("UrlHint");
registrar.addr("HashReg")
registrar.addr("UrlHint");


/////////////////////////////////////
eth.getBlockTransactionCount("pending");
miner.start(1); admin.sleepBlocks(1); miner.stop();

source = "contract test {\n" +
"   /// @notice will multiply `a` by 7.\n" +
"   function multiply(uint a) returns(uint d) {\n" +
"      return a * 7;\n" +
"   }\n" +
"} ";
contract = eth.compile.solidity(source).test;
txhash = eth.sendTransaction({from: primary, data: contract.code});
```

```
eth.getBlock("pending", true).transactions;

miner.start(1); admin.sleepBlocks(1); miner.stop();
contractaddress = eth.getTransactionReceipt(txhash).contractAddress;
eth.getCode(contractaddress);

multiply7 = eth.contract(contract.info.abiDefinition).at(contractaddress);
fortytwo = multiply7.multiply.call(6);

/////////////////////////////////

// register a name for the contract
registrar.reserve.sendTransaction(primary,  {from: primary});
registrar.setAddress.sendTransaction("multiply7", contractaddress, true, {from: primary})
////////////////////////

admin.stopNatSpec();
filename = "/info.json";
contenthash = admin.saveInfo(contract.info, "/tmp" + filename);
admin.register(primary, contractaddress, contenthash);
eth.getBlock("pending", true).transactions;
miner.start(1); admin.sleepBlocks(1); miner.stop();

admin.registerUrl(primary, contenthash, "file://" + filename);
eth.getBlock("pending", true).transactions;
miner.start(1); admin.sleepBlocks(1); miner.stop();

////////////////////

// retrieve contract address using global registrar entry with 'multply7'
contractaddress = registrar.addr("multiply7);
// retrieve the info using the url
info = admin.getContractInfo(contractaddress);
multiply7 = eth.contract(info.abiDefinition).at(contractaddress);
// try Natspec
admin.startNatSpec();
fortytwo = multiply7.multiply.sendTransaction(6, { from: primary });
```

Introduction

Now that you mastered the basics on how to get started and how to send ether, it's time to get your hands dirty in what really makes ethereum stand out of the crowd: smart contracts. Smart contracts are pieces of code that live on the blockchain and execute commands exactly how they were told to. They can read other contracts, make decisions, send ether and execute other contracts. Contracts will exist and run as long as the whole network exists, and will only stop if they run out of gas or if they were programmed to self destruct.

What can you do with contracts? You can do almost anything really, but for this guide let's do some simple things: You will get funds through a crowdfunding that, if successful, will supply a radically transparent and democratic organization that will only obey its own citizens, will never swerve away from its constitution and cannot be censored or shut down. And all that in less than 300 lines of code.

So let's start now.

Your first citizen: the greeter

Now that you’ve mastered the basics of Ethereum, let’s move into your first serious contract. The Frontier is a big open territory and sometimes you might feel lonely, so our first order of business will be to create a little automatic companion to greet you whenever you feel lonely. We’ll call him the “Greeter”.

The Greeter is an intelligent digital entity that lives on the blockchain and is able to have conversations with anyone who interacts with it, based on its input. It might not be a talker, but it’s a great listener. Here is its code:

```
contract mortal {
    /* Define variable owner of the type address*/
    address owner;

    /* this function is executed at initialization and sets the owner of the contract */
    function mortal() { owner = msg.sender; }

    /* Function to recover the funds on the contract */
    function kill() { if (msg.sender == owner) suicide(owner); }
}

contract greeter is mortal {
    /* define variable greeting of the type string */
    string greeting;

    /* this runs when the contract is executed */
    function greeter(string _greeting) public {
        greeting = _greeting;
    }

    /* main function */
    function greet() constant returns (string) {
        return greeting;
    }
}
```

You'll notice that there are two different contracts in this code: *"mortal"* and *"greeter"*. This is because Solidity (the high level contract language we are using) has *inheritance*, meaning that one contract can inherit characteristics of another. This is very useful to simplify coding as common traits of contracts don't need to be rewritten every time, and all contracts can be written in smaller, more readable chunks. So by just declaring that *greeter is mortal* you inherited all characteristics from the "mortal" contract and kept the greeter code simple and easy to read.

The inherited characteristic *"mortal"* simply means that the greeter contract can be killed by its owner, to clean up the blockchain and recover funds locked into it when the contract is no longer needed. Contracts in ethereum are, by default, immortal and have no owner, meaning that once deployed the author has no special privileges anymore. Consider this before deploying.

Installing a compiler

Before you are able to Deploy it though, you'll need two things: the compiled code, and the Application Binary Interface, which is a sort of reference template that defines how to interact with the contract.

The first you can get by using a compiler. You should have a solidity compiler built in on your geth console. To test it, use this command:

```
eth.getCompilers()
```

If you have it installed, it should output something like this:

```
['Solidity' ]
```

If instead the command returns an error, then you need to install it.

Using an online compiler

If you don't have solC installed, we have a online solidity compiler available. But be aware that **if the compiler is compromised, your contract is not safe**. For this reason, if you want to use the online compiler we encourage you to host your own.

Install SolC on Ubuntu

Press control+c to exit the console (or type *exit*) and go back to the command line. Open the terminal and execute these commands:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
which solc
```

Take note of the path given by the last line, you'll need it soon.

Install SolC on Mac OSX

You need brew in order to install on your mac

```
brew install cpp-ethereum
brew linkapps cpp-ethereum
which solc
```

Take note of the path given by the last line, you'll need it soon.

Install SolC on Windows

You need chocolatey in order to install solc.

```
cinst -pre solC-stable
```

Windows is more complicated than that, you'll need to wait a bit more.

If you have the SolC Solidity Compiler installed, you need now reformat by removing spaces so it fits into a string variable (there are some online tools that will do this):

Compile from source

```
git clone https://github.com/ethereum/cpp-ethereum.git
mkdir cpp-ethereum/build
cd cpp-ethereum/build
cmake -DJSONRPC=OFF -DMINER=OFF -DETHKEY=OFF -DSERPENT=OFF -DGUI=OFF -DTESTS=OFF -DJSCONS
make -j4
make install
which solc
```

Linking your compiler in Geth

Now go back to the console and type this command to install solC, replacing *path/to/solc* to the path that you got on the last command you did:

```
admin.setSolc("path/to/solc")
```

Now type again:

```
eth.getCompilers()
```

If you now have solC installed, then congratulations, you can keep reading. If you don't, then go to our forums or subreddit and berate us on failing to make the process easier.

Compiling your contract

If you have the compiler installed, you need now reformat your contract by removing line-breaks so it fits into a string variable (there are some online tools that will do this):

```
var greeterSource = 'contract mortal { address owner; function mortal() { owner = msg.sen

var greeterCompiled = web3.eth.compile.solidity(greeterSource)
```

You have now compiled your code. Now you need to get it ready for deployment, this includes setting some variables up, like what is your greeting. Edit the first line below to something more interesting than 'Hello World!" and execute these commands:

```
var _greeting = "Hello World!"
var greeterContract = web3.eth.contract(greeterCompiled.greeter.info.abiDefinition);

var greeter = greeterContract.new(_greeting,{from:web3.eth.accounts[0], data: greeterComp
  if(!e) {

    if(!contract.address) {
      console.log("Contract transaction send: TransactionHash: " + contract.transactionHa

    } else {
      console.log("Contract mined! Address: " + contract.address);
      console.log(contract);
    }

  }
})
```

Using the online compiler

If you don't have solC installed, you can simply use the online compiler. Copy the source code above to the online solidity compiler and then your compiled code should appear on the left pane. Copy the code on the box labeled **Geth deploy** to a text file. Now change the first line to your greeting:

```
var _greeting = "Hello World!"
```

Now you can paste the resulting text on your geth window. Wait up to thirty seconds and you'll see a message like this:

```
Contract mined! address: 0xdaa24d02bad7e9d6a80106db164bad9399a0423e
```

You will probably be asked for the password you picked in the beginning, because you need to pay for the gas costs to deploying your contract. This contract is estimated to need 172 thousand gas to deploy (according to the online solidity compiler), at the time of writing, gas on the test net is priced at 1 to 10 microethers per unit of gas (nicknamed "szabo" = 1 followed by 12 zeroes in wei). To know the latest price in ether all you can see the latest gas prices at the network stats page and multiply both terms.

Notice that the cost is not paid to the ethereum developers, instead it goes to the *Miners*, people who are running computers who keep the network running. Gas price is set by the market of the current supply and demand of computation. If the gas prices are too high, you can be a miner and lower your asking price.

After less than a minute, you should have a log with the contract address, this means you've sucessfully deployed your contract. You can verify the deployed code (compiled) by using this command:

```
eth.getCode(greeter.address)
```

If it returns anything other than "0x" then congratulations! Your little Greeter is live! If the contract is created again (by performing another eth.sendTransaction), it will be published to a new address.

Run the Greeter

In order to call your bot, just type the following command in your terminal:

```
greeter.greet();
```

Since this call changes nothing on the blockchain, it returns instantly and without any gas cost. You should see it return your greeting:

```
'Hello World!'
```

Getting other people to interact with your code

In order to other people to run your contract they need two things: the address where the contract is located and the ABI (Application Binary Interface) which is a sort of user manual, describing the name of its functions and how to call them. In order to get each of them run these commands:

```
greeterCompiled.greeter.info.abiDefinition;
greeter.address;
```

Then you can instantiate a javascript object which can be used to call the contract on any machine connected to the network. Replace 'ABI' and 'address' to create a contract object in javascript:

```
var greeter = eth.contract(ABI).at(Address);
```

This particular example can be instantiated by anyone by simply calling:

```
var greeter2 = eth.contract([{constant:false,inputs:[],name:'kill',outputs:[],type:'funct
```

Replace *greeterAddress* with your contract's address.

Tip: if the solidity compiler isn't properly installed in your machine, you can get the ABI from the online compiler. To do so, use the code below carefully replacing *greeterCompiled.greeter.info.abiDefinition* with the abi from your compiler.

