Geth introduction

WHAT

Before we write smart contracts, we are going to acquaint ourselves with Geth and, by extension, the mechanics of Ethereum. This is not a nice-to-know aside, it is crucial to the proper coding of smart contracts.

Geth is the Ethereum node software coded in Go language. It is widely used and well-maintained.

HOW

Geth already comes bundled with Mist, but here we are going to install it separately, launch it with specific configurations then interact with it through simple commands. In any case, it is not the software itself that is using a lot of storage space, but rather the blockchain it downloads and the indices it creates.

We are going to work with Geth on Linux, in effect, in the virtual machine you installed earlier. This means you will also be ready to install Geth on cloud computers.

EXERCISES

Install

The official install documentation is found here. So, to install on Ubuntu:

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

Your computer or your VM will need at least 3 GB of RAM and 128 MB of video RAM for Geth to be able to mine. Make sure you allocate at least this, and certainly more if you open a browser in your VM too.

Run

Geth comes with a set of preconfigured settings to connect to well-known networks.

Main net

If you now launch Geth:

```
$ geth version
Geth
Version: 1.6.7-stable
Git Commit: ab5646c532292b51e319f290afccf6a44f874372
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.8.1
Operating System: linux
GOPATH=
GOROOT=/usr/lib/go-1.8

$ geth
```


Launching Geth

You should see it synchronising from the main network. You can see the data it creates, and increasing fast too, in:

```
$ ls ~/.ethereum/
geth  geth.ipc  keystore

$ ls ~/.ethereum/geth/
chaindata  ethash  LOCK  nodekey  nodes

$ du -hs ~/.ethereum
3.9M /home/xxx/.ethereum

$ du -hs ~/.ethereum/geth/chaindata
3.9M /home/xxx/.ethereum/geth/chaindata
```

Stop it with CTRL-C because it will take too long anyway. Also:

- `geth.ipc` exists if and only if Geth is running.
- this `chaindata/` folder is where the downloaded data and indexed data is stored. It is safe to delete its content in order to start afresh. *Safe* in the sense that you can always download the whole lot at the next run.

Test net

Now let's have it synchronise against the test network nicknamed "Ropsten", with `id 3`.

Test nets change regularly, and there is more than 1 at any time. The original test net was named *Morden* and had `id 2`.

```
$ geth --testnet
```

Similarly, you should see it synchronising from the test network. Be patient, it may take 5 minutes for the synchronisation to start. See the data it creates in:

```
$ ls ~/.ethereum/testnet
geth  geth.ipc  keystore

$ ls ~/.ethereum/testnet/geth
chaindata  ethash  LOCK  nodekey  nodes

$ du -hs ~/.ethereum/testnet
112M /home/xxx/.ethereum/testnet
```

You could let it run its course because the default `--syncmode` is `"fast"` so it should take you less than an hour. But let's move on, so `CTRL-C`. Let's clear the chain data already downloaded, unless you want to keep it, that is:

```
$ geth --testnet removedb
WARN [08-02|20:53:51] No etherbase set and no accounts found as default
/home/xxx/.ethereum/testnet/geth/chaindata
Remove this database? [y/N] y
Remove this database? [y/N] y
INFO [08-02|20:53:54] Database successfully deleted           database=chaindata
INFO [08-02|20:53:54] Database doesn't exist, skipping        database=lightchai

$ geth removedb
WARN [08-02|20:56:24] No etherbase set and no accounts found as default
/home/xxx/.ethereum/geth/chaindata
Remove this database? [y/N] y
Remove this database? [y/N] y
INFO [08-02|20:56:25] Database successfully deleted           database=chaindata
INFO [08-02|20:56:25] Database doesn't exist, skipping        database=lightchai
```

And let's create our own network.

If you are interested in other test nets, here is a non-exhaustive list:

- Rinkeby
- Kovan

Private network

Creating your own network, what does it mean? We have already seen how a blockchain can fork, and how, when so doing, can create an alternative reality, an alternative truth. To create your own network is also to create an alternative reality. But unlike a fork, you create an entirely new

blockchain, one whose transactions and blocks, including the first one, are not shared with other networks, or blockchains.

