

Ethereum Guide

Gav Wood

Published
with GitBook



Table of Contents

Introduction	0
Installation	1
CLI Tools	2
Getting started	2.1
Interactive Console	2.2
Mining	2.3
PoA Private Chains	2.4
ethkey	2.5
Mix	3
Project Editor	3.1
Scenarios Editor	3.2
State Viewer	3.3
Transaction Explorer	3.4
JavaScript console	3.5
Transaction debugger	3.6
Dapps deployment	3.7
Code Editor	3.8
Whisper	4
Recipes and How-tos	5
Cold Wallet Storage Device	5.1

TurboEthereum Guide

This book is intended as a practical user guide for the "Turbo" Ethereum software distribution, originally named after the language in which it is written, C++.

TurboEthereum is a large distribution of software including a number of diverse tools. This book begins with the installation instructions, before proceeding to introductions, walk-throughs and references for the various tools that make up TurboEthereum.

The full software suite of TurboEthereum includes:

- **AlethOne** (`alethone` , "A1") The mainline Ethereum desktop miner. It connects and syncs to the Ethereum network and lets you mine, and send transactions. It will also let you do pool mining.
- **AlethZero** (`alethzero` , "AZ") The power-user Ethereum client. It connects and syncs to the Ethereum network and lets you mine, make transactions, run DApps and inspect the blockchain. It has plugins to allow arbitrary extension.
- **++eth** (`eth`) The mainline CLI Ethereum client. Run it in the background and it will connect to the Ethereum network; you can mine, make transactions and inspect the blockchain.
- **Mix** (`mix`) The integrated development environment for DApp authoring. Quickly prototype and debug decentralised applications on the Ethereum platform.
- `ethkey` A key/wallet management tool for Ethereum keys. This lets you add, remove and change your keys as well as *cold wallet device*-friendly transaction inspection and signing.
- `ethminer` A standalone miner. This can be used to check how fast you can mine and will mine for you in concert with `eth` , `geth` and `pyethereum` .
- `ethvm` The Ethereum virtual machine emulator. You can use this to run EVM code.
- `solc` The Solidity compiler. You can use this to compile Solidity programs into assembly or machine code.
- `rlp` An serialisation/deserialisation tool for the Recursive Length Prefix format.

Installation

Installation is a different process dependent on which platform you run. At present, TurboEthereum supports three platforms: Ubuntu, Mac OS X and Windows.

For installing the desktop tools on Windows and Mac, just grab the [latest release](#). (For Windows you might also need [this](#).)

For installing on Ubuntu or Homebrew, instructions follow.

Installing on Ubuntu 14.04 and later (64-bit)

Warning: The `ethereum-qt` PPA will upgrade your system-wide Qt5 installation, from 5.2 on Trusty and 5.3 on Utopic, to 5.5.

For the latest stable version:

```
sudo add-apt-repository ppa:ethereum/ethereum-qt
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install cpp-ethereum
```

If you want to use the cutting edge developer version:

```
sudo add-apt-repository ppa:ethereum/ethereum-qt
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install cpp-ethereum
```

Installing the Mix IDE

Mix, the developer IDE is still in its infancy and still needs a lot of work. If you are adventurous, you can try to run Mix by installing the cutting edge developer version of cpp-ethereum (see above) and then add this:

```
sudo add-apt-repository ppa:ethereum/ethereum-qt
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install mix
mix
```

Installing on OS X and Homebrew

If you want the full suite of CLI tools, include `eth` and `ethminer`, you'll need [Homebrew](#).

Once you've got Homebrew installed, tap the ethereum brew:

```
brew tap ethereum/ethereum
```

Then, for the stable version:

```
brew install cpp-ethereum
brew linkapps cpp-ethereum
```

or, for the latest cutting edge developer version:

```
brew reinstall cpp-ethereum --devel
brew linkapps cpp-ethereum
```

Add the `--with-gui` option to include the AlethZero and the Mix IDE in the installation; you can then find `AlethZero` and `Mix` in your Applications folder.

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

CLI Tools

eth: The Command-Line Interface Client

Note: This chapter is for getting started with the command-line client `eth`. If you are interested only in AlethZero, move on to the next chapter.

`eth` or **`++eth`** as it is sometimes referred to, is the TurboEthereum CLI client. To use it, you should open a terminal on your system. `eth` normally just runs in the background. If you want an interactive console (you do!), it has an option: `console` (it has two others - `import` and `export`). We'll get to those later. So now, start `eth`:

```
eth console
```

You'll see a little bit of information:

```
(++)Ethereum
Beware. You're entering the Frontier!
16:13:52| Id: ##59650f8a...
16:13:58| Opened blockchain DB. Latest: #d4e56740... (rebuild not needed)
16:13:58| Opened state DB.
```

Using the Testnet

There are two Ethereum "networks": the *mainnet* (the current version of which is called "Frontier") and the *testnet* (currently called "Morden"). They're independent of each other. The only difference between the two is that ether is essentially gratis on the testnet. By default, you'll connect to the mainnet. If you want to connect to the Morden testnet instead, start `eth` with the `--testnet` option instead:

```
eth console --testnet
```

Setting your Master Password

After this information, the first thing it will do is ask you for a master password.

```
Please enter a MASTER password to protect your key store (make it strong!):
```

The master password is a password that protects your privacy and acts as a default security measure for your various Ethereum identities. Even with access to your computer nobody can work out who your online Ethereum addresses are without this password. It's also a default security password for your other keys, if you don't want to be remembering too many password. Anyway, it's the first line of defence and ought to be strong.

Enter a password, preferably taking into account [sage advice on password creation](#). Then when it asks for a confirmation...

```
Please confirm the password by entering it again:
```

...enter it again.

It will pause shortly while it figures out your network environment and starts it all up. After a little while, you'll see some information on the software as well as on the account it created for you.

Your First Account

This is your newly created default account (or 'identity'. I use the words interchangeably). In my case, it was the account that begins with `XE712F44`. This is an *ICAP code*, similar to an IBAN code that you might have used when doing banking transfers. You and only you have the special *secret key* for this account. It's guarded by the password you just typed. Don't ever tell anyone your password or they'll be able to send ether from this account and use it for nefarious means.

```
Transaction Signer: XE712F44Q0ZBKULD20DLAEE802YJ7XRG4 (be5af9b0-9917-b9bc-8f95-65cb9f042
Mining Beneficiary: XE712F44Q0ZBKULD20DLAEE802YJ7XRG4 (be5af9b0-9917-b9bc-8f95-65cb9f042
```

`eth` is nice. It tells you that any transactions you do will come from your account beginning with `XE712F44`. Similarly by default, if you mine successfully with the inbuilt miner, the proceeds will go into the same account.

You'll notice that there are two other codes in parentheses. The first is the *UUID* of the key. This is a code, only used on your computer, which allows us to identify which file the key is stored in without giving any any information of what account the key is for. In this case, the UUID begins with `be5af9b0`.

The second piece of information that is parenthesised is the first few digits of the hex key. Older clients and Ethereum software depend on this to identify accounts. We don't use it any more because it's longer and doesn't have any way of determining if an address is invalid,

so errors with mistyping can easily have major consequences.

Let's check that you do indeed have the key file for this account!

Find that Key!

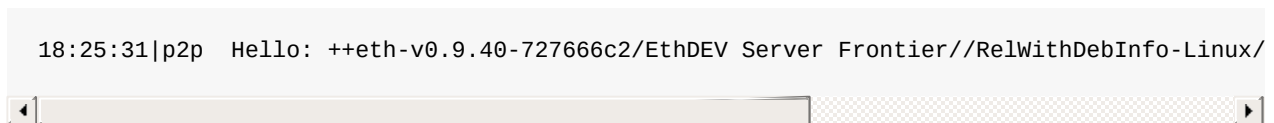
If you're using Linux or MacOS, open another terminal and navigate to `~/ .web3/keys` . This is where all of your keys are stored. Enter `ls` and make sure there's a file that corresponds to the account's UUID.

For Windows users, just use Explorer to navigate into your home folder's AppData/Web3/keys directory (you might need to enable Show Hidden Files to get there).

If you don't find a file with the same name as the UUID, then something is terribly wrong (out of disk space, possibly)! Get yourself on the forums and ask before going any further.