Cleaning up after yourself:

You must be very excited to have your first contract live, but this excitement wears off sometimes, when the owners go on to write further contracts, leading to the unpleasant sight of abandoned contracts on the blockchain. In the future, blockchain rent might be implemented in order to increase the scalability of the blockchain but for now, be a good citizen and humanely put down your abandoned bots.

Unlike last time we will not be making a call as we wish to change something on the blockchain. This requires a transaction be sent to the network and a fee to be paid for the changes made. The suicide is subsidized by the network so it will cost much less than a usual transaction.

```
greeter.kill.sendTransaction({from:eth.accounts[0]})
```

You can verify that the deed is done simply seeing if this returns 0:

```
eth.getCode(greeter.contractAddress)
```

Notice that every contract has to implement its own kill clause. In this particular case only the account that created the contract can kill it.

If you don't add any kill clause it could potentially live forever (or at least until the frontier contracts are all wiped) independently of you and any earthly borders, so before you put it live check what your local laws say about it, including any possible limitation on technology export, restrictions on speech and maybe any legislation on the civil rights of sentient digital beings. Treat your bots humanely.

Register a name for your coin

The commands mentioned only work because you have token javascript object instantiated on your local machine. If you send tokens to someone they won't be able to move them forward because they don't have the same object and wont know where to look for your contract or call its functions. In fact if you restart your console these objects will be deleted and the contracts you've been working on will be lost forever. So how do you instantiate the contract on a clean machine?

There are two ways. Let's start with the quick and dirty, providing your friends with a reference to your contract's ABI:

```
token = eth.contract([{constant:false,inputs:[{name:'receiver',type:'address'},{name:'amo
```

Just replace the address at the end for your own token address, then anyone that uses this snippet will immediately be able to use your contract. Of course this will work only for this specific contract so let's analyze step by step and see how to improve this code so you'll be able to use it anywhere.

All accounts are referenced in the network by their public address. But addresses are long, difficult to write down, hard to memorize and immutable. The last one is specially important if you want to be able to generate fresh accounts in your name, or upgrade the code of your contract. In order to solve this, there is a default name registrar contract which is used to associate the long addresses with short, human-friendly names.

Names have to use only alphanumeric characters and, cannot contain blank spaces. In future releases the name registrar will likely implement a bidding process to prevent name squatting but for now, it works on a first come first served basis: as long as no one else registered the name, you can claim it.

First, if you register a name, then you won't need the hardcoded address in the end. Select a nice coin name and try to reserve it for yourself. First, select your name:

```
var tokenName = "MyFirstCoin"
```

Then, check the availability of your name:

```
registrar.addr(tokenName)
```

If that function returns "0x00..", you can claim it to yourself:

```
registrar.reserve.sendTransaction(tokenName, {from: eth.accounts[0]});
```

Wait for the previous transaction to be picked up. Wait up to thirty seconds and then try:

```
registrar.owner(myName)
```

If it returns your address, it means you own that name and are able to set your chosen name to any address you want:

```
registrar.setAddress.sendTransaction(tokenName, token.address, true,{from: eth.accounts[0
```

*You can replace **token.address** for **eth.accounts[0]** if you want to use it as a personal nickname.*

Wait a little bit for that transaction to be picked up too and test it:

```
registrar.addr("MyFirstCoin")
```

You can send a transaction to anyone or any contract by name instead of account simply by typing

```
eth.sendTransaction({from: eth.accounts[0], to: registrar.addr("MyFirstCoin"), value: web
```

Tip: don't mix registrar.addr for registrar.owner. The first is to which address that name is pointed at: anyone can point a name to anywhere else, just like anyone can forward a link to google.com, but only the owner of the name can change and update the link. You can set both to be the same address.

This should now return your token address, meaning that now the previous code to instantiate could use a name instead of an address.

```
token = eth.contract([{constant:false,inputs:[{name:'receiver',type:'address'},{name:'amo
```

This also means that the owner of the coin can update the coin by pointing the registrar to the new contract. This would, of course, require the coin holders trust the owner set at registrar.owner("MyFirstCoin")

Of course this is a rather unpleasant big chunk of code just to allow others to interact with a contract. There are some avenues being investigated to upload the contract ABI to the network, so that all the user will need is the contract name. You can read about these approaches but they are very experimental and will certainly change in the future.

The Coin

What is a coin? Coins are much more interesting and useful than they seem, they are in essence just a tradeable token, but can become much more, depending on how you use them. Its value depends on what you do with it: a token can be used to control access (**an entrance ticket**), can be used for voting rights in an organization (**a share**), can be placeholders for an asset held by a third party (**a certificate of ownership**) or even be simply used as an exchange of value within a community (**a currency**).

You could do all those things by creating a centralized server, but using an Ethereum token contract comes with some free functionalities: for one, it's a decentralized service and tokens can be still exchanged even if the original service goes down for any reason. The code can guarantee that no tokens will ever be created other than the ones set in the original code. Finally, by having each user hold their own token, this eliminates the scenarios where one single server break-in can result in the loss of funds from thousands of clients.

You could create your own token on a different blockchain, but creating on ethereum is easier — so you can focus your energy on the innovation that will make your coin stand out - and it's more secure, as your security is provided by all the miners who are supporting the ethereum network. Finally, by creating your token in Ethereum, your coin will be compatible with any other contract running on ethereum.

The Code

This is the code for the contract we're building:

```
contract token {
    mapping (address => uint) public coinBalanceOf;
    event CoinTransfer(address sender, address receiver, uint amount);

  /* Initializes contract with initial supply tokens to the creator of the contract */
  function token(uint supply) {
        coinBalanceOf[msg.sender] = supply;
    }

  /* Very simple trade function */
    function sendCoin(address receiver, uint amount) returns(bool sufficient) {
        if (coinBalanceOf[msg.sender] < amount) return false;
        coinBalanceOf[msg.sender] -= amount;
        coinBalanceOf[receiver] += amount;
        CoinTransfer(msg.sender, receiver, amount);
        return true;
    }
}
```

If you have ever programmed, you won't find it hard to understand what it does: it is a contract that generates 10 thousand tokens to the creator of the contract, and then allows anyone with enough balance to send it to others. These tokens are the minimum tradeable unit and cannot be subdivided, but for the final users could be presented as a 100 units subdividable by 100 subunits, so owning a single token would represent having 0.01% of the total. If your application needs more fine grained atomic divisibility, then just increase the initial issuance amount.

In this example we declared the variable "coinBalanceOf" to be public, this will automatically create a function that checks any account's balance.

Compile and Deploy

So let's run it!

```
var tokenSource = ' contract token { mapping (address => uint) public coinBalanceOf; even

var tokenCompiled = eth.compile.solidity(tokenSource)
```

Now let’s set up the contract, just like we did in the previous section. Change the "initial Supply" to the amount of non divisible tokens you want to create. If you want to have divisible units, you should do that on the user frontend but keep them represented in the minimun unit of account.

```
var supply = 10000;
var tokenContract = web3.eth.contract(tokenCompiled.token.info.abiDefinition);
var token = tokenContract.new(
  supply,
  {
    from:web3.eth.accounts[0],
    data:tokenCompiled.token.code,
    gas: 1000000
  }, function(e, contract){
    if(!e) {

      if(!contract.address) {
        console.log("Contract transaction send: TransactionHash: " + contract.transaction

      } else {
        console.log("Contract mined! Address: " + contract.address);
        console.log(contract);
      }

    }
})
```

Online Compiler

If you don't have solC installed, you can simply use the online compiler. Copy the contract code to the online solidity compiler, if there are no errors on the contract you should see a text box labeled **Geth Deploy**. Copy the content to a text file so you can change the first line to set the initial supply, like this:

```
var supply = 10000;
```

Now you can paste the resulting text on your geth window. Wait up to thirty seconds and you'll see a message like this:

```
Contract mined! address: 0xdaa24d02bad7e9d6a80106db164bad9399a0423e
```

Check balance watching coin transfers

If everything worked correctly, you should be able to check your own balance with:

```
token.coinBalanceOf(eth.accounts[0]) + " tokens"
```

It should have all the 10 000 tokens that were created once the contract was published. Since there is not any other defined way for new coins to be issued, these are all that will ever exist.

You can set up a **Watcher** to react whenever anyone sends a coin using your contract. Here's how you do it:

```
var event = token.CoinTransfer({}, '', function(error, result){
  if (!error)
    console.log("Coin transfer: " + result.args.amount + " tokens were sent. Balances now
});
```

Sending coins

Now of course those tokens aren't very useful if you hoard them all, so in order to send them to someone else, use this command:

```
token.sendCoin.sendTransaction(eth.accounts[1], 1000, {from: eth.accounts[0]})
```

If a friend has registered a name on the registrar you can send it without knowing their address, doing this:

```
token.sendCoin.sendTransaction(registrar.addr("Alice"), 2000, {from: eth.accounts[0]})
```

Note that our first function **coinBalanceOf** was simply called directly on the contract instance and returned a value. This was possible since this was a simple read operation that incurs no state change and which executes locally and synchronously. Our second function **sendCoin** needs a **.sendTransaction()** call. Since this function is meant to change the state (write operation), it is sent as a transaction to the network to be picked up by miners and included in the canonical blockchain. As a result the consensus state of all participant nodes will adequately reflect the state changes resulting from executing the transaction. Sender address needs to be sent as part of the transaction to fund the fuel needed to run the transaction. Now, wait a minute and check both accounts balances:

```
token.coinBalanceOf.call(eth.accounts[0])/100 + "% of all tokens"
token.coinBalanceOf.call(eth.accounts[1])/100 + "% of all tokens"
token.coinBalanceOf.call(registrar.addr("Alice"))/100 + "% of all tokens"
```

Improvement suggestions

Right now this cryptocurrency is quite limited as there will only ever be 10,000 coins and all are controlled by the coin creator, but you can change that. You could for example reward ethereum miners, by creating a transaction that will reward who found the current block:

```
mapping (uint => address) miningReward;
function claimMiningReward() {
  if (miningReward[block.number] == 0) {
    coinBalanceOf[block.coinbase] += 1;
    miningReward[block.number] = block.coinbase;
  }
}
```

You could modify this to anything else: maybe reward someone who finds a solution for a new puzzle, wins a game of chess, install a solar panel—as long as that can be somehow translated to a contract. Or maybe you want to create a central bank for your personal country, so you can keep track of hours worked, favours owed or control of property. In that case you might want to add a function to allow the bank to remotely freeze funds and destroy tokens if needed.