To define a new network, all you need, nothing more, nothing less, is:

- a network id
- a genesis file

Indeed, other clients will not agree to connect to your blockchain unless they have the same network id *and* the same genesis file. Additionally, if and when it is needed, to help computers interconnect, you will then need to guide Geth on how to find *peers*.

> For the avoidance of doubt:
>
> - even if another node has a long chain of blocks starting with the right genesis, and you want to connect to it, you absolutely need the genesis file yourself. Just having the network id will not be enough.
> - even if you think of your private network as private, because of the Ethereum protocol, anyone with the right parameters can connect to it. You are not protected from random connections. Unless you specifically instruct Geth not to look for and accept connections.

Network id

The network id is any positive number you like, although in order to be a good citizen, you may want to avoid:

- `1`, which is reserved for the main chain, commonly referred to as "the" Ethereum network.
- `2`, which is reserved for the late Morden test chain.
- `3`, which is reserved for the Ropsten test chain.
- `4`, which is reserved for the Rinkeby test chain.

> All hard-forks that happened since the launch of a chain have this id, and that includes the DAO hard-fork. So both ETH and ETC have
> `id == 1`
> .

Genesis block

This block is the first block and the only one without a predecessor. When accounts have been "pre-mined", it is here that allocations are afforded. This is, in effect, the only block that is agreed on by humans, socially, and not by machines, cryptographically.

When you create, or recreate, your own chain, you need to create this genesis block only once.

For Geth, it will be a JSON file similar to this one, which we are going to save and use:

```
// Remove this comment and save the rest in "genesis42.json"
{
    "config": {
        "chainId": 42,
        "homesteadBlock": 0,
        "eip150Block": 0,
        "eip150Hash":"0x0000000000000000000000000000000000000000000000000000000000000000
        "eip155Block": 0,
        "eip158Block": 0
    },
    "nonce": "0x0000000000000042",
    "timestamp": "0x00",
    "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000
    "extraData": "0x00",
    "gasLimit": "0x4c4b40",
    "difficulty": "0x0400",
    "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000
    "coinbase": "0x0000000000000000000000000000000000000000",
    "alloc": {
    }
}
```

Notice how difficulty is set extremely low, here `0x400`, i.e. `1,024`. This is to ensure that Geth, on your host or in your VM, is able to mine right off the bat. So the first blocks will be mined extremely fast before stabilising to 10-15 seconds for every block, when Geth has adjusted the difficulty. We think it is better to start low and go up, than the other way round.

If you omitted this `difficulty` parameter, or spelled it wrongly, or placed the genesis in the wrong location, it would start as per the main net, i.e. `0x400000000`. You can confirm this later with `> eth.getBlock(0).difficulty`. The way to fix that is to delete the content of `~/.ethereum/net42/geth/chaindata` and start again.

The `config` element defines at which blocks this network uses certain protocol changes that have appeared later in the life of the main net.

In practice, you do not need to preallocate ether to your account because you will be able to mine hundreds of Ethers in a matter of minutes. If you still want to preallocate Ether to certain accounts, you just need to add the balance in `wei`:

```
...
    "alloc": {
        "0x1fb891f92eb557f4d688463d0d7c560552263b5a": {
            "balance": "20000000000000000000"
        }
    }
}
```

> Note that the main net and both the Ropsten and Rinkeby test nets all have their own genesis files, which are hard-coded into the Geth client for convenience.