Syncing up

You'll now start seeing a little bit of information as it tries to connect to the network. You might see a line like:

A terminal window screenshot showing a log message: `18:25:31|p2p Hello: ++eth-v0.9.40-727666c2/EthDEV Server Frontier//RelWithDebInfo-Linux/`. The terminal has a light blue background and a scrollbar on the right.

This is it telling you that it's managed to contact another node. After a little while it will begin to synchronise to the network. This will probably give you an awful lot of messages. If there are too many for you to handle, reduce them by changing the verbosity. We can set the verbosity to zero (the lowest and quietest) by typing:

```
web3.admin.setVerbosity(0)
```

It'll reply `true` to tell you that all is fine:

```
> web3.admin.setVerbosity(0)
true
```

As it synchronises, the latest block number will constantly rise, usually rather fast. Once it is synchronised, it'll still rise but much more slowly - at around 1 block every 15 seconds.

You can check its progress by using the console to get the latest block number. To do this, type:

```
web3.eth.blockNumber
```

You'll end up with something like:

```
> web3.eth.blockNumber
11254
```

Got ETH?

You can easily check to see if you have ether in your account using the `eth.getBalance` function of `web3`. For this to work you'll need the address of which to get the balance. In my case, the address is the aforementioned `XE712F44Q0ZBKND20DLAEE802YJ7XRG4`:

```
> web3.eth.getBalance("XE712F44Q0ZBKND20DLAEE802YJ7XRG4")
0
```

That's not much, but then it is after all a newly created account. Let's query the balance of an account that actually has some funds, the Ethereum Foundation wallet:

```
> web3.eth.getBalance("XE86PXQKKKORDZQ1RWT9LGUGYZ1U57A56Y2")
11901464239480000000000000000
```

Ooh, rather a lot more. The answer is given in Wei, the lowest denomination of ether. To work out what this is in sensible terms, use `web3.fromWei` and provide a sensible unit, e.g. `grand` (a grand, for those unfamiliar with English slang, is one thousand Ether):

```
> web3.fromWei(web3.eth.getBalance("de0b295669a9fd93d5f28d9ec85e40f4cb697bae"), 'grand')
11901.46423948
```

Wow that's nearly 12 million ether.

And Finally...

Aside from the full power of Javascript, there are loads of functions you can use in the console; to see them just type `web3`.

When you're done playing, simply type `web3.admin.exit()` to exit `eth`.

eth Interactive Console

In order to interact with the client you have two options. If you have no client running, you can start one and provide the `-i` argument, which will start a client with the interactive console. Alternatively, if you already have a client running, and you started it with the `-j` flag, you can just run `ethconsole`, which will give you exactly the same environment without running a second client.

The interactive console is a javascript console which contains a subset of the [JSON-RPC api](#) plus some administration functions. To see all available functions type `web3` in the console prompt and press enter. For only the administrative functions type `web3.admin`.

- [Network connectivity](#)
- [Mining](#)
- [Miscellaneous administration](#)

Network connectivity

Querying network information

To figure out if you have any peers and if you are properly connected to the network you can type `web3.net`. This will conveniently provide something like:

```
> web3.net
{
  listening: true,
  getListening: [Function],
  peerCount: 2,
  getPeerCount: [Function]
}
```

To query any individual value you can just call it. For example:

```
> web3.net.peerCount
2
```

Interacting with the network

If you would like to interact with the network you can use the network admin functions. You can query them with `web3.admin.net`.

```
> web3.admin.net
{
  start: [Function],
  stop: [Function],
  connect: [Function],
  peers: [Function],
  nodeInfo: [Function]
}
```

Starting the network

If the client is not connected to the network you can start listening with

```
web3.admin.net.start()
```

```
> web3.net.listening
false
> web3.admin.net.start()
  < 12:52:24|p2p Worker stopping 17126 ms
  < 12:52:24|ethsync Worker stopping 17146 ms
  i 12:52:24|p2p UPnP device not found.
> true
> web3.net.listening
true
```

Stopping the network

In the same spirit you can stop listening with `web3.admin.net.stop()`

```
> web3.net.listening
true
> web3.admin.net.stop()
true
> web3.net.listening
false
>
```

Getting a list of peers

If you would like to obtain a list with information on the peers you are connected to you can

use `web3.admin.net.peers()` .

```
> web3.admin.net.peers()
[
  {
    caps: {
      eth: 61
    },
    clientVersion: 'Geth/v1.0.2-4591ae56/linux/go1.4.2',
    host: '52.16.188.185',
    id: 'a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69ad0dce72a4d8db',
    lastPing: 41,
    notes: {
      ask: 'nothing',
      manners: 'nice',
      sync: 'ongoing'
    },
    port: 0
  },
  {
    caps: {
      eth: 61
    },
    clientVersion: '++eth-v0.9.40-a1e4483e/Gav's Node//RelWithDebInfo-Linux/g++/int',
    host: '92.51.165.126',
    id: '5374c1bfff8df923d3706357eeb4983cd29a63be40a269aaa2296ee5f3b2119a8978c0ed68b8f6fc84a',
    lastPing: 18,
    notes: {
      ask: 'nothing',
      manners: 'nice',
      sync: 'holding'
    },
    port: 30300
  }
]
```

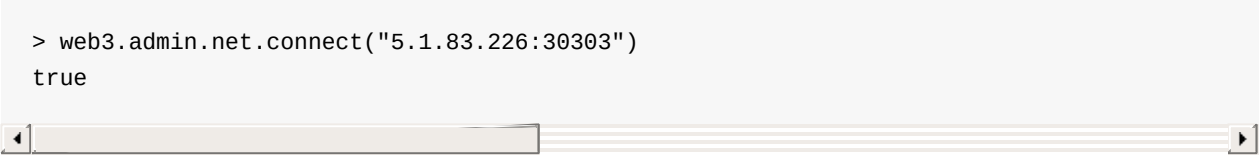
Getting your node information

To obtain your node ID along with other information about your node's address in the network then use `web3.admin.net.nodeInfo()`

```
> web3.admin.net.nodeInfo()
{
  address: '209.131.41.48',
  enode: 'enode://5bf4613faca50a0ff181915b2d8e5f0a87c82ed5a57dabc9812937bdacb167cf1420652930143a743d',
  id: '5bf4613faca50a0ff181915b2d8e5f0a87c82ed5a57dabc9812937bdacb167cf1420652930143a743d',
  listenAddr: '209.131.41.48:30303',
  name: '++eth-v0.9.41-cb61d09d/Lafteris\' node//RelWithDebInfo-Linux/g++/int',
  port: 30303
}
```

Connecting to other nodes

Sometimes, peer discovery may not work properly, or you may want to connect to a particular node in the network. In those cases you can use the `web3.admin.net.connect()` function to manually connect to a peer.

A screenshot of a terminal window with a light gray background. It shows a command prompt where the user has entered the command `> web3.admin.net.connect("5.1.83.226:30303")` and the terminal has responded with `true`. Below the text is a horizontal scrollbar.

```
> web3.admin.net.connect("5.1.83.226:30303")
true
```

If the above was successful we can see our new peer in the list:

```
> web3.admin.net.peers()
[
  {
    caps: {
      eth: 61
    },
    clientVersion: '++eth-v0.9.40-7faadaf4/EthDEV Frontier//RelWithDebInfo-Linux/g++/JIT',
    host: '5.1.83.226',
    id: '979b7fa28feeb35a4741660a16076f1943202cb72b6af70d327f053e248bab9ba81760f39d0701ef1d',
    lastPing: 31,
    notes: {
      ask: 'nothing',
      manners: 'nice',
      sync: 'holding'
    },
    port: 30303
  },
  {
    caps: {
      eth: 61
    },
    clientVersion: 'Geth/v1.0.2-4591ae56/linux/go1.4.2',
    host: '52.16.188.185',
    id: 'a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69ad0dce72a4d8db',
    lastPing: 41,
    notes: {
      ask: 'nothing',
      manners: 'nice',
      sync: 'ongoing'
    },
    port: 0
  },
  {
    caps: {
      eth: 61
    },
    clientVersion: '++eth-v0.9.40-a1e4483e/Gav's Node//RelWithDebInfo-Linux/g++/int',
    host: '92.51.165.126',
    id: '5374c1bfff8df923d3706357eeb4983cd29a63be40a269aaa2296ee5f3b2119a8978c0ed68b8f6fc84a',
    lastPing: 17,
    notes: {
      ask: 'nothing',
      manners: 'nice',
      sync: 'ongoing'
    },
    port: 30300
  }
]
```