Register a name for your coin

The commands mentioned only work because you have token javascript object instantiated on your local machine. If you send tokens to someone they won't be able to move them forward because they don't have the same object and wont know where to look for your contract or call its functions. In fact if you restart your console these objects will be deleted and the contracts you've been working on will be lost forever. So how do you instantiate the contract on a clean machine?

There are two ways. Let's start with the quick and dirty, providing your friends with a reference to your contract’s ABI:

```
token = eth.contract([{constant:false,inputs:[{name:'receiver',type:'address'},{name:'amo
```

Just replace the address at the end for your own token address, then anyone that uses this snippet will immediately be able to use your contract. Of course this will work only for this specific contract so let's analyze step by step and see how to improve this code so you'll be able to use it anywhere.

All accounts are referenced in the network by their public address. But addresses are long, difficult to write down, hard to memorize and immutable. The last one is specially important if you want to be able to generate fresh accounts in your name, or upgrade the code of your contract. In order to solve this, there is a default name registrar contract which is used to associate the long addresses with short, human-friendly names.

Names have to use only alphanumeric characters and, cannot contain blank spaces. In future releases the name registrar will likely implement a bidding process to prevent name squatting but for now, it works on a first come first served basis: as long as no one else registered the name, you can claim it.

First, if you register a name, then you won't need the hardcoded address in the end. Select a nice coin name and try to reserve it for yourself. First, select your name:

```
var tokenName = "MyFirstCoin"
```

Then, check the availability of your name:

```
registrar.addr(tokenName)
```

If that function returns "0x00..", you can claim it to yourself:

```
registrar.reserve.sendTransaction(tokenName, {from: eth.accounts[0]});
```

Wait for the previous transaction to be picked up. Wait up to thirty seconds and then try:

```
registrar.owner(myName)
```

If it returns your address, it means you own that name and are able to set your chosen name to any address you want:

```
registrar.setAddress.sendTransaction(tokenName, token.address, true,{from: eth.accounts[0
```

*You can replace **token.address** for **eth.accounts[0]** if you want to use it as a personal nickname.*

Wait a little bit for that transaction to be picked up too and test it:

```
registrar.addr("MyFirstCoin")
```

You can send a transaction to anyone or any contract by name instead of account simply by typing

```
eth.sendTransaction({from: eth.accounts[0], to: registrar.addr("MyFirstCoin"), value: web
```

Tip: don't mix registrar.addr for registrar.owner. The first is to which address that name is pointed at: anyone can point a name to anywhere else, just like anyone can forward a link to google.com, but only the owner of the name can change and update the link. You can set both to be the same address.

This should now return your token address, meaning that now the previous code to instantiate could use a name instead of an address.

```
token = eth.contract([{constant:false,inputs:[{name:'receiver',type:'address'},{name:'amo
```

This also means that the owner of the coin can update the coin by pointing the registrar to the new contract. This would, of course, require the coin holders trust the owner set at registrar.owner("MyFirstCoin")

Of course this is a rather unpleasant big chunk of code just to allow others to interact with a contract. There are some avenues being investigated to upload the contract ABI to the network, so that all the user will need is the contract name. You can read about these approaches but they are very experimental and will certainly change in the future.

Learn More

- Meta coin standard is a proposed standardization of function names for coin and token contracts, to allow them to be automatically added to other ethereum contract that utilizes trading, like exchanges or escrow.
- Formal proofing is a way where the contract developer will be able to assert some invariant qualities of the contract, like the total cap of the coin. *Not yet implemented*.

Crowdfund your idea

Sometimes a good idea takes a lot of funds and collective effort. You could ask for donations, but donors prefer to give to projects they are more certain that will get traction and proper funding. This is an example where a crowdfunding would be ideal: you set up a goal and a deadline for reaching it. If you miss your goal, the donations are returned, therefore reducing the risk for donors. Since the code is open and auditable, there is no need for a centralized trusted platform and therefore the only fees everyone will pay are just the gas fees.

In a crowdfunding prizes are usually given. This would require you to get everyone's contact information and keep track of who owns what. But since you just created your own token, why not use that to keep track of the prizes? This allows donors to immediately own something after they donated. They can store it safely, but they can also sell or trade it if they realize they don't want the prize anymore. If your idea is something physical, all you have to do after the project is completed is to give the product to everyone who sends you back a token. If the project is digital the token itself can immediately be used for users to participate or get entry on your project.

The code

The way this particular crowdsale contract works is that you set an exchange rate for your token and then the donors will immediately get a proportional amount of tokens in exchange of their ether. You will also choose a funding goal and a deadline: once that deadline is over you can ping the contract and if the goal was reached it will send the ether raised to you, otherwise it goes back to the donors. Donors keep their tokens even if the project doesn't reach its goal, as a proof that they helped.

```
contract token { mapping (address => uint) public coinBalanceOf; function token() {}  fun

contract Crowdsale {

    address public beneficiary;
    uint public fundingGoal; uint public amountRaised; uint public deadline; uint public 
    token public tokenReward;
    Funder[] public funders;
    event FundTransfer(address backer, uint amount, bool isContribution);

    /* data structure to hold information about campaign contributors */
    struct Funder {
        address addr;
        uint amount;
    }

    /*  at initialization, setup the owner */
    function Crowdsale(address _beneficiary, uint _fundingGoal, uint _duration, uint _pri
        beneficiary = _beneficiary;
        fundingGoal = _fundingGoal;
        deadline = now + _duration * 1 minutes;
        price = _price;
        tokenReward = token(_reward);
    }

    /* The function without name is the default function that is called whenever anyone s
    function () {
        uint amount = msg.value;
        funders[funders.length++] = Funder({addr: msg.sender, amount: amount});
        amountRaised += amount;
        tokenReward.sendCoin(msg.sender, amount / price);
        FundTransfer(msg.sender, amount, true);
    }

    modifier afterDeadline() { if (now >= deadline) _ }

    /* checks if the goal or time limit has been reached and ends the campaign */
    function checkGoalReached() afterDeadline {
        if (amountRaised >= fundingGoal){
            beneficiary.send(amountRaised);
            FundTransfer(beneficiary, amountRaised, false);
        } else {
            FundTransfer(0, 11, false);
            for (uint i = 0; i < funders.length; ++i) {
              funders[i].addr.send(funders[i].amount);
              FundTransfer(funders[i].addr, funders[i].amount, false);
            }
        }
        suicide(beneficiary);
    }
}
```

Set the parameters

Before we go further, let's start by setting the parameters of the crowdsale:

```
var _beneficiary = eth.accounts[1];    // create an account for this
var _fundingGoal = web3.toWei(100, "ether"); // raises 100 ether
var _duration = 30;     // number of minutes the campaign will last
var _price = web3.toWei(0.02, "ether"); // the price of the tokens, in ether
var _reward = token.address;   // the token contract address.
```

On Beneficiary put the new address that will receive the raised funds. The funding goal is the amount of ether to be raised. Deadline is measured in blocktimes which average 12 seconds, so the default is about 4 weeks. The price is tricky: but just change the number 2 for the amount of tokens the contributors will receive for each ether donated. Finally reward should be the address of the token contract you created in the last section.

In this example you are selling on the crowdsale half of all the tokens that ever existed, in exchange for 100 ether. Decide those parameters very carefully as they will play a very important role in the next part of our guide.

Deploy

You know the drill: if you are using the solC compiler,remove line breaks and copy the following commands on the terminal:

```
var crowdsaleCompiled = eth.compile.solidity(' contract token { mapping (address => uint)

var crowdsaleContract = web3.eth.contract(crowdsaleCompiled.Crowdsale.info.abiDefinition)
var crowdsale = crowdsaleContract.new(
  _beneficiary,
  _fundingGoal,
  _duration,
  _price,
  _reward,
  {
    from:web3.eth.accounts[0],
    data:crowdsaleCompiled.Crowdsale.code,
    gas: 1000000
  }, function(e, contract){
    if(!e) {

      if(!contract.address) {
        console.log("Contract transaction send: TransactionHash: " + contract.transaction

      } else {
        console.log("Contract mined! Address: " + contract.address);
        console.log(contract);
      }

    }    })
```

If you are using the *online compiler* Copy the contract code to the online solidity compiler, and then grab the content of the box labeled Geth Deploy**. Since you have already set the parameters, you don't need to change anything to that text, simply paste the resulting text on your geth window.**

Wait up to thirty seconds and you'll see a message like this:

```
Contract mined! address: 0xdaa24d02bad7e9d6a80106db164bad9399a0423e
```

If you received that alert then your code should be online. You can always double check by doing this:

```
eth.getCode(crowdsale.address)
```

Now fund your newly created contract with the necessary tokens so it can automatically distribute rewards to the contributors!

```
token.sendCoin.sendTransaction(crowdsale.address, 5000,{from: eth.accounts[0]})
```

After the transaction is picked, you can check the amount of tokens the crowdsale address has, and all other variables this way:

```
"Current crowdsale must raise " + web3.fromWei(crowdsale.fundingGoal.call(), "ether") + "
```

Put some watchers on

You want to be alerted whenever your crowdsale receives new funds, so paste this code:

```
var event = crowdsale.FundTransfer({}, '', function(error, result){
  if (!error)

    if (result.args.isContribution) {
        console.log("\n New backer! Received " + web3.fromWei(result.args.amount, "ether"

        console.log( "\n The current funding at " +( 100 *  crowdsale.amountRaised.call()

        var timeleft = Math.floor(Date.now() / 1000)-crowdsale.deadline();
        if (timeleft>3600) {  console.log("Deadline has passed, " + Math.floor(timeleft/3
        } else if (timeleft>0) {  console.log("Deadline has passed, " + Math.floor(timele
        } else if (timeleft>-3600) {  console.log(Math.floor(-1*timeleft/60) + " minutes
        } else {  console.log(Math.floor(-1*timeleft/3600) + " hours until deadline")
        }

    } else {
        console.log("Funds transferred from crowdsale account: " + web3.fromWei(result.ar
    }

});
```