Start the chain

Let's suppose that the network id is `42` and the genesis file is in `~/Documents/genesis42.json`. Beside the mentioned parameters, in order to avoid mixing up with the main chain data, you also need to specify a data folder, in our case let's choose `~/.ethereum/net42`. Let's save the following code in `~/Documents/launch42.sh`, this will save you from repeatedly mis-typing it:

```
#!/bin/bash

# Run this once, but it does not hurt to run it every time
geth --datadir ~/.ethereum/net42 init ~/Documents/genesis42.json
# Run this every time you start your Geth "42", and add flags here as you need
geth --datadir ~/.ethereum/net42 --networkid 42
```

Don't forget to make your file executable:

```
$ cd ~/Documents

$ chmod a+x launch42.sh

# Launch it now
$ ./launch42.sh
```

Confirm that you have these lines:

```
INFO [08-02|21:20:31] Writing custom genesis block
INFO [08-02|21:20:31] Successfully wrote genesis state         database=chaindata
...
INFO [08-02|21:20:31] Writing custom genesis block
INFO [08-02|21:20:31] Successfully wrote genesis state         database=lightchai
```

If you do not see these lines, then the `init ~/Documents/genesis42.json` command was unsuccessful.

Notice too this line, which opens the communication channel through which Mist talked to Geth:

```
INFO [08-02|21:20:33] IPC endpoint opened: /home/xxx/.ethereum/net42/geth.ipc
```

Here, it is unlikely, possible but unlikely, to synchronise by itself with random strangers on the internet, and there is not much you can do beside `CTRL-C` to stop it.

> The `init` command has to be run once. In particular, if you clear the content of `~/.ethereum/net42/chaindata`, you need to run `init` again.

Geth console

In order to interact with our node, let's add a console to it. Update the line in `launch42.sh`:

```
...
geth --datadir ~/.ethereum/net42 --networkid 42 console
```

Launch again: `$ ./launch42.sh`

The console is a Javascript console that lets you send commands to Geth. Since it is a Javascript one, all Javascript things apply, such as declaring variables. What do we have here in this console?

Type `eth`, that gives you the list of available methods. You can also type `eth.` then press `TAB`twice, for code suggestions. Let's go through a few commands.

```
> eth.accounts
[]
```

No account, for you have not created any yet. What exactly happened here with this command, you ask? Well, the console is connected to Geth via the `geth.ipc` file. Now, when you type and

return `eth.accounts` you are actually making a synchronous call to Geth to retrieve its list of accounts.

Let's make a synchronous call to get the latest block number:

```
> eth.blockNumber
0
```

We are still on the first block, for you have not synchronised or mined anything.

```
> eth.getBlock(0)
{
  difficulty: 1024,
  extraData: "0x00",
  gasLimit: 5000000,
  gasUsed: 0,
  hash: "0x426ab4e1d1a9090a6f7570326453f75677bede506468633c3e75a3ec4ec0a0c9",
  logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000000000
  miner: "0x0000000000000000000000000000000000000000",
  mixHash: "0x0000000000000000000000000000000000000000000000000000000000000000",
  nonce: "0x0000000000000042",
  number: 0,
  parentHash: "0x0000000000000000000000000000000000000000000000000000000000000000
  receiptsRoot: "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b4
  sha3Uncles: "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
  size: 506,
  stateRoot: "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421"
  timestamp: 0,
  totalDifficulty: 1024,
  transactions: [],
  transactionsRoot: "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e3
  uncles: []
}
```

Here we find again the parameters of our custom genesis block. Notice also how `receiptsRoot`, `stateRoot` and `transactionsRoot` are equal. These 3 roots are the 3 Merkle roots that each Ethereum block header contains. And since we have nothing but an empty tree for the 3 of them, they share the same root hash.

Let's keep on poking:

```
> eth.mining
false
```

Obviously, no miner has started.

```
> net.version
"42"
```

Your network id again, as specified in the command-line.

```
> net.peerCount
0
```

You are alone on this network.

A small trick to see the list of initial commands. Type 2 spaces then hit TAB twice. You will get:

```
>
Array                JSON                 SyntaxError          clearTimeout
BigNumber            Math                 TypeError            console
Boolean              NaN                  URIError             constructor
Date                 Number               Web3                 debug
Error                Object               _setInterval         decodeURI
EvalError            RangeError           _setTimeout          decodeURIComponent
Function             ReferenceError       admin                encodeURI
GlobalRegistrar      RegExp               callback             encodeURIComponent
Infinity             String               clearInterval        escape
```

Accounts

Before you can do anything meaningful with Geth, you need accounts. You can create as many accounts as you want. Creating an account, really, just means creating a private key locally. It does not mean that other computers are aware of it. Before you have sent any transaction, that is.