Mining

You can also start and stop mining using the interactive console. To start mining use

```
web3.admin.eth.setMining(true)
```

```
> web3.admin.eth.setMining(true)
  i 13:48:01|miner0 Loading full DAG of seedhash: #b903bd76...
> web3.admin.eth.setMining(true)
DAG 13:48:01|miner0 Generating DAG file. Progress: 0 %
DAG 13:48:04|miner0 Generating DAG file. Progress: 1 %
DAG 13:48:07|miner0 Generating DAG file. Progress: 2 %
< 13:48:08|eth Stop worker 249 ms
< 13:48:09|eth Stop worker 479 ms
< 13:48:09|eth pause 480 ms
< 13:48:09|eth Stop worker 480 ms
< 13:48:09|eth pause 480 ms
```

Then again to stop mining simply invoke `web3.admin.eth.setMining(false)`

Miscellaneous administration

Exiting the client

You can exit the client with a `ctrl-c` signal but you can also use `web3.admin.exit()`

```
> web3.admin.exit()
true
< 13:36:50|eth Stop worker 510 ms
  i 13:36:50|eth Closing blockchain DB
  i 13:36:50|eth Closing state DB
< 13:36:50|ethsync Worker stopping 582 ms
< 13:36:50|p2p Worker stopping 581 ms
lifteris@archlenovo ~/ew/cpp-ethereum$
```

Changing the log verbosity

If you would like to see more log messages you can change the log verbosity by

```
web3.admin.setVerbosity()
```

 . This function takes a numeric argument from 0 to 99.

[illegible]

For a healthy logging level use the value of 1.

Mining on Ethereum

Mining is a common term for securing the Ethereum network and validating new transactions in exchange for a small payment. Anyone can mine, though it really helps if you can a good GPU. How often you are paid out depends on who else is mining and how much mining power (read: computation power) your hardware has.

We use a custom-made algorithm named Ethash, a combination of the Hashimoto and Dagger algorithms, designed by Tim Hughes, Vitalik Buterin and Matthew Wampler-Doty. It is memory-bandwidth-hard making is an excellent candidate for GPU mining but a bad candidate for custom hardware. We plan on switching to a proof-of-stake algorithm inover the course of the next 9 months with the Serenity release of Ethereum.

Because the algorithm is memory hard, you'll need 2GB of RAM per GPU with which you wish to mine, at least for the foreseeable future. (The dataset starts at 1GB and grows every few days, so you might be able to get away with 1.5GB for the first few months, if such graphics cards exist.)

ASICs and FPGAs is be strongly discouraged by being rendered financially inefficient, which was confirmed in an independent audit. Don't expect to see them on the market, and if you do, proceed with extreme caution.

Setting things up on Linux

For this quick guide, you'll need Ubuntu 14.04 or 15.04 and the fglrx graphics drivers. You an use NVidia drivers and other platforms, too, but you'll have to find your own way to getting a working OpenCL install with them.

If you're on 15.04, Go to "Software and Updates > Additional Drivers" and set it to "Using video drivers for the AMD graphics accelerator from fglrx". Once the drivers are installed and in use, you're all set, go to the next section!

If you're on 14.04, go to "Software and Updates > Additional Drivers" and set it to "Using video drivers for the AMD graphics accelerator from fglrx". Unfortunately, for some of you this will not work due to a known bug in Ubuntu 14.04.02 preventing you from switching to the proprietary graphics drivers required to GPU mine.

So, if you encounter this bug, and before you do anything else, go to "Software and updates > Updates" and select "Pre-released updates trusty proposed". Then, go back to "Software and Updates > Additional Drivers" and set it to "Using video drivers for the AMD graphics

accelerator from fglrx"). Reboot.

Once rebooted, it's well worth having a check that the drivers have now indeed been installed correctly.

Whatever you do, if you are on 14.04.02 do not alter the drivers or the drivers configuration once set. For example, the usage of `aticonfig --initial` can and likely will 'break' your setup. If you accidentally alter their configuration, you'll need to de-install the drivers, reboot, reinstall the drivers and reboot.

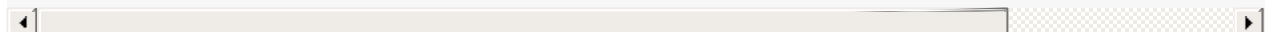
Mining with eth

Mining on Ethereum with `eth` is simple. If you need to mine with a single GPU then, just running `eth` will be sufficient. If not you can use a combination of `eth` and `ethminer`. This works on all platforms, though Linux is usually the easiest to set up.

Mining on a single GPU

In order to mine on a single GPU all that needs to be done is to run `eth` with the following arguments:

```
eth -i -v 1 -a 0xcadb3223d4eebcaa7b40ec5722967ced01cfc8f2 --client-name "OPTIONALNAMEHERE"
```



- `-i` Requests an interactive javascript console so that we can interact with the client
- `-v 1` Set verbosity to 1. Let's not get spammed by messages.
- `-a YOURWALLETADDRESS` Set the coinbase, where the mining rewards will go to. The above address is just an example. This argument is really important, make sure to not make a mistake in your wallet address or you will receive no ether payout.
- `--client-name "OPTIONAL"` Set an optional client name to identify you on the network
- `-x 50` Request a high amount of peers. Helps with finding peers in the beginning.
- `-m on` Actually launch with mining on.
- `-G` set GPU mining on.

While the client is running you can interact with it using the [interactive console](#).

Mining on multiple GPUs

Mining with multiple GPUs and `eth` is very similar to mining with [geth and multiple GPUs](#).

1. Ensure that an eth++ node is running with your coinbase address properly set:

```
eth -i -v 1 -a 0xcadb3223d4eebcaa7b40ec5722967ced01cfc8f2 --client-name "OPTIONALNAME"
```

Notice that we also added the `-j` argument so that the client can have the JSON-RPC server enabled to communicate with the ethminer instances. Additionally we removed the mining related arguments since `ethminer` will now do the mining for us.

2. For each of your GPUs execute a different ethminer instance:

```
ethminer --no-precompute -G --opencl-device XX
```

Where `xx` is an index number corresponding to the openCL device you want the ethminer to use.

In order to easily get a list of OpenCL devices you can execute `ethminer --list-devices` which will provide a list of all devices OpenCL can detect, with also some additional information per device. Below is a sample output:

```
[0] GeForce GTX 770
    CL_DEVICE_TYPE: GPU
    CL_DEVICE_GLOBAL_MEM_SIZE: 4286345216
    CL_DEVICE_MAX_MEM_ALLOC_SIZE: 1071586304
    CL_DEVICE_MAX_WORK_GROUP_SIZE: 1024
```

Finally the `--no-precompute` argument requests that the ethminers don't create the [DAG](#) of the next epoch ahead of time.

Benchmarking

Mining power tends to scale with memory bandwidth. Our implementation is written in OpenCL, which is typically supported better by AMD GPUs over NVidia. Empirical evidence confirms that AMD GPUs offer a better mining performance in terms of price than their NVidia counterparts. R9 290x appears to be the best card at present.

To benchmark a single-device setup you can use `ethminer` in benchmarking mode through the `-M` option:

```
ethminer -G -M
```

If you have many devices and you'll like to benchmark each individually, you can use the `--opengl-device` option similarly to the previous section:

```
ethminer -G -M --opengl-device XX
```

Use `ethminer --list-devices` to list possible numbers to substitute for the `XX`.

PoA Private Chains

NOTE: This chapter is work in progress.

TurboEthereum supports Proof-of-Authority (PoA) private chains through the Fluidity core ethereum client `flu`. Proof-of-authority chains utilise a number of secret keys (authorities) to collaborate and create the longest chain instead of the public Ethereum network's proof-of-work scheme (Ethash).