Register the contract

You are now set. Anyone can now contribute by simply sending ether to the crowdsale address, but to make it even simpler, let's register a name for your sale. First, pick a name for your crowdsale:

```
var name = "mycrowdsale"
```

Check if that's available and register:

```
registrar.addr(name)
registrar.reserve.sendTransaction(name, {from: eth.accounts[0]});
```

Wait for the previous transaction to be picked up and then:

```
registrar.setAddress.sendTransaction(name, crowdsale.address, true,{from: eth.accounts[0]
```

Contribute to the crowdsale

Contributing to the crowdsale is very simple, it doesn't even require instantiating the contract. This is because the crowdsale responds to simple ether deposits, so anyone that sends ether to the crowdsale will automatically receive a reward. Anyone can contribute to it by simply executing this command:

```
var amount = web3.toWei(5, "ether") // decide how much to contribute

eth.sendTransaction({from: eth.accounts[0], to: crowdsale.address, value: amount, gas: 10
```

Alternatively, if you want someone else to send it, they can even use the name registrar to contribute:

```
eth.sendTransaction({from: eth.accounts[0], to: registrar.addr("mycrowdsale"), value: amo
```

Now wait a minute for the blocks to pickup and you can check if the contract received the ether by doing any of these commands:

```
web3.fromWei(crowdsale.amountRaised.call(), "ether") + " ether"
token.coinBalanceOf.call(eth.accounts[0]) + " tokens"
token.coinBalanceOf.call(crowdsale.address) + " tokens"
```

Recover funds

Once the deadline is passed someone has to wake up the contract to have the funds sent to either the beneficiary or back to the funders (if it failed). This happens because there is no such thing as an active loop or timer on ethereum so any future transactions must be pinged by someone.

```
crowdsale.checkGoalReached.sendTransaction({from:eth.accounts[0], gas: 2000000})
```

You can check your accounts with these lines of code:

```
web3.fromWei(eth.getBalance(eth.accounts[0]), "ether") + " ether"
web3.fromWei(eth.getBalance(eth.accounts[1]), "ether") + " ether"
token.coinBalanceOf.call(eth.accounts[0]) + " tokens"
token.coinBalanceOf.call(eth.accounts[1]) + " tokens"
```

The crowdsale instance is setup to self destruct once it has done its job, so if the deadline is over and everyone got their prizes the contract is no more, as you can see by running this:

```
eth.getCode(crowdsale.address)
```

So you raised a 100 ethers and successfully distributed your original coin among the crowdsale donors. What could you do next with those things?

Democracy DAO

So far you have created a tradeable token and you successfully distributed it among all those who were willing to help fundraise a 100 ethers. That's all very interesting but what exactly are those tokens for? Why would anyone want to own or trade it for anything else valuable? If you can convince your new token is the next big money maybe others will want it, but so far your token offers no value per se. We are going to change that, by creating your first decentralized autonomous organization, or DAO.

Think of the DAO as the constitution of a country, the executive branch of a government or maybe like a robotic manager for an organization. The DAO receives the money that your organization raises, keeps it safe and uses it to fund whatever its members want. The robot is incorruptible, will never defraud the bank, never create secret plans, never use the money for anything other than what its constituents voted on. The DAO will never disappear, never run away and cannot be controlled by anyone other than its own citizens.

The token we distributed using the crowdsale is the only citizen document needed. Anyone who holds any token is able to create and vote on proposals. Similar to being a shareholder in a company, the token can be traded on the open market and the vote is proportional to amounts of tokens the voter holds.

Take a moment to dream about the revolutionary possibilities this would allow, and now you can do it yourself, in under a 100 lines of code:

The Code

```
contract token { mapping (address => uint) public coinBalanceOf;   function token() { }


contract Democracy {

    uint public minimumQuorum;
    uint public debatingPeriod;
    token public voterShare;
    address public founder;
    Proposal[] public proposals;
    uint public numProposals;

    event ProposalAdded(uint proposalID, address recipient, uint amount, bytes32 data, st
    event Voted(uint proposalID, int position, address voter);
    event ProposalTallied(uint proposalID, int result, uint quorum, bool active);

    struct Proposal {
        address recipient;
        uint amount;
        bytes32 data;
        string description;
        uint creationDate;
        bool active;
        Vote[] votes;
        mapping (address => bool) voted;
    }

    struct Vote {
        int position;
        address voter;
    }

    function Democracy(token _voterShareAddress, uint _minimumQuorum, uint _debatingPerio
        founder = msg.sender;
        voterShare = token(_voterShareAddress);
        minimumQuorum = _minimumQuorum || 10;
        debatingPeriod = _debatingPeriod * 1 minutes || 30 days;
    }


    function newProposal(address _recipient, uint _amount, bytes32 _data, string _descrip
        if (voterShare.coinBalanceOf(msg.sender)>0) {
            proposalID = proposals.length++;
            Proposal p = proposals[proposalID];
            p.recipient = _recipient;
            p.amount = _amount;
            p.data = _data;
            p.description = _description;
            p.creationDate = now;
            p.active = true;
            ProposalAdded(proposalID, _recipient, _amount, _data, _description);
            numProposals = proposalID+1;
        }
    }

    function vote(uint _proposalID, int _position) returns (uint voteID){
        if (voterShare.coinBalanceOf(msg.sender)>0 && (_position >= -1 || _position <= 1
            Proposal p = proposals[_proposalID];
            if (p.voted[msg.sender] == true) return;
            voteID = p.votes.length++;
            p.votes[voteID] = Vote({position: _position, voter: msg.sender});
            p.voted[msg.sender] = true;
            Voted(_proposalID,  _position, msg.sender);
        }
    }

    function executeProposal(uint _proposalID) returns (int result) {
        Proposal p = proposals[_proposalID];
        /* Check if debating period is over */
        if (now > (p.creationDate + debatingPeriod) && p.active){
            uint quorum = 0;
            /* tally the votes */
            for (uint i = 0; i <  p.votes.length; ++i) {
                Vote v = p.votes[i];
                uint voteWeight = voterShare.coinBalanceOf(v.voter);
                quorum += voteWeight;
                result += int(voteWeight) * v.position;
            }
            /* execute result */
            if (quorum > minimumQuorum && result > 0 ) {
                p.recipient.call.value(p.amount)(p.data);
                p.active = false;
            } else if (quorum > minimumQuorum && result < 0) {
                p.active = false;
            }
            ProposalTallied(_proposalID, result, quorum, p.active);
        }
    }
}
```

There's a lot of going on but it's simpler than it looks. The rules of your organization are very simple: anyone with at least one token can create proposals to send funds from the country's account. After a week of debate and votes, if it has received votes worth a total of 100 tokens or more and has more approvals than rejections, the funds will be sent. If the quorum hasn't been met or it ends on a tie, then voting is kept until it's resolved. Otherwise, the proposal is locked and kept for historical purposes.

So let's recap what this means: in the last two sections you created 10,000 tokens, sent 1,000 of those to another account you control, 2,000 to a friend named Alice and distributed 5,000 of them via a crowdsale. This means that you no longer control over 50% of the votes in the DAO, and if Alice and the community bands together, they can outvote any spending decision on the 100 ethers raised so far. This is exactly how a democracy should work. If you don't want to be a part of your country anymore the only thing you can do is sell your own tokens on a decentralized exchange and opt out, but you cannot prevent the others from doing so.

Set Up your Organization

So open your console and let's get ready to finally put your country online. First, let's set the right parameters, pick them with care:

```
var _voterShareAddress = token.address;
var _minimumQuorum = 10; // Minimun amount of voter tokens the proposal needs to pass
var _debatingPeriod = 60; // debating period, in minutes;
```

With these default parameters anyone with any tokens can make a proposal on how to spend the organization's money. The proposal has 1 hour to be debated and it will pass if it has at least votes from at least 0.1% of the total tokens and has more support than rejections. Pick those parameters with care, as you won't be able to change them in the future.

```
var daoCompiled = eth.compile.solidity('contract token { mapping (address => uint) public

var democracyContract = web3.eth.contract(daoCompiled.Democracy.info.abiDefinition);

var democracy = democracyContract.new(
    _voterShareAddress,
    _minimumQuorum,
    _debatingPeriod,
    {
      from:web3.eth.accounts[0],
      data:daoCompiled.Democracy.code,
      gas: 3000000
    }, function(e, contract){
      if(!e) {

        if(!contract.address) {
          console.log("Contract transaction send: TransactionHash: " + contract.transacti

        } else {
          console.log("Contract mined! Address: " + contract.address);
          console.log(contract);
        }

      }
    })
```

If you are using the *online compiler* Copy the contract code to the online solidity compiler, and then grab the content of the box labeled Geth Deploy**. Since you have already set the parameters, you don't need to change anything to that text, simply paste the resulting text on your geth window.**

Wait a minute until the miners pick it up. It will cost you about 850k Gas. Once that is picked up, it's time to instantiate it and set it up, by pointing it to the correct address of the token contract you created previously.

If everything worked out, you can take a look at the whole organization by executing this string:

```
"This organization has " +  democracy.numProposals() + " proposals and uses the token at
```

If everything is setup then your DAO should return a proposal count of 0 and an address marked as the "founder". While there are still no proposals, the founder of the DAO can change the address of the token to anything it wants.