Let's create one, type:

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0x6c503786685f23abae76976fe0aa843f75ea9e71"
> INFO [08-02|21:37:47] New wallet appeared                          url=keystore:///

// As a synchronous call
> eth.accounts
["0x6c503786685f23abae76976fe0aa843f75ea9e71"]

// As an asynchronous call
> eth.getAccounts(function(err, acc1){ console.log(err, acc1); })
null 0x6c503786685f23abae76976fe0aa843f75ea9e71
undefined

// As a synchronous call
```

```
> eth.coinbase
"0x6c503786685f23abae76976fe0aa843f75ea9e71"

// As an asynchronous call
> eth.getCoinbase(function (err, coinbase) { console.log(err, coinbase); })
null 0x6c503786685f23abae76976fe0aa843f75ea9e71
undefined
```

You need to save or remember the passphrase as it is used to encrypt your private key on disk. The same passphrase is then required to decrypt the private key when you need to use it:

```
> personal.unlockAccount(eth.accounts[0])
Unlock account 0x6c503786685f23abae76976fe0aa843f75ea9e71
Passphrase:
true
```

Now the account is unlocked, but only for around 5 minutes, after which time it is back to being locked. So, remember this `personal.unlockAccount` command, you will have to use it a fair number of times. Also, as seen in its documentation, you can pass it a password and a duration.

Your encrypted private key is stored in:

```
$ ls ~/.ethereum/net42/keystore/
UTC--2016-05-23T16-27-26.470952057Z--6c503786685f23abae76976fe0aa843f75ea9e71

$ cat ~/.ethereum/net42/keystore/UTC--2016-05-23T16-27-26.470952057Z--6c503786685
{"address":"6c503786685f23abae76976fe0aa843f75ea9e71","crypto":{"cipher":"aes-128
```

If you want to back-up your private key, this is the file, and passphrase, you need to back-up.

Let's see how many Ethers this account holds:

```
// As a synchronous call
> eth.getBalance(eth.accounts[0])
0

// As an asynchronous call
> eth.getBalance(eth.accounts[0], function(err, bal) { console.log(err, bal); })
null 0
undefined
```

Of course, a new account starts empty. Or rather, when the blockchain contains no mention of this account, then its balance defaults to `0`. That's something you will see again later. When something was not set or seen before, its value is `0`. Not `null`, nor `undefined`, nor `nil`...

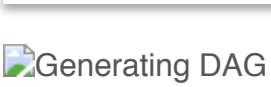
Mining

Ok, let's get some mining rewards! The miner needs to be started with a command. It uses `eth.coinbase` for the mining rewards. `eth.coinbase`, which, unless you specify it, defaults to `eth.accounts[0]`. Confirm that `eth.coinbase` returns you an account and let's start mining:

```
> eth.coinbase
"0x6c503786685f23abae76976fe0aa843f75ea9e71"

> miner.start(1)
true
```

> You will need to use this command frequently. The `1` means that it only starts one process. You don\'t need more on your development node because you are only competing against yourself. Increasing the number of processes will only increase the eventual difficulty, it will not increase the speed at which blocks are created.



The first launch will take time as it creates the 1.5GB Ethash file. This extra-large Ethash file is part of Ethereum's pick of a hashing algorithm that foils ASIC-based mining. The hashing function used for mining is so taxing in terms of RAM that only computers can mine, with the current state of technology.

Take this opportunity to stretch your legs and refill your cup.

After a few blocks, which should take about a minute, *unless you did not size the RAM of your VM adequately*:

```
> miner.stop()
true

// As a synchronous call
> eth.blockNumber
7

// As an asynchronous