`flu` is configured through a JSON-format file which specifies all of the various settings concerning the blockchain. Here is an example JSON file:

```
{
  "sealEngine": "BasicAuthority",
  "options": {
    "authorities": [
      "0xfa0c706a1410c8785baa7498325cf7461b325583",
      "0x7766d151b2c63cb096f624daa091ccb27a2c693f"
    ]
  },
  "params": {
    "accountStartNonce": "0x",
    "maximumExtraDataSize": "0x1000000",
    "blockReward": "0x",
    "registrar": "",
    "networkID" : "0x45"
  },
  "genesis": {
    "author": "0x0000000000000000000000000000000000000000",
    "timestamp": "0x00",
    "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData": "0x",
    "gasLimit": "0x1000000000000"
  },
  "accounts": {
    "0000000000000000000000000000000000000000000000000000000000000001": { "wei": "1", "precompiled": { "name"
    "0000000000000000000000000000000000000000000000000000000000000002": { "wei": "1", "precompiled": { "name"
    "0000000000000000000000000000000000000000000000000000000000000003": { "wei": "1", "precompiled": { "name"
    "0000000000000000000000000000000000000000000000000000000000000004": { "wei": "1", "precompiled": { "name"
  },
  "network": {
    "nodes": [
      "enode://f12709ce6a10fdc76cea6129c6e85e44225d4539ac1e5b26d2cb73f436b9f34c2a1a
      "enode://9bd11ae2cdbdd670dcbd34fa04ff71d840a9fb658d166f4d58192ec8f3d23c07cda4
    ]
  }
}
```

Breaking it down:

```
"sealEngine": "BasicAuthority",
```

This states that the seal engine of the chain which we wish to use is the `BasicAuthority` seal engine; this creates a proof-of-authority chain (rather than the Ethash proof-of-work chain on the public network).

```
"options": {
  "authorities": [
    "0xfa0c706a1410c8785baa7498325cf7461b325583",
    "0x7766d151b2c63cb096f624daa091ccb27a2c693f"
  ]
},
```

This provides a number of options for our `BasicAuthority` seal engine. In this case, we provide the Ethereum addresses of the two accounts which are authorised to sign a new block.

Make sure you own one of these or you'll find it very difficult to append new blocks.

```
"params": {
  "accountStartNonce": "0x",
  "maximumExtraDataSize": "0x1000000",
  "blockReward": "0x",
  "registrar": "",
  "networkID" : "0x45"
},
"genesis": {
  "author": "0x0000000000000000000000000000000000000000",
  "timestamp": "0x00",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "0x",
  "gasLimit": "0x100000000000000"
},
```

This provides additional parameters for the chain's consensus logic. (The difference between parameters and options is too subtle for this guide.) In this case, we're setting allowing a generous gas limit and extra-data field size and otherwise making everything zero. This is fine for most private chains.

```
"accounts": {
  "0000000000000000000000000000000000000000000000000000000000000001": { "wei": "1", "precompiled": { "name"
  "0000000000000000000000000000000000000000000000000000000000000002": { "wei": "1", "precompiled": { "name"
  "0000000000000000000000000000000000000000000000000000000000000003": { "wei": "1", "precompiled": { "name"
  "0000000000000000000000000000000000000000000000000000000000000004": { "wei": "1", "precompiled": { "name"
},
```

This defines any accounts that are pre-specified in the genesis block. You can specify the balance, nonce, code and storage of each of the accounts. You can also use the powerful precompiled contract system to place precompiled contracts in each of the accounts together with gas-cost rules.

In this case we're placing a token value in each of the pre-compiled contracts accounts, standard practice to avoid the first transactor getting stung for using them. We're also setting them up similarly to the public network, with each of the four algorithms in the first four slots together with the gas costs from the public network.

Further precompiled contract algorithms can be added through creating and linking a library using the macro `ETH_REGISTER_PRECOMPILED`. For example, to create an algorithm which placed a simple byte-wise XOR checksum of the input into the output:

```
ETH_REGISTER_PRECOMPILED(xorchecksum)(bytesConstRef _in, bytesRef _out)
{
    // No point doing any computation if there's nowhere to place the result.
    if (_out.size() >= 1)
    {
        // XOR every byte in the input together into the accumulator acc.
        byte acc = 0;
        for (unsigned i = 0; i < _in.size(); ++i)
            acc ^= _in[i];
        // Place the result into the output.
        _out[0] = acc;
    }
}
```

With this defined, you would be at liberty to name a contract precompiled with:

```
"precompiled": {
    "name": "xorchecksum",
    "linear": { "base": 1, "word": 1 }
}
```

You can select the costs (1 gas plus 1 gas for each 32-byte word in the input) as you choose.

```
"network": {
    "nodes": [
        "enode://f12709ce6a10fdc76cea6129c6e85e44225d4539ac1e5b26d2cb73f436b9f34c2a1a623e"
        "enode://9bd11ae2cdbdd670dcbdb34fa04ff71d840a9fb658d166f4d58192ec8f3d23c07cda490e7"
    ]
}
```

Finally we set up the network. In this case we define which node IDs are allowed to connect (and be connected to) and give some IPs/ports for them. Note even if the IPs/ports are not known, it is important that all node IDs on the private network are listed. Listing IDs with an invalid IP/port, as in the second entry, is fine.

Setting up a PoA Private Network

To set up your PoA private network, first determine the network ID of each node which will be sitting on it. Author a JSON file similar to that above but with the `network.nodes` array populated with those IDs (with, wherever possible, the right IPs/ports as this will make bootstrapping easier).

Finding the node ID is easy with `flu` : when it is run simply look out for a line beginning:

```
Node ID: enode://...
```

Determine, of those nodes, which will be able to sign new blocks. For each of them make sure you have at least one of the keys listed in your JSON file's `options.authorities` section.

Running the `flu` client on a fresh machine (or with a fresh `--path` directory) will result in a keystore with a single, freshly generated, account key.

You can find out what it is by looking for a line beginning:

```
Created key: 0x...
```

The client (perhaps on later invocations) can use this account to sign blocks if it is included in the `options.authorities` list.

You may author the rest of the JSON file as you choose, perhaps altering the `extraData` genesis field to customise the chain and avoid it being confused with other, older or newer chains.

When ready, you can start one or more nodes with this file (let's assume it is called `myconfig.json` and that you've copied it into each machine's `~` path:

```
$ flu console --config ~/myconfig.json
```

The `console` specifies that we want to have a Javascript console for interacting with our node (omit it for non-interactive mode) and the `--config` specifies our JSON configuration file.

To begin sealing, use `web3.admin.eth.setMining(true)` at the console. All of the rest of the JS API is available for you.

There are a number of options to help you configure things to your particular situation:

- `--path` Alter the path of the blockchain/state databases and the keys/secrets database. Defaults to `~/.fluidity`.
- `--master` Specify the password which is used to encrypt/decrypt the keys database and the default secret.
- `--client-name` Specify this particular node's name.
- `--public-ip` Specify the public IP of this node (i.e. the one that we advertise).
- `--listen-ip` Specify the IP that we are actually listening on (e.g. the local adaptor IP).
- `--listen-port` Specify the port which we should listen on.
- `--start-sealing` Start sealing blocks immediately.
- `--jsonrpc` Enable JSON-RPC server on port 8545.
- `--jsonrpc-port` Enable JSON-RPC server on the given port.
- `--jsonrpc-cors-domain` Configure the JSON-RPC server's CORS domain.

By enabling the JSON-RPC server you can use e.g. `ethconsole` to connect to and interact with the private chain.

ethkey

`ethkey` is a CLI tool that allows you to interact with the Ethereum wallet. With it you can list, inspect, create, delete and modify keys and inspect, create and sign transactions.

Keys and Wallets

When using Ethereum you will own one or more `keys`. These are special files that allow you access to a particular account. Such access might allow you to spend funds, register a name or transfer an asset. Keys are standardised and compatible across major clients. They are always protected by password-based encryption. Also they do not directly identify the actual account that the key represents. To determine this, the key must be decrypted through providing the correct password.

In TurboEthereum, your *wallet* keeps a track of each key that you own along with what *address* it represents. An address is just way of referring to a particular *account* in Ethereum. It, too, is protected by a password, which is generally provided when the client begins.