Register your organization name

Let's also register a name for your contract so it's easily accessible (don't forget to check your name availability with registrar.addr("nameYouWant") before reserving!)

```
var name = "MyPersonalDemocracy"
registrar.reserve.sendTransaction(name, {from: eth.accounts[0]})
var democracy = eth.contract(daoCompiled.Democracy.info.abiDefinition).at(democracy.addre
democracy.setup.sendTransaction(registrar.addr("MyFirstCoin"),{from:eth.accounts[0]})
```

Wait for the previous transactions to be picked up and then:

```
registrar.setAddress.sendTransaction(name, democracy.address, true,{from: eth.accounts[0]
```

The Democracy Watchbots

```
var event = democracy.ProposalAdded({}, '', function(error, result){
  if (!error)
    console.log("New Proposal #"+ result.args.proposalID +"!\n Send " + web3.fromWei(resu
});
var eventVote = democracy.Voted({}, '', function(error, result){
  if (!error)
    var opinion = "";
    if (result.args.position > 0) {
      opinion = "in favor"
    } else if (result.args.position < 0) {
      opinion = "against"
    } else {
      opinion = "abstaining"
    }

    console.log("Vote on Proposal #"+ result.args.proposalID +"!\n " + result.args.voter
});
var eventTally = democracy.ProposalTallied({}, '', function(error, result){
  if (!error)
    var totalCount = "";
    if (result.args.result > 1) {
      totalCount = "passed"
    } else if (result.args.result < 1) {
      totalCount = "rejected"
    } else {
      totalCount = "a tie"
    }
    console.log("Votes counted on Proposal #"+ result.args.proposalID +"!\n With a total
});
```

Interacting with the DAO

After you are satisfied with what you want, it's time to get all that ether you got from the crowdfunding into your new organization:

```
eth.sendTransaction({from: eth.accounts[1], to: democracy.address, value: web3.toWei(100,
```

This should take only a minute and your country is ready for business! Now, as a first priority, your organisation needs a nice logo, but unless you are a designer, you have no idea how to do that. For the sake of argument let's say you find that your friend Bob is a great designer who's willing to do it for only 10 ethers, so you want to propose to hire him.

```
recipient = registrar.addr("bob");
amount =  web3.toWei(10, "ether");
shortNote = "Logo Design";

democracy.newProposal.sendTransaction( recipient, amount, '', shortNote,  {from: eth.acco
```

After a minute, anyone can check the proposal recipient and amount by executing these commands:

```
"This organization has " +  (Number(democracy.numProposals())+1) + " pending proposals";
```

Keep an eye on the organization

Unlike most governments, your country's government is completely transparent and easily programmable. As a small demonstration here's a snippet of code that goes through all the current proposals and prints what they are and for whom:

```
function checkAllProposals() {
    console.log("Country Balance: " + web3.fromWei( eth.getBalance(democracy.address), "e
    for (i = 0; i< (Number(democracy.numProposals())); i++ ) {
        var p = democracy.proposals(i);
        var timeleft = Math.floor(((Math.floor(Date.now() / 1000)) - Number(p[4]) - Numbe
        console.log("Proposal #" + i + " Send " + web3.fromWei( p[1], "ether") + " ether
    }
}

checkAllProposals();
```

A concerned citizen could easily write a bot that periodically pings the blockchain and then publicizes any new proposals that were put forth, guaranteeing total transparency.

Now of course you want other people to be able to vote on your proposals. You can check the crowdsale tutorial on the best way to register your contract app so that all the user needs is a name, but for now let's use the easier version. Anyone should be able to instantiate a local copy of your country in their computer by using this giant command:

```
democracy = eth.contract( [{ constant: true, inputs: [{ name: '', type: 'uint256' } ], na
```

Then anyone who owns any of your tokens can vote on the proposals by doing this:

```
var proposalID = 0;
var position = -1; // +1 for voting yea, -1 for voting nay, 0 abstains but counts as quor
democracy.vote.sendTransaction(proposalID, position, {from: eth.accounts[0], gas: 1000000

var proposalID = 1;
var position = 1; // +1 for voting yea, -1 for voting nay, 0 abstains but counts as quoru
democracy.vote.sendTransaction(proposalID, position, {from: eth.accounts[0], gas: 1000000
```

Unless you changed the basic parameters in the code, any proposal will have to be debated for at least a week until it can be executed. After that anyone—even a non-citizen—can demand the votes to be counted and the proposal to be executed. The votes are tallied and weighted at that moment and if the proposal is accepted then the ether is sent immediately and the proposal is archived. If the votes end in a tie or the minimum quorum hasn’t been reached, the voting is kept open until the stalemate is resolved. If it loses, then it's archived and cannot be voted again.

```
var proposalID = 1;
democracy.executeProposal.sendTransaction(proposalID, {from: eth.accounts[0], gas: 100000
```

If the proposal passed then you should be able to see Bob's ethers arriving on his address:

```
web3.fromWei(eth.getBalance(democracy.address), "ether") + " ether";
web3.fromWei(eth.getBalance(registrar.addr("bob")), "ether") + " ether";
```

Try for yourself: This is a very simple democracy contract, which could be vastly improved: currently, all proposals have the same debating time and are won by direct vote and simple majority. Can you change that so it will have some situations, depending on the amount proposed, that the debate might be longer or that it would require a larger majority? Also think about some way where citizens didn't need to vote on every issue and could temporarily delegate their votes to a special representative. You might have also noticed that we added a tiny description for each proposal. This could be used as a title for the proposal or could be a hash of a larger document describing it in detail.

Let's go exploring!

You have reached the end of this tutorial, but it's just the beginning of a great adventure. Look back and see how much you accomplished: you created a living, talking robot, your own cryptocurrency, raised funds through a trustless crowdfunding and used it to kickstart your own personal democratic organization.

For the sake of simplicity, we only used the democratic organization you created to send ether around, the native currency of ethereum. While that might be good enough for some, this is only scratching the surface of what can be done. In the ethereum network contracts have all the same rights as any normal user, meaning that your organization could do any of the transactions that you executed coming from your own accounts.

What could happen next?

- The greeter contract you created at the beginning could be improved to charge ether for its services and could funnel those funds into the DAO.
- The tokens you still control could be sold on a decentralized exchange or traded for goods and services to fund further develop the first contract and grow the organization.
- Your DAO could own its own name on the name registrar, and then change where it's redirecting in order to update itself if the token holders approved.
- The organization could hold not only ethers, but any kind of other coin created on ethereum, including assets whose value are tied to the bitcoin or dollar.
- The DAO could be programmed to allow a proposal with multiple transactions, some scheduled to the future. It could also own shares of other DAO's, meaning it could vote on larger organization or be a part of a federation of DAO's.
- The Token Contract could be reprogrammed to hold ether or to hold other tokens and distribute it to the token holders. This would link the value of the token to the value of other assets, so paying dividends could be accomplished by simply moving funds to the token address.

This all means that this tiny society you created could grow, get funding from third parties, pay recurrent salaries, own any kind of crypto-assets and even use crowdsales to fund its activities. All with full transparency, complete accountability and complete immunity from any human interference. While the network lives the contracts will execute exactly the code they were created to execute, without any exception, forever.

So what will your contract be? Will it be a country, a company, a non-profit group? What will your code do?

That's up to you.

Specifications of all ethereum technologies, languages, protocols, etc.

Whitepapers and design rationale

- Ethereum Whitepaper
- Design Rationale
- Ethereum Yellow Paper
- ÐΞVp2p Whitepaper (WiP)
- Ethash

Specs

- JavaScript API
- Generic JSON RPC
- [JSRE admin API](https://github.com/ethereum/go-ethereum/wiki/JavaScript-Console#console-api
- RLP
- ÐΞVp2p Wire Protocol
- Web3 Secret Storage
- Patricia Tree
- Wire protocol
- Light client protocol
- Solidity, Docs & ABI
- NatSpec
- Contract ABI
- Ethash
- Ethash C API
- Ethash DAG
- ICAP: Inter-exchange Client Address Protocol

Peer-to-peer communications between nodes running Ethereum/Whisper/&c. clients are designed to be governed by a simple wire-protocol making use of existing ÐΞV technologies and standards such as RLP wherever practical.

This document is intended to specify this protocol comprehensively.

Low-Level

ÐΞVp2p nodes may connect to each other over TCP only. Peers are free to advertise and accept connections on any port(s) they wish, however, a default port on which the connection may be listened and made will be 30303.

Though TCP provides a connection-oriented medium, ÐΞVp2p nodes communicate in terms of packets. These packets are formed as a 4-byte synchronisation token (0x22400891), a 4-byte "payload size", to be interpreted as a big-endian integer and finally an N-byte RLP-serialised data structure, where N is the aforementioned "payload size". To be clear, the payload size specifies the number of bytes in the packet "following" the first 8.

Payload Contents

There are a number of different types of payload that may be encoded within the RLP. This "type" is always determined by the first entry of the RLP, interpreted as an integer.

ÐΞVp2p is designed to support arbitrary sub-protocols (aka *capabilities*) over the basic wire protocol. Each sub-protocol is given as much of the message-ID space as it needs (all such protocols must statically specify how many message IDs they require). On connection and reception of the `Hello` message, both peers have equivalent information about what subprotocols they share (including versions) and are able to form consensus over the composition of message ID space.

Message IDs are assumed to be compact from ID 0x10 onwards (0x00-0x10 is reserved for ÐΞVp2p messages) and given to each shared (equal-version, equal name) sub-protocol in alphabetic order. Sub-protocols that are not shared are ignored. If multiple versions are shared of the same (equal name) sub-protocol, the numerically highest wins, others are ignored.

P2P

Hello [`0x00` : `P` , `p2pVersion` : `P` , `clientId` : `B` , [[`cap1` : `B_3` , `capVersion1` : `P`], [`cap2` : `B_3` , `capVersion2` : `P`], `...`], `listenPort` : `P` , `nodeId` : `B_64`] First packet sent over the connection, and sent once by both sides. No other messages may be sent until a Hello is received.

- `p2pVersion` Specifies the implemented version of the P2P protocol. Now must be 1.
- `clientId` Specifies the client software identity, as a human-readable string (e.g. "Ethereum(++)/1.0.0").
- `cap` Specifies a peer capability name as a length-3 ASCII string. Current supported capabilities are `eth` , `shh` .
- `capVersion` Specifies a peer capability version as a positive integer. Current supported versions are 34 for `eth` , and 1 for `shh` .
- `listenPort` specifies the port that the client is listening on (on the interface that the present connection traverses). If 0 it indicates the client is not listening.
- `nodeId` is the Unique Identity of the node and specifies a 512-bit hash that identifies this node.

Disconnect [`0x01` : `P` , `reason` : `P`] Inform the peer that a disconnection is imminent; if received, a peer should disconnect immediately. When sending, well-behaved hosts give their peers a fighting chance (read: wait 2 seconds) to disconnect to before disconnecting themselves.

- `reason` is an optional integer specifying one of a number of reasons for disconnect:
    - `0x00` Disconnect requested;
    - `0x01` TCP sub-system error;
    - `0x02` Breach of protocol, e.g. a malformed message, bad RLP, incorrect magic number &c.;
    - `0x03` Useless peer;
    - `0x04` Too many peers;
    - `0x05` Already connected;
    - `0x06` Incompatible P2P protocol version;
    - `0x07` Null node identity received - this is automatically invalid;
    - `0x08` Client quitting;
    - `0x09` Unexpected identity (i.e. a different identity to a previous connection/what a trusted peer told us).
    - `0x0a` Identity is the same as this node (i.e. connected to itself);
    - `0x0b` Timeout on receiving a message (i.e. nothing received since sending last ping);
    - `0x10` Some other reason specific to a subprotocol.

Ping [`0x02` : `P`] Requests an immediate reply of `Pong` from the peer.

Pong [`0x03` : `P`] Reply to peer's `Ping` packet.

NotImplemented (was GetPeers) [`0x04` : `...`]

NotImplemented (was Peers) [`0x05` : `...`]

Node identity and reputation

In a later version of this protocol, node ID will become the public key. Nodes will have to demonstrate ownership over their ID by interpreting a packet encrypted with their node ID (or perhaps signing a random nonce with their private key).

A proof-of-work may be associated with the node ID through the big-endian magnitude of the public key. Nodes with a great proof-of-work (public key of lower magnitude) may be given preference since it is less likely that the node will alter its ID later or masquerade under multiple IDs.

Nodes are free to store ratings for given IDs (how useful the node has been in the past) and give preference accordingly. Nodes may also track node IDs (and their provenance) in order to help determine potential man-in-the-middle attacks.

Clients are free to mark down new nodes and use the node ID as a means of determining a node's reputation. In a future version of this wire protocol, n

Example Packets

`0x22400891000000088400000043414243`

A Hello packet specifying the client id is "ABC".

Peer 1: `0x22400891000000028102`

Peer 2: `0x22400891000000028103`

A Ping and the returned Pong.

Session Management

Upon connecting, all clients (i.e. both sides of the connection) must send a `Hello` message. Upon receiving the `Hello` message and verifying compatibility of the network and versions, a session is active and any other P2P messages may be sent.

At any time, a Disconnect message may be sent.

This is the main entry point for NatSpec and generally details a safe and efficient *standard* for ethereum contract metadata distribution.

By metadata we mean all information related to a contract that is thought to be relevant to and immutably linked to a specific version of a contract on the ethereum blockchain. This includes:

- Contract source code
- ABI definition
- NatSpec user doc
- NatSpec developer's doc

These resources have their *standard specification* in json format, ideally meant to be produced by IDE infrastructures or compilers directly.

For instance, the solidity compiler offers a `doxygen` style way of specifying natspec with inline smart comments. Upon compilation it creates both NatSpec user doc as well ABI definition. But note that there is nothing inherently solidity specific about these data, and other contract languages are encouraged to implement their NatSpec/ABI support potentially with IDE-s extending it.

Since DAPPs and IDEs will typically want to interact with these resources, standardising their deployment and distribution is important for a smooth ethereum experience.

A specially important example of this is the **NatSpec transaction confirmation notice scheme**, which we will use to illustrate the point. However, the strategy described here trivially extends to arbitrary immutable metadata fixed to an ethereum contract.

Transaction Confirmation Notice

The NatSpec user doc allows contract creators to attach custom confirmation notices to each method. A powerful feature of NatSpec is to provide templating which allows parts of the user notice to be instantiated depending on the parameters of the actual transaction sent to the contract.

Trusted ethereum client implementations are required to call back from their backend instantiating the natspec transaction notice from actual transaction data and present it to the user for confirmation.

This serves as a first line of defense against illegitimate transactions sent by malicious DAPPs in the user's name.

Surely, this is only feasible if

- the node can trust the authenticity of metadata sources.
- nodes have secure reliable access to the metadata resources

Let's see how this is achieved.

Metadata Authentication

Proposal The metadocs are assumed to be in a single JSON structure called `cmd` file, that stands for *contract metadata doc*.

The `cmd` file's content is hashed and content hash is registered on a name registry via a contract on the ethereum blockchain under the code hash, see https://github.com/ethereum/dapp-bin/blob/master/NatSpecReg/contract.sol *Registering* in this context will simply mean a key value pair is recorded in an immutable contract storage as a result of a transaction sent to the registry contract.

This provides a public immutable authentication for contract metadata, since:

- the authenticity of the link between the contract and metadata is secured by ethereum consensus
- the authenticity of actual metadata content is secured by content hashing
- the binding is tamper proof

DAPP IDE environments are supposed to support the functionality, that when you create a contract, all its standard metadata is

- bundled in a single `cmd` json file
- compiled source code is keccak hashed -> `Sha3(<evmcode>)`
- `cmd` json is keccak hashed -> `Sha3(<cmdjson>)`
- the metadata is registered with the contract by sending a transaction to a trusted name registry. The transaction simply records `Sha3(<evmcode>) -> Sha3(<cmdjson>)` as a key value pair in contract storage.

In order to avoid malicious agents hijacking metadata by overwriting the namereg entry, we propose the following business process:

- the link should be registered before a contract is deployed and is immutable from then on.
- before the corresponding contract is deployed, we check if the correct metadata bundle is found in the registry and confirmed to a satisfying level of certainty (X blocks).

How it looks like in `Alethzero` is illustrated [here]|(https://github.com/ethereum/wiki/wiki/NatSpec-Example)

Metadata Access

In order to provide a robust (failure resistant, 100% uptime) service, decentralised file storage services can and will be used. In the case of content addressed systems like Swarm, accessing and fetching the resource you will only need the `cmd` content hash (using `BZZHASH(<cmdjson>)`).

Until that point is reached, we will fall back to using HTTP distribution. To enable this we will include one or many URLHint contracts, which provide hints of URLs that allow downloading of particular content hashes. Find the contract in dapp-bin.

The content downloaded should be treated in many ways (and hashed) to discover what the content is. Possible ways include base 64 encoding, hex encoding and raw, and any content-cropping needed (e.g. a HTML page should have everything up to body tags removed).

It will be up to the dapp/content uploader to keep URLHint entries updated.

The address of the URLHint contract will be specified on an ad-hoc basis and users will be able to enter additional ones into their browser.

This functionality of uploading and registering location is supposed to be also supported by IDE and dev infrastructures. Developers (or IDE) are encouraged to register cmd location *before* content hash registration (and therefore before actual contract creation) in order to avoid metadata hijacking. Note that if content hash registration is correct, hijacking the location cannot allow malicious content (since content hash authenticates the `cmd`), however it can still prevent your metadata from being accessible. Therefore

- the url-hint contract storage should be by definition immutable
- content-hash to url-hint should be registered before contract creation

(surely with old web urls, we are always exposed to server failure/hacking, etc)

Read more here

Name registry contracts

Interoperability requires that clients know where to look to resolve a contract's metadata, i.e., which name registry to use. Currently, we solve this by

- creating the two registry contracts after the genesis block
- and hardwiring their addresses in the client code (e.g., in the natspec transaction confirmation notice module on the client side and the deployment module on the dev/IDE side).

Interop with ABI

Functions

Basic design

We assume the ABI is strongly typed, known at compilation time and static. No introspection mechanism will be provided. We assert that all contracts will have the interface definitions of any contracts they call available at compile-time.

This specification does not address contracts whose interface is dynamic or otherwise known only at run-time. Should these cases become important they can be adequately handled as facilities built within the Ethereum ecosystem.

Function Selector

The first four bytes of the call data for a function call specifies the function to be called. It is the first (left, high-order in big-endian) four bytes of the Keccak (SHA-3) hash of the signature of the function. The signature is defined as the canonical expression of the basic prototype, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma - no spaces are used.

Argument Encoding

Starting from the fifth byte, the encoded arguments follow. This encoding is also used in other places, e.g. the return values and also event arguments are encoded in the same way, without the four bytes specifying the function.

Types

The following elementary types exist:

- `uint<N>` : unsigned integer type of `N` bits, `0 < N <= 256` , `N % 8 == 0` . e.g. `uint32` , `uint8` , `uint256` .
- `int<N>` : two's complement signed integer type of `N` bits, `0 < N <= 256` , `N % 8 == 0` .
- `address` : equivalent to `bytes20` , except for the assumed interpretation and language

typing.
- `uint` , `int` : synonyms for `uint256` , `int256` respectively (not to be used for computing the function selector).
- `bool` : equivalent to `uint8` restricted to the values 0 and 1
- `real<N>x<M>` : fixed-point signed number of `N+M` bits, `0 < N + M <= 256` , `N % 8 == M % 8 == 0` . Corresponds to the int256 equivalent binary value divided by `2^M` .
- `ureal<N>x<M>` : unsigned variant of `real<N>x<M>` .
- `real` , `ureal` : synonyms for `real128x128` , `ureal128x128` respectively (not to be used for computing the function selector).
- `bytes<N>` : binary type of `N` bytes, `N >= 0` .

The following (fixed-size) array type exists:

- `<type>[N]` : a fixed-length array of the given fixed-length type.

The following non-fixed-size types exist:

- `bytes` : dynamic sized byte sequence.
- `string` : dynamic sized unicode string assumed to be UTF-8 encoded.
- `<type>[]` : a variable-length array of the given fixed-length type.

Formal Specification of the Encoding

We will now formally specify the encoding, such that it will have the following properties, which are especially useful if some arguments are nested arrays:

Properties:

1. The number of reads necessary to access a value is at most the depth of the value inside the argument array structure, i.e. four reads are needed to retrieve `a_i[k][l][r]` . In a previous version of the ABI, the number of reads scaled linearly with the total number of dynamic parameters in the worst case.

2. The data of a variable or array element is not interleaved with other data and it is relocatable, i.e. it only uses relative "addresses"

We distinguish static and dynamic types. Static types are encoded in-place and dynamic types are encoded at a separately allocated location after the current block.

Definition: The following types are called "dynamic":

- `bytes`
- `string`
- `T[]` for any `T`
- `T[k]` for any dynamic `T` and any `k > 0`

All other types are called "static".