While all clients have keys, some do not have wallets; these clients typically store the address in the key in plain view. This substantially reduces privacy.

Creating a Wallet

We'll assume you have not yet run a client such as `eth` or anything in the Aleth series of clients. If you have, you should skip this section.

To create a wallet, run `ethkey` with the `createwallet` command:

```
> ethkey createwallet
Please enter a MASTER passphrase to protect your key store (make it strong!):
```

You'll be asked for a "master" passphrase. This protects your privacy and acts as a default password for any keys. You'll need to confirm it by entering the same text again.

Listing the Keys in your Wallet

We can list the keys within the wallet simply by using the `list` command:

```
> ethkey list  
No keys found.
```

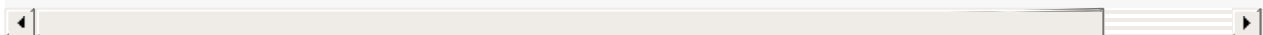
We haven't yet created any keys, and it's telling us so! Let's create one.

Creating your First Key

One of the nice things about Ethereum is that creating a key is tantamount to creating an account. You don't need to tell anybody else you're doing it, you don't even need to be connected to the Internet. Of course your new account will not contain any Ether. But it'll be yours and you can be certain that without your key and your password, nobody else can ever access it.

To create a key, we use the `new` command. To use it we must pass a name - this is the name we'll give to this account in the wallet. Let's call it "test":

```
> ethkey new test  
Enter a passphrase with which to secure this account (or nothing to use the master passp
```



It will prompt you to enter a passphrase to protect this key. If you just press enter, it'll use the default "master" passphrase. Typically this means you won't need to enter the passphrase for the key when you want to use the account (since it remembers the master passphrase). In general, you should try to use a different passphrase for each key since it prevents one compromised passphrase from giving access to other accounts. However, out of convenience you might decide that for low-security accounts to use the same passphrase.

Here, let's give it the incredibly imaginative passphrase of `123`.

Once you enter a passphrase, it'll ask you to confirm it by entering again. Enter `123` a second time.

Because you gave it its own passphrase, it'll also ask you to provide a hint for this password which will be displayed to you whenever it asks you to enter it. The hint is stored in the wallet and is itself protected by the master passphrase. Enter the truly awful hint of `321 backwards`.

```
> ethkey new test
Enter a passphrase with which to secure this account (or nothing to use the master passph
Please confirm the passphrase by entering it again:
Enter a hint to help you remember this passphrase: 321 backwards
Created key 055dde03-47ff-dded-8950-0fe39b1fa101
  Name: test
  Password hint: 321 backwards
  ICAP: XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ
  Raw hex: 0092e965928626f8880629cec353d3fd7ca5974f
```

Notice the last two lines there. One is the ICAP address, the other is the raw hexadecimal address. The latter is an older representation of address that you'll sometimes see and is being phased out in favour of the shorter ICAP address which also includes a checksum to avoid problems with mistyping. All normal (aka *direct*) ICAP addresses begin with `XE` so you should be able to recognise them easily.

Notice also that the key has another identifier after `Created key`. This is known as the UUID. This is a unique identifier for the key that has absolutely nothing to do with the account itself. Knowing it does nothing to help an attacker discover who you are on the network. It also happens to be the filename for the key, which you can find in either `~/ .web3/keys` (Mac or Linux) or `$HOME/AppData/Web3/keys` (Windows).

Now let's make sure it worked properly by listing the keys in the wallet:

```
> ethkey list
055dde03-47ff-dded-8950-0fe39b1fa101 0092e965... XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ test
```

It reports one key on each line (for a total of one key here). In this case our key is stored in a file `055dde...` and has an ICAP address beginning `XE472EVK...`. Not especially easy things to remember so rather helpful that it has its proper name, `test`, too.

ICAP or Raw hex?

You might see addresses passed as hex-only strings, especially with old software. These are dangerous since they don't include a checksum or special code to detect typos. You should generally try to keep clear of them.

Occasionally, however, it's important to convert between the two. `ethkey` provides the `inspect` command for this purpose. When passed any address, file or UUID, it will tell you information about it including both formats of address.

For example, to get it to tell us about our account, we might use:

```
> ethkey inspect test
test (0092e965...)
  ICAP: XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ
  Raw hex: 0092e965928626f8880629cec353d3fd7ca5974f
```

We could just as easily use the ICAP `XE472EVK...` or raw hex `0092e965...` :

```
> ethkey inspect XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ
test (0092e965...)
  ICAP: XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ
  Raw hex: 0092e965928626f8880629cec353d3fd7ca5974f
> ethkey inspect 0092e965928626f8880629cec353d3fd7ca5974f
test (0092e965...)
  ICAP: XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ
  Raw hex: 0092e965928626f8880629cec353d3fd7ca5974f
```

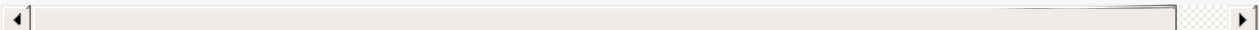
Backing up Your Keys

You should always back up your keys! Any backup solution that protects your home directory should also protect your keys (since that's where they live). However for added piece of mind make an explicit backup of your keys by copying the contents of the `~/ .web3/keys` (Mac or Linux, or `$HOME/AppData/Web3/keys` for Windows) to an external disk. You might also open the files in a text editor, print them and keep them in a lawyer's safe for additional piece of mind. **If they get lost, nobody can help you!**

Decoding a Transaction

Here's an unsigned transaction. It authorises the donation of 1 ether to me:

```
ec80850ba43b74008252089400be78bf8a425471eca0cf1d255118bc080abf95880de0b6b3a7640000801b808
```



On its own, it won't do much. We can see this by decoding it in `ethkey` :

```
> ethkey decode ec80850ba43b74008252089400be78bf8a425471eca0cf1d255118bc080abf95880de0b6b
Transaction 705d490edc318b50223efa7bb9c19d65f05c3c527e4f8e60535b46a2ed128706
type: message
to: XE6934MX3U67M48MPHYMC1A1X306AFKEXH (00be78bf...)
data: none
from: <unsigned>
value: 1 ether (1000000000000000000 wei)
nonce: 0
gas: 21000
gas price: 50 Gwei (50000000000 wei)
signing hash: f2790ed53c803ee882c892e1d9715181dfc93780d755fbe4ffefd90701e15c31
```

Note that it states the transaction is `<unsigned>` to the right of `from:`. This means that at present it's useless. Signing it would make it useful (to me, at least, since it'd make me one Ether richer), or dangerous (to you if you didn't want to give me that Ether).

Signing a Transaction

`ethkey` can be used to sign a pre-existing, but unsigned, transaction (it can also create a transaction and sign it itself). In this case, the transaction is actually harmless anyway since we're signing with the key of a fresh account that has no Ether to be transferred.

The command we'll use is `sign`. To use it we must identify the account with which we wish to sign. This can be the ICAP (`XE472EVK...`), the hex address (`0092e965...`), the UUID (`055dde...`), the key file or simply the plain old name (`test`). Secondly you must describe transaction it should sign. This can be done through passing the hex or through a file containing the hex.

```
> ethkey sign test ec80850ba43b74008252089400be78bf8a425471eca0cf1d255118bc080abf95880de0
Enter passphrase for key (hint:321 backwards):
```

It will ask you for the passphrase from earlier, along with the ludicrously transparent hint. Enter `123`, the correct answer and it will provide you with the unsigned transaction (`a37c58...`), a `:` and the signed transaction (`f86c80...`):

```
a37c588c853dc20bbaef53b680e23642a03122897bbb9a53d25d0d8f3665a94f: f86c80850ba43b740082520
```

Let's make sure it worked by decoding it.


```
> ethkey decode f86c80850ba43b74008252089400be78bf8a425471eca0cf1d255118bc080abf95880de0b
Transaction a37c588c853dc20bbaef53b680e23642a03122897bbb9a53d25d0d8f3665a94f
  type: message
  to: XE6934MX3U67M48MPHYMC1A1X306AFKEXH (00be78bf...)
  data: none
  from: XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ (0092e965...)
  value: 1 ether (1000000000000000000 wei)
  nonce: 0
  gas: 21000
  gas price: 50 Gwei (50000000000 wei)
  signing hash: f2790ed53c803ee882c892e1d9715181dfc93780d755fbe4ffefd90701e15c31
  v: 1
  r: 7638c34170f3e04313bbb6c5bfc10a0c665200515a1aa5e922c7ae6c0dd085fa
  s: 79ab46048e643bb4042bcb22da86d2646eb0b727f23aa3e165102b824563c70d
```

Being a signed transaction, it has the three fields at the end (`v` , `r` and `s`) and, importantly, the address from whom the transaction is sent (`from:`). You'll notice that the sender address (`XE472EVK...`) is indeed ours from before!