Definition: `len(a)` is the number of bytes in a binary string `a`. The type of `len(a)` is assumed to be `uint256`.

We define `enc`, the actual encoding, as a mapping of values of the ABI types to binary strings such that `len(enc(X))` depends on the value of `X` if and only if the type of `X` is dynamic.

Definition: For any ABI value `X`, we recursively define `enc(X)`, depending on the type of `X` being

- `T[k]` for any `T` and `k`:

  `enc(X) = head(X[0]) ... head(X[k-1]) tail(X[0]) ... tail(X[k-1])`

  where `head` and `tail` are defined for `X[i]` being of a static type as `head(X[i]) = enc(X[i])` and `tail(X[i]) = ""` (the empty string) and as `head(X[i]) = enc(len(head(X[0]) ... head(X[k-1]) tail(X[0]) ... tail(X[i-1])))` `tail(X[i]) = enc(X[i])` otherwise.

  Note that in the dynamic case, `head(X[i])` is well-defined since the lengths of the head parts only depend on the types and not the values. Its value is the offset of the beginning of `tail(X[i])` relative to the start of `enc(X)`.

- `T[]` where `X` has `k` elements (`k` is assumed to be of type `uint256`):

  `enc(X) = enc(k) enc([X[1], ..., X[k]])`

  i.e. it is encoded as if it were an array of static size `k`, prefixed with the number of elements.

- `bytes`, of length `k` (which is assumed to be of type `uint256`):

  `enc(X) = enc(k) pad_right(X)`, i.e. the number of bytes is encoded as a `uint256` followed by the actual value of `X` as a byte sequence, followed by the minimum number of zero-bytes such that `len(enc(X))` is a multiple of 32.

- `string`:

  `enc(X) = enc(enc_utf8(X))`, i.e. `X` is utf-8 encoded and this value is interpreted as of `bytes` type and encoded further. Note that the length used in this subsequent encoding is the number of bytes of the utf-8 encoded string, not its number of characters.

- `uint<N>`: `enc(X)` is the big-endian encoding of `X`, padded on the higher-order (left) side with zero-bytes such that the length is a multiple of 32 bytes.

- `address`: as in the `uint160` case
- `int<N>`: `enc(X)` is the big-endian two's complement encoding of `X`, padded on the higher-oder (left) side with `0xff` for negative `X` and with zero bytes for positive `X`

such that the length is a multiple of 32 bytes.

- `bool` : as in the `uint8` case, where `1` is used for `true` and `0` for `false`
- `real<N>x<M>` : `enc(X)` is `enc(X * 2**M)` where `X * 2**M` is interpreted as a `int256` .
- `real` : as in the `real128x128` case
- `ureal<N>x<M>` : `enc(X)` is `enc(X * 2**M)` where `X * 2**M` is interpreted as a `uint256` .
- `ureal` : as in the `ureal128x128` case
- `bytes<N>` : `enc(X)` is the sequence of bytes in `X` padded with zero-bytes to a length of 32.

Note that for any `X` , `len(enc(X))` is a multiple of 32.

Function Selector and Argument Encoding

All in all, a call to the function `f` with parameters `a_1, ..., a_n` is encoded as

`function_selector(f) enc([a_1, ..., a_n])`

and the return values `v_1, ..., v_k` of `f` are encoded as

`enc([v_1, ..., v_k])`

where the types of `[a_1, ..., a_n]` and `[v_1, ..., v_k]` are assumed to be fixed-size arrays of length `n` and `k` , respectively. Note that strictly, `[a_1, ..., a_n]` can be an "array" with elements of different types, but the encoding is still well-defined as the assumed common type `T` (above) is not actually used.

Examples

Given the contract:

```
contract Foo {
  function bar(real[2] xy) {}
  function baz(uint32 x, bool y) returns (bool r) { r = x > 32 || y; }
  function sam(bytes name, bool z, uint[] data) {}
}
```

Thus for our `Foo` example if we wanted to call `baz` with the parameters `69` and `true` , we would pass 68 bytes total, which can be broken down into:

- `0xcdcd77c0` : the Method ID. This is derived as the first 4 bytes of the Keccak hash of the ASCII form of the signature `baz(uint32,bool)` .
- `0x0000000000000000000000000000000000000000000000000000000000000045` : the first

parameter, a uint32 value `69` padded to 32 bytes
- `0x0000000000000000000000000000000000000000000000000000000000000001` : the second parameter - boolean `true` , padded to 32 bytes

In total:

```
0xcdcd77c000000000000000000000000000000000000000000000000000000000000000045000000000000000
```

It returns a single `bool` . If, for example, it were to return `false` , its output would be the single byte array `0x0000000000000000000000000000000000000000000000000000000000000000` , a single bool.

If we wanted to call `bar` with the argument `[2.125, 8.5]` , we would pass 68 bytes total, broken down into:

- `0x3e279860` : the Method ID. This is derived from the signature `bar(real128x128[2])` . Note that `real` is substituted for its canonical representation `real128x128` .
- `0x0000000000000000000000000000000240000000000000000000000000000000` : the first part of the first parameter, a real128x128 value `2.125` .
- `0x0000000000000000000000000000000880000000000000000000000000000000` : the first part of the first parameter, a real128x128 value `8.5` .

In total:

```
0x3e27986000000000000000000000000000000002400000000000000000000000000000000000000000000000
```

If we wanted to call `sam` with the arguments `"dave"` , `true` and `[1,2,3]` , we would pass 292 bytes total, broken down into:

- `0x8FF261B0` : the Method ID. This is derived from the signature `sam(bytes,bool,uint256[])` . Note that `uint` is substituted for its canonical representation `uint256` .
- `0x0000000000000000000000000000000000000000000000000000000000000060` : the location of the data part of the first parameter (dynamic type), measured in bytes from the start of the arguments block. In this case, `0x60` .
- `0x0000000000000000000000000000000000000000000000000000000000000001` : the second parameter: boolean true.
- `0x00000000000000000000000000000000000000000000000000000000000000c0` : the location of the data part of the third parameter (dynamic type), measured in bytes. In this case, `0xc0` .
- `0x0000000000000000000000000000000000000000000000000000000000000004` : the data part of the first argument, it starts with the length of the byte array in elements, in this case, 4.

- `0x6461766500000000000000000000000000000000000000000000000000000000` : the contents of the first argument: the UTF-8 (equal to ASCII in this case) encoding of `"dave"` , padded on the right to 32 bytes.
- `0x0000000000000000000000000000000000000000000000000000000000000003` : the data part of the third argument, it starts with the length of the array in elements, in this case, 3.
- `0x0000000000000000000000000000000000000000000000000000000000000001` : the first entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000002` : the second entry of the third parameter.
- `0x0000000000000000000000000000000000000000000000000000000000000003` : the third entry of the third parameter.

In total:

```
0x8FF261B000000000000000000000000000000000000000000000000000000000000000600000000000000000
```

Use of Dynamic Types

A call to a function with the signature `f(uint,uint32[],bytes10,bytes)` with values `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` is encoded in the following way:

We take the first four bytes of `sha3("f(uint256,uint32[],bytes10,bytes)")` , i.e. `0x8be65246` . Then we encode the head parts of all four arguments. For the static types `uint256` and `bytes10` , these are directly the values we want to pass, whereas for the dynamic types `uint32[]` and `bytes` , we use the offset in bytes to the start of their data area, measured from the start of the value encoding (i.e. not counting the first four bytes containing the hash of the function signature). These are:

- `0x0000000000000000000000000000000000000000000000000000000000000123` (`0x123` padded to 32 bytes)
- `0x0000000000000000000000000000000000000000000000000000000000000080` (offset to start of data part of second parameter, 4*32 bytes, exactly the size of the head part)
- `0x3132333435363738393000000000000000000000000000000000000000000000` (`"1234567890"` padded to 32 bytes on the right)
- `0x00000000000000000000000000000000000000000000000000000000000000e0` (offset to start of data part of fourth parameter = offset to start of data part of first dynamic parameter + size of data part of first dynamic parameter = 4*32* + *3*32 (see below))

After this, the data part of the first dynamic argument, `[0x456, 0x789]` follows:

- `0x0000000000000000000000000000000000000000000000000000000000000002` (number of elements of the array, 2)
- `0x0000000000000000000000000000000000000000000000000000000000000456` (first element)

- `0x0000000000000000000000000000000000000000000000000000000000000789` (second element)

Finally, we encode the data part of the second dynamic argument, `"Hello, world!"` :

- `0x000000000000000000000000000000000000000000000000000000000000000d` (number of elements (bytes in this case): 13)
- `0x48656c6c6f2c20776f726c642100000000000000000000000000000000000000` (`"Hello, world!"` padded to 32 bytes on the right)

All together, the encoding is (spaces added for clarity):

```
0x8be65246 0000000000000000000000000000000000000000000000000000000000000123
0000000000000000000000000000000000000000000000000000000000000080
3132333435363738393000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000e0
0000000000000000000000000000000000000000000000000000000000000002
0000000000000000000000000000000000000000000000000000000000000456
0000000000000000000000000000000000000000000000000000000000000789
000000000000000000000000000000000000000000000000000000000000000d
48656c6c6f2c20776f726c642100000000000000000000000000000000000000
```

Events

Events are an abstraction of the Ethereum logging/event-watching protocol. Log entries provide the contract's address, a series of up to four topics and some arbitrary length binary data. Events leverage the existing function ABI in order to interpret this (together with an interface spec) as a properly typed structure.

Given an event name and series of event parameters, we split them into two sub-series: those which are indexed and those which are not. Those which are indexed, which may number up to 3, are used alongside the Keccak hash of the event signature to form the topics of the log entry. Those which as not indexed form the byte array of the event.

In effect, a log entry using this ABI is described as:

- `address` : the address of the contract (intrinsically provided by Ethereum);
- `topics[0]` : `keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` (`canonical_type_of` is a function that simply returns the canonical type of a given argument, e.g. for `uint indexed foo` , it would return `uint256`). If the event is declared as `anonymous` the `topics[0]` is not generated;
- `topics[n]` : `EVENT_INDEXED_ARGS[n - 1]` (`EVENT_INDEXED_ARGS` is the series of `EVENT_ARGS` that are indexed);
- `data` : `abi_serialise(EVENT_NON_INDEXED_ARGS)` (`EVENT_NON_INDEXED_ARGS` is the series of `EVENT_ARGS` that are not indexed, `abi_serialise` is the ABI serialisation function used for returning a series of typed values from a function, as described above).

JSON

The JSON format for a contract's interface is given by an array of function and/or event descriptions. A function description is a JSON object with the fields:

- `type` : `"function"` or `"constructor"` (can be omitted, defaulting to function);
- `name` : the name of the function (only present for function types);
- `inputs` : an array of objects, each of which contains:
- `name` : the name of the parameter;
- `type` : the canonical type of the parameter.
- `outputs` : an array of objects similar to `inputs` , can be omitted.

An event description is a JSON object with fairly similar fields:

- `type` : always `"event"`
- `name` : the name of the event;
- `inputs` : an array of objects, each of which contains:
- `name` : the name of the parameter;
- `type` : the canonical type of the parameter.
- `indexed` : `true` if the field is part of the log's topics, `false` if it one of the log's data segment.
- `anonymous` : `true` if the event was declared as `anonymous` .

For example,

```
contract Test {
function Test(){ b = 0x12345678901234567890123456789012; }
event Event(uint indexed a, bytes32 b)
event Event2(uint indexed a, bytes32 b)
function foo(uint a) { Event(a, b); }
bytes32 b;
}
```

would result in the JSON:

```
[{
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32","ind
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32","ind
"name":"Event2"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32","ind
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

Example Javascript Usage

```
var Test = eth.contract(
[{
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32","ind
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32","ind
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]);
var theTest = new Test(addrTest);

// examples of usage:
// every log entry ("event") coming from theTest (i.e. Event & Event2):
var f0 = eth.filter(theTest);
// just log entries ("events") of type "Event" coming from theTest:
var f1 = eth.filter(theTest.Event);
// also written as
var f1 = theTest.Event();
// just log entries ("events") of type "Event" and "Event2" coming from theTest:
var f2 = eth.filter([theTest.Event, theTest.Event2]);
// just log entries ("events") of type "Event" coming from theTest with indexed parameter
var f3 = eth.filter(theTest.Event, {'a': 69});
// also written as
var f3 = theTest.Event({'a': 69});
// just log entries ("events") of type "Event" coming from theTest with indexed parameter
var f4 = eth.filter(theTest.Event, {'a': [69, 42]});
// also written as
var f4 = theTest.Event({'a': [69, 42]});

// options may also be supplied as a second parameter with `earliest`, `latest`, `offset`
var options = { 'max': 100 };
var f4 = theTest.Event({'a': [69, 42]}, options);

var trigger;
f4.watch(trigger);

// call foo to make an Event:
theTest.foo(69);

// would call trigger like:
//trigger(theTest.Event, {'a': 69, 'b': '0x12345678901234567890123456789012'}, n);
// where n is the block number that the event triggered in.
```

Implementation:

```
// e.g. f4 would be similar to:
web3.eth.filter({'max': 100, 'address': theTest.address, 'topics': [ [69, 42] ]});
// except that the resultant data would need to be converted from the basic log entry for
{
  'address': theTest.address,
  'topics': [web3.sha3("Event(uint256,bytes32)"), 0x00...0045 /* 69 in hex format */],
  'data': '0x12345678901234567890123456789012',
  'number': n
}
// into data good for the trigger, specifically the three fields:
  Test.Event // derivable from the first topic
  {'a': 69, 'b': '0x12345678901234567890123456789012'} // derivable from the 'indexed' bo
  n // from the 'number'
```

Event result:

```
[ {
  'event': Test.Event,
  'args': {'a': 69, 'b': '0x12345678901234567890123456789012'},
  'number': n
  },
  { ...
  } ...
]
```

JUST DONE! [develop branch]

NOTE: THIS IS OLD - IGNORE IT unless reading for historical purposes

- Internal LogFilter, log-entry matching mechanism and eth_installFilter needs to support matching multiple values (OR semantics) *per* topic index (at present it will only match topics with AND semantics and set-inclusion, not per-index).

i.e. at present you can only ask for each of a number of given topic values to be matched throughout each topic:

- `topics: [69, 42, "Gav"]` would match against logs with 3 topics `[42, 69, "Gav"]`, `["Gav", 69, 42]` but **not** against logs with topics `[42, 70, "Gav"]`.

we need to be able to provide one of a number of topic values, and, each of these options for each topic index:

- `topics: [[69, 42], [] /* anything */, "Gav"]` should match against logs with 3 topics `[42, 69, "Gav"]`, `[42, 70, "Gav"]` but **not** against `["Gav", 69, 42]`.

Solidity Tutorial

See also Russian version (русский перевод)

English version moved to a new site.

State Machine

Computation in the EVM is done using a stack-based bytecode language that is like a cross between Bitcoin Script, traditional assembly and Lisp (the Lisp part being due to the recursive message-sending functionality). A program in EVM is a sequence of opcodes, like this:

```
PUSH1 0 CALLDATALOAD SLOAD NOT PUSH1 9 JUMPI STOP JUMPDEST PUSH1 32 CALLDATALOAD PUSH1 0
```

The purpose of this particular contract is to serve as a name registry; anyone can send a message containing 64 bytes of data, 32 for the key and 32 for the value. The contract checks if the key has already been registered in storage, and if it has not been then the contract registers the value at that key.

During execution, an infinitely expandable byte-array called "memory", the "program counter" pointing to the current instruction, and a stack of 32-byte values is maintained. At the start of execution, memory and stack are empty and the PC is zero. Now, let us suppose the contract with this code is being accessed for the first time, and a message is sent in with 123 wei (10^{18} wei = 1 ether) and 64 bytes of data where the first 32 bytes encode the number 54 and the second 32 bytes encode the number 2020202020.

Thus, the state at the start is:

```
PC: 0 STACK: [] MEM: [], STORAGE: {}
```

The instruction at position 0 is PUSH1, which pushes a one-byte value onto the stack and jumps two steps in the code. Thus, we have:

```
PC: 2 STACK: [0] MEM: [], STORAGE: {}
```

The instruction at position 2 is CALLDATALOAD, which pops one value from the stack, loads the 32 bytes of message data starting from that index, and pushes that on to the stack. Recall that the first 32 bytes here encode 54.

```
PC: 3 STACK: [54] MEM: [], STORAGE: {}
```

SLOAD pops one from the stack, and pushes the value in contract storage at that index. Since the contract is used for the first time, it has nothing there, so zero.

```
PC: 4 STACK: [0] MEM: [], STORAGE: {}
```

NOT pops one value and pushes 1 if the value is zero, else 0

```
PC: 5 STACK: [1] MEM: [], STORAGE: {}
```

Next, we PUSH1 9.

```
PC: 7 STACK: [1, 9] MEM: [], STORAGE: {}
```

The JUMPI instruction pops 2 values and jumps to the instruction designated by the first only if the second is nonzero. Here, the second is nonzero, so we jump. If the value in storage index 54 had not been zero, then the second value from top on the stack would have been 0 (due to NOT), so we would not have jumped, and we would have advanced to the STOP instruction which would have led to us stopping execution.

```
PC: 9 STACK: [] MEM: [], STORAGE: {}
```

Here, we PUSH1 32.

```
PC: 11 STACK: [32] MEM: [], STORAGE: {}
```

Now, we CALLDATALOAD again, popping 32 and pushing the bytes in message data starting from byte 32 until byte 63.

```
PC: 13 STACK: [2020202020] MEM: [], STORAGE: {}
```

Next, we PUSH1 0.

```
PC: 14 STACK: [2020202020, 0] MEM: [], STORAGE: {}
```

Now, we load message data bytes 0-31 again (loading message data is just as cheap as loading memory, so we don't bother to save it in memory)

```
PC: 16 STACK: [2020202020, 54] MEM: [], STORAGE: {}
```

Finally, we SSTORE to save the value 2020202020 in storage at index 54.

```
PC: 17 STACK: [] MEM: [], STORAGE: {54: 2020202020}
```

At index 17, there is no instruction, so we stop. If there was anything left in the stack or memory, it would be deleted, but the storage will stay and be available next time someone sends a message. Thus, if the sender of this message sends the same message again (or perhaps someone else tries to reregister 54 to 3030303030), the next time the `JUMPI` at position 7 would not process, and execution would STOP early at position 8.

Fortunately, you do not have to program in low-level assembly; a high-level language exists, especially designed for writing contracts, known as Solidity exists to make it much easier for you to write contracts (there are several others, too, including LLL, Serpent and Mutan, which you may find easier to learn or use depending on your experience). Any code you write in these languages gets compiled into EVM, and to create the contracts you send the transaction containing the EVM bytecode.

There are two types of transactions: a sending transaction and a contract creating transaction. A sending transaction is a standard transaction, containing a receiving address, an ether amount, a data bytearray and some other parameters, and a signature from the private key associated with the sender account. A contract creating transaction looks like a standard transaction, except the receiving address is blank. When a contract creating transaction makes its way into the blockchain, the data bytearray in the transaction is interpreted as EVM code, and the value returned by that EVM execution is taken to be the code of the new contract; hence, you can have a transaction do certain things during initialization. The address of the new contract is deterministically calculated based on the sending address and the number of times that the sending account has made a transaction before (this value, called the account nonce, is also kept for unrelated security reasons). Thus, the full code that you need to put onto the blockchain to produce the above name registry is as follows:

```
PUSH1 16 DUP PUSH1 12 PUSH1 0 CODECOPY PUSH1 0 RETURN STOP PUSH1 0 CALLDATALOAD SLOAD NOT
```

The key opcodes are CODECOPY, copying the 16 bytes of code starting from byte 12 into memory starting at index 0, and RETURN, returning memory bytes 0-16, ie. code byes 12-28 (feel free to "run" the execution manually on paper to verify that those parts of the code and memory actually get copied and returned). Code bytes 12-28 are, of course, the actual code as we saw above.

Virtual machine opcodes

A complete listing of the opcodes in the EVM can be found in the yellow paper. Note that high-level languages will often have their own wrappers for these opcodes, sometimes with very different interfaces.