The signed transaction can be sent in an e-mail in a similar way to how you might send a cheque in the mail. It can also be placed on the network to enact it; through the web3 API `web3.sendRawTransaction`

Killing an Account

Let's now delete our key we've made. Deleting a key actually actually deletes the underlying file. After doing this there's no going back (unless you have a backup). To avoid losing anything, we're first going to back up our account. First, let's copy the key file somewhere safe:

```
> mkdir ~/backup-keys
> cp ~/.web3/keys/* ~/backup-keys
```

or, for Windows:

```
> md $HOME/backup-keys
> copy $HOME/AppData/Web3/keys/*.* $HOME/backup-keys
```

Now, we'll delete the key with the `kill` command:

```
> ethkey kill test
1 key(s) deleted.
```

And bang! It's gone.

Check by calling `list` :

```
> ethkey list
No keys found.
```

Restoring an Account from a Backup

Now let's suppose we made a horrible mistake and want to recover the account. Luckily we made a backup!

We could simply copy it back into the original `keys` directory. This would indeed make the key "available", however it would only be identifiable by its UUID (the filename minus the `.json`). This is a bit of a pain.

Better would be to reimport it into the wallet, which makes it addressable by its ICAP and hex, and gives it a name and password hint to boot. To do this, we need to use the `import` command, which takes the file and the name of the key:

```
> ethkey import ~/backup-keys/* test
```

or, for Windows:

```
> ethkey import $HOME/backup-keys/*.test
```

Here it will need to know the passphrase for the key, mainly to determine the address of the key for placing into the wallet. There's no hint now because the wallet doesn't know anything about it. Enter the `123` passphrase.

It will then ask you to provide a hint (assuming it's different to the master password, which ours is). Enter the same hint.

```
Enter the passphrase for the key:
Enter a hint to help you remember the key's passphrase: 321 backwards
Imported key 055dde03-47ff-dded-8950-0fe39b1fa101
Name: test
Password hint: 321 backwards
ICAP: XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ
Raw hex: 0092e965928626f8880629cec353d3fd7ca5974f
```

Finally it will tell you that all went well and the key is reimported. We should recognise our address by now with the `XE472EVK...` .

To double-check, we can list the keys:

```
> ethkey list
055dde03-47ff-dded-8950-0fe39b1fa101 0092e965... XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ test
```

All restored!

Importing a key from another client (e.g. Geth)

Because our keys all share the same format it's really easy to import keys from other clients like Geth. In fact it's exactly the same process as restoring a key from a previous backup as we did in the last step.

If we assume we have a geth key at `mygethkey.json`, then to import it to use `eth`, simply use:

```
> ethkey import mygethkey.json "My Old Geth Key"
```

It will prompt you for your passphrase to ascertain the address for the key.

Changing the Password

Security people reckon that it is prudent to change your password regularly. You can do so easily with `ethkey` using the `recode` command (which actually does a whole lot more, but that's advanced usage).

To do so, simply pass in the name(s) of any keys whose passwords you wish to change. Let's change our key's password:

```
> ethkey recode test
Enter old passphrase for key 'test' (hint: 321 backwards):
```

So it begins by asking for your key's old passphrase. Enter in the correct answer `123`.

It will then ask you for the new password (enter `321`) followed by a confirmation (enter the same) and a password hint (`123 backwards`).

```
Enter new passphrase for key 'test':
Please confirm the passphrase by entering it again:
Enter a hint to help you remember this passphrase: 123 backwards
Re-encoded key 'test' successfully.
```

You'll finally get a confirmation that the re-encoding took place; your key is now encrypted by the new password.

The Rest

There's much more to discover with `ethkey` ; it provides a suite of commands for playing with "bare" secrets, those not in the wallet (the `listbare` , `newbare` , ... commands), with brain wallets (`newbrain` , `importbrain` , `inspect`), and allows keys to be imported without actually ever being decrypted (`importwithaddress`) and conversion between ICAP and hex (`inspectbare`).

Options allow you to alter transactions before you sign them and even create transactions from scratch. You can also configure the method by which keys are encrypted, changing the encryption function or its parameters.

See `ethkey --help` for more information. Enjoy!

Mix

The IDE Mix is intended to help you as a developer to create, debug and deploy contracts and dapps (both contracts backend and frontend).

Start by creating a new project that consists of

- contracts
- html files
- JavaScript files
- style files
- image files

Project Editor

You can use projects to manage the creation and testing of a dapp. The project will contain data related to both backend and frontend as well as the data related to your scenarios (blockchain interaction) for debugging and testing. The related files will be created and saved automatically in the project directory.

Creating a new project

The development of a dapp start with the creation of a new project. Create a new project in the “edit” menu. Enter the project name, e.g. "Ratings" and select a path for the project file.

Editing backend contract file

By default, a new project contains a contract “Contract” for backend development on the blockchain using the Solidity language and the “index.html” for the frontend. Check the Solidity tutorial for references.

Edit the empty default contract “Contract”, e.g.

```
contract Rating {  
  
    function setRating(bytes32 _key, uint256 _value) {  
  
        ratings[_key] = _value;  
  
    }  
    mapping (bytes32 => uint256) public ratings;  
}
```

Check the Solidity tutorial for help getting started with the solidity programming language.

Save changes

Editing frontend html files

Select default index.html file and enter the following code

```
<!doctype>
<html>
<head>
<script type="text/javascript">
function getRating() {
    var param = document.getElementById("query").value;
    var res = contracts["Rating"].contract.ratings(param);
    document.getElementById("queryres").innerText = res;
}

function setRating() {
    var key = document.getElementById("key").value;
    var value = parseInt(document.getElementById("value").value);
    var res = contracts["Rating"].contract.setRating(key, value);
}
</script>
</head>
<body bgcolor="#E6E6FA">
    <h1>Ratings</h1>
    <div>
        Store:
        <input type="string" id="key">
        <input type="number" id="value">
        <button onclick="setRating()">Save</button>
    </div>
    <div>
        Query:
        <input type="string" id="query" onkeyup='getRating()''>
        <div id="queryres"></div>
    </div>
</body>
</html>
```

Then it is possible to add many contract files as well as many HTML, JavaScript, css files

Scenarios Editor

Scenarios can be used to test and debug contracts.

A scenario is effectively a local blockchain where blocks can be mined without PoW – otherwise testing would be quite slow ;).

A scenario consists of a sequence of transactions. Usually, a scenario would start with the contract creation scenarios of the dapp. In addition, further transactions can be added to test and debug the dapp. Scenarios can be modified, i.e. transactions can be removed. Note that a scenario needs to be rebuilt for modifications to become effective. Further testing can be done using local JS calls via the JS API.

In case it's not open, access the scenario and debugger pane by pressing F7 or Windows > Show right or the debug button in the upper right corner of the main window.

Creating and setting up a new scenario

When you launch Mix for the first time, an empty scenario, i.e. not containing any transactions, will be created. Add an account named “MyAccount” and set it's initial balance to 1 ether. Click OK. Rename the scenario to “Deploy”.

Modifying initial ether balance of an account

Actually, we want to do a lot of tests Edit the Genesis block parameters and set your initial account balance to 1000 ether. Rebuild the scenario for the change to become effective.

Rebuilding a scenario

Each time a transaction is modified or an account added, the scenario has to be rebuilt for modifications to become effective. Note that if a scenario is rebuilt the web frontend (local storage) may also need to be reset (this is not done automatically by Mix).

Creating a transaction

Let's get some ether sent to Bob. Create another account named “Bob” with zero ether balance. Create a new transaction in the scenario pane. Click “Add Tx...” and send 300 ether to Bob. Add a block.

Altering and reusing scenarios

Create a new scenario or start from a scenario with several transactions that you duplicate first

Rename the scenario

Modify scenario by specifying transactions that shall be removed

Rebuild the scenario

Display calls

A contract call is a function invocation. This is not a transaction as a contract call cannot change the state. A contract call is not part of the blockchain but for practical and ux design reason, it is convenient to display calls at the same functional level as a transaction. The JS icon warn you that this is not a transaction but a call. To show/hide call, click on the menu Scenario -> Display calls.

State Viewer

This panel is located below the block chain panel, in the scenario view. Once the blockchain has been run, this panel shows the state of the blockchain.

By state we mean all accounts balance (including contract and normal account), and the storage (global variable of all deployed contract). The content of this panel is not static, it depends on the selected transaction on the blockchain panel. The state shown here is the state resulting of the execution of the selected transaction.

The screenshot shows the 'State Viewer' panel in a blockchain scenario view. The interface includes a top toolbar with buttons for 'New Scenario', 'Edit Title', 'Delete', 'New', 'Reset', 'Save', and 'Duplicate'. Below the toolbar are buttons for 'Rebuild Scenario', 'Add Tx...', 'Add Block', and 'New Account...'. The main area is divided into sections: 'STARTING PARAMETERS', 'BLOCK 1', 'BLOCK 2', 'BLOCK 3', and 'PENDING TRANSACTIONS'. Each block shows a transaction hash (e.g., '0x38f388fa... (user1)') and the corresponding function call (e.g., 'BasicContract.BasicContract()'). The 'BLOCK 3' transaction is highlighted in blue. Below the blocks is the 'Accounts' section, which lists the balances of several accounts, including 'user2', 'user1', 'testCtr', and 'user4'. The 'testCtr' account is highlighted in blue. The 'Storage' section shows the state of the 'testCtr' contract, including an array, a variable 'i', and a variable 's'.

STARTING PARAMETERS

BLOCK 1

0x38f388fa... (user1) → BasicContract.BasicContract()

BLOCK 2

0x38f388fa... (user1) → 0xf025d811... (BasicContract.testBytes4())

BLOCK 3

0x38f388fa... (user1) → testCtr.testCtr()

PENDING TRANSACTIONS

Accounts

0x06400992be45bc64a52b5c55d3df84596d6cb4a1 (user2) = 1 Mether
 0x38f388fadf4a6a35c61c3f88194ecSae162c8944 (user1) = 1 Mether
 0x78dfc5983baecf65f73e3de3a96cee24e6b7981e (testCtr) = 0 wei
 0xb0dcdc575ef06dc30aaa069d8043c9d463c931c (user4) = 1 Mether
 BasicContract - f025d81196b72fba60a1d4dddad12eeb8360d828

array = ["8","4","5","7"]
 i = "5"
 s = [""]

testCtr - 78dfc5983baecf65f73e3de3a96cee24e6b7981e

pp = "3"

In that case, 2 contracts are deployed, the selected transaction (deployment of testCtr) is the last one. so the state view shows the storage of both TestCtr and BasicContract.

Transaction Explorer

Using the transaction pane

The transaction pane enables you to explore transactions receipts, including

Input parameters Return parameters As well as Event logs To display the transaction explorer, click on the down triangle icon which is on the right of each transaction, this will expand transaction details:

The screenshot displays the Transaction Explorer interface. At the top, there is a 'STARTING PARAMETERS' section. Below it, 'BLOCK 1' is shown with a transaction from '0x38f388fa... (user1)' to 'BasicContract.BasicContract()'. 'BLOCK 2' is expanded, showing a transaction from '0x38f388fa... (user1)' to '0xf025d811... (BasicContract.testBytes4())'. The transaction details for BLOCK 2 include: 'From: 0x38f388fa... (user1)', 'To: 0xf025d811... (BasicContract.testBytes4())', 'Value: 0 wei', 'Input: 0x79616e6e00', and 'Output: undefined 3'. The 'Events' section lists: 'event1(10000)', 'event1(4)', 'event1(10001)', and 'event1(10002)'. At the bottom of the transaction details, there are buttons for 'Edit transaction...' and 'Debug transaction...'. 'BLOCK 3' is partially visible at the bottom, showing a transaction to 'testCtr.testCtr()'. A 'PENDING TRANSACTIONS' section is at the very bottom.

Then you can either copy the content of this transaction in the clipboard, Edit the current transaction (you will have to rerun the blockchain then), or debug the transaction.

JavaScript console

Mix exposes the following objects into the global window context

web3 - Ethereum JavaScript API

contracts: A collection of contract objects. A key to the collection is the contract name. A value is an object with the following properties:

contract: contract object instance (created as in web3.eth.contract)

address: contract address from the last deployed state (see below)

interface: contract ABI

Check the JavaScript API Reference for further information.

Using the JS console to add transactions and local calls

In case the name of the contract is "Sample" with a function named "set", it is possible to make a transaction to call "set" by writing:

```
contracts["Sample"].contract.set(14)
```

If a call can be made this will be done by writing:

```
contracts["Sample"].contract.get.call()
```

It is also possible to use all properties and functions of the web3 object:

<https://github.com/ethereum/wiki/wiki/JavaScript-API>

Transaction debugger

Mix supports both Solidity and assembly level contract code debugging. You can toggle between the two modes to retrieve the relevant information you need. At any execution point the following information is available:

VM stack – See Yellow Paper for VM instruction description

Call stack – Grows when contract is calling into another contract. Double click a stack frame to view the machine state in that frame

Storage – Storage data associated with the contract

Memory – Machine memory allocated up to this execution point

Call data – Transaction or call parameters

Accessing the debug mode

When transaction details are expanded, you can switch to the debugger view by clicking on the "Debug Transaction" button

Toggling between debug modes and stepping through transactions

This opens the Solidity debugging mode. Switch between Solidity and EVM debugging mode using the Menu button (Debug -> Show VM code)

- Step through a transaction in solidity debugging mode
- Step through a transaction in EVM debugging mode

Dapps deployment

This feature allows users to deploy the current project as a Dapp in the main blockchain. This will deploy contracts and register frontend resources.

The deployment process includes three steps:

- **Deploy contract:** This step will deploy contracts in the main blockchain.
- **Package dapp:** This step is used to package and upload frontend resources.
- **Register:** To render the Dapp, the Ethereum browser (Mist or AlethZero) needs to access this package. This step will register the URL where the resources are stored.

To Deploy your Dapp, Please follow these instructions:

Click on `Deploy` , `Deploy to Network` . This modal dialog displays three parts (see above):

- **Deploy contract**
- *Select Scenario*

"Ethereum node URL" is the location where a node is running, there must be a node running in order to initiate deployment.

"Pick Scenario to deploy" is a mandatory step. Mix will execute transactions that are in the selected scenario (all transactions except transactions that are not related to contract creation or contract call). Mix will display all the transactions in the panel below with all associated input parameters.

"Gas Used": depending on the selected scenario, Mix will display the total gas used.

- *Deploy Scenario*

"Deployment account" allow selecting the account that Mix will use to execute transactions.

"Gas Price" shows the default gas price of the network. You can also specify a different value.

"Deployment cost": depending on the value of the gas price that you want to use and the selected scenario. this will display the amount ether that the deployment need.

"Deployed Contract": before any deployment this part is empty. This will be filled once the deployment is finished by all contract addresses that have been created.

"Verifications". This will shows the number of verifications (number of blocks generated on top of the last block which contains the last deployed transactions). Mix keep track of all the transactions. If one is missing (unvalidated) it will be displayed in this panel.

- **Package dapp**
- *Generate local package*

The action "Generate Package" will create the package.dapp in the specified folder

"Local package Uri" the content of this field can be pasted directly in AlethZero in order to use the dapp before uploading it.

- *Upload and share package*

This step has to be done outside of Mix. package.dapp file has to be hosted by a server in order to be available by all users.

"Copy Base64" will copy the base64 value of the package to the clipboard.

"Host in pastebin.com" will open pastebin.com in a browser (you can then host your package as base64).

- **Package dapp**

"Root Registrar address" is the account address of the root registrar contract

"Http URL" is the url where resources are hosted (pastebin.com or similar)

"Ethereum URL" is the url that users will use in AlethZero or Mist to access your dapp.

"Formatted Ethereum URL" is the url that users will use in AlethZero or Mist to access your dapp.

"Gas Price" shows the default gas price of the network. You can also specify a different value.

"Registration Cost" will display the amount of ether you need to register your dapp url.

Code Editor

This editor provides basic functionalities of a code editor.

- In Solidity or JavaScript mode, an autocompletion plugin is available (Ctrl + Space).
- Increasing/decreasing the font size (Ctrl +, Ctrl -)
- In Solidity mode, you can display the gas estimation (Tools -> Display Gas Estimation). This will highlight all statements which requires a minimum amount of gas. Color turns to red if the gas required becomes important. It will also display the max execution cost of a transaction (for each function).

The Whisper Specific Commands

Whisper is a hybrid DHT/point-to-point communications system. It allows for transient publication and subscription of messages using a novel topic-based routing system. It gives exceptional levels of privacy, and can be configured to provide a privacy/efficiency tradeoff per application.

Here follows a few gotchas in the present whisper API on the `eth` interactive console. It will be updated shortly.

Points to note:

- All API for whisper is contained in the `web3.shh` object. Type it in the interactive console to get a list of all functions and variables provided.
- Topic values are user-readable strings and as such should be given as plain text. E.g., topic `"zxcv"` in the console corresponds directly to the topic `"zxcv"`.
- Payload should be in JSONRPC standard data representation (i.e. hexadecimal encoding). E.g., text `"zxcv"` must be represented as `"0x847a786376"`. This can be done through usage of the `web3.fromAscii` function.
- The function `web3.shh.request()` returns the request object instead of sending it to the core (except for some complicated functions). E.g. `web3.shh.post.request({topicis: ["0xC0FFEE"]})`
- When sending/posting a message, don't omit the `ttl` parameter, because default value is zero, and therefore the message will immediately expire, before being sent.
- The functions `filter.get()` and `web3.shh.getMessages()` should return the same messages, but in a different format.

Examples of usage

```
var f = web3.shh.filter({topics: ["qwerty"]})
f.get()
web3.shh.getMessages("qwerty")
web3.shh.post({topics: ["qwerty"], payload: "0x847a786376", ttl: "0x1E", workToProve: "0x
```

Cold Wallet Storage Device

A Cold Wallet Storage Device (CWSD) is a device (duh) used to store keys and sign transactions which never touches the internet, or indeed any communications channels excepting those solely for basic user interaction. The use of such a device is pretty much necessary for storing any large sum of value or other blockchain-based asset, promise or instrument. For example, a device like this has been used for operating blockchain-based keys worth many millions of dollars.

For this how-to, we'll assume that the CWSD is a simple Ubuntu-based computer (a netbook works pretty well) with TurboEthereum preinstalled as per the first chapter; I will assume that you've taken the proper precautions to avoid any malware getting on to the machine (though without an internet connection, there's not too much damage malware can realistically cause).

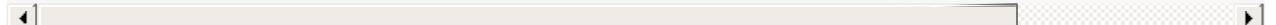
Kill the network

The first thing to do is to make sure you've disabled any network connection, wireless or otherwise. Maybe compile a kernel without ICP/IP and Bluetooth, maybe just destroy or remove the network hardware of the computer. It is this precaution that puts the 'C' in CWSD.

Generate the keys

The next thing to do is to generate the key (or keys) that this machine will store. Run `ethkey` to create a wallet and then again to make as many keys as you would like to use. You can always make more later. For now I'll make one:

```
> ethkey createwallet
Please enter a MASTER passphrase to protect your key store (make it strong!): password
Please confirm the passphrase by entering it again: password
> ethkey new supersecret
Enter a passphrase with which to secure this account (or nothing to use the master passph
Please confirm the passphrase by entering it again: password
Enter a hint to help you remember this passphrase: just 'password'
Created key 055dde03-47ff-dded-8950-0fe39b1fa101
Name: supersecret
Password hint: just 'password'
ICAP: XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ
Raw hex: 0092e965928626f8880629cec353d3fd7ca5974f
```



It will prompt for a password and confirmation for both commands. I'm just going to use the password "password" for both.

This "supersecret" key has an address of `XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ` .

Signing with the keys

Signing with the keys can happen in two ways: The first is to export a transaction to sign from e.g. AlethZero, perhaps saving to a USB pendrive. Let's assume that is what we have done and we have the hex-encoded transaction at `/mnt/paygav.tx` .

In order to sign this transaction we just need a single `ethkey` invocation:

```
> ethkey sign supersecret /tmp/paygav.tx
```

It will prompt you for the passphrase and finally place the signed hex in a file

`/mnt/paygav.tx.signed` . Easy. If we just want to copy and paste the hex (we're too paranoid to use pen drives!) then we would just do:

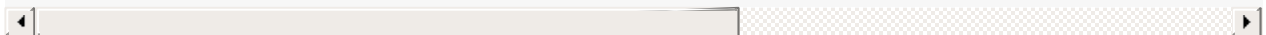
```
> echo "<hex-encoded transaction here>" | ethkey sign supersecret
```

At which it will ask for your passphrase and spit out the hex of the signed transaction.

Alternatively, if we don't yet have an unsigned transaction, but we actually want to construct a transactions locally, we can do that too.

Let's assume our "supersecret" account has received some ether in the meantime and we want to pay somebody 2.1 grand of this ether (2100 ether for those not used to my English colloquialisms). That's easy, too.

```
> ethkey sign supersecret --tx-dest <destination address> --tx-gas 55000 --tx-gasprice 50
```



Note the `--tx-value` (the amount to transfer) and the `--tx-gasprice` (the price we pay for a single unit of gas) must be specified in Wei, hence the large numbers there. `--tx-nonce` only needs to be specified if it's not the first transaction sent from this account.

Importing the key

You may want to eventually import the key to your everyday device. This may be to use it directly there or simply to facilitate the creation of unsigned transactions for later signing on the CWSD. Assuming you have a strong passphrase, importing the key on to a hot device

itself should not compromise the secret's safety too much (though obviously it's materially less secure than being on a physically isolated machine).

To do this, simply copy the JSON file(s) in your `~/.web3/keys` path to somewhere accessible on your other (non-CWSD) computer. Let's assume this other computer now has our "supersecret" key at `/mnt/supersecret.json`. There are two ways of importing it into your Ethereum wallet. The first is simplest:

```
> ethkey import /mnt/supersecret.json supersecret
Enter the passphrase for the key: password
Enter a hint to help you remember the key's passphrase: just 'password'
Imported key 055dde03-47ff-dded-8950-0fe39b1fa101
Name: supersecret
Password hint: just 'password'
ICAP: XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ
Raw hex: 0092e965928626f8880629cec353d3fd7ca5974f
```

A key can only be added to the wallet whose address is known; to figure out the address, `ethkey` will you to type your passphrase.

This is less than ideal since if the machine is actually compromised (perhaps with a keylogger), then an attacker could slurp up your passphrase and key JSON and be able to fraudulently use that account as they pleased. Ouch.

A more secure way, especially if you're not planning on using the key directly from this hot machine in the near future, is to provide the address manually on import. It won't ask you for the passphrase and thus potentially compromise the secret's integrity (assuming the machine is actually compromised in the first place!).

To do this, I would remember the "supersecret" account was

`XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ` and tell `ethkey` as such while importing:

```
> ethkey importwithaddress XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ supersecret
Enter a hint to help you remember the key's passphrase: just 'password'
Imported key 055dde03-47ff-dded-8950-0fe39b1fa101
Name: supersecret
Password hint: just 'password'
ICAP: XE472EVKU3CGMJF2YQ0J9R01Y90BC0LDFZ
Raw hex: 0092e965928626f8880629cec353d3fd7ca5974f
```

In both cases, we'll be able to see the key in e.g. AlethZero as one of our own, though we will not be able to sign with it without entering the passphrase. Assuming you never enter the passphrase on the hot machine (but rather do all signing on the CWSD) then you should be

reasonably safe. Just be warned that the security of the secret is lying on the network security of your hot machine *and* the strength of your key's passphrase. I really wouldn't count on the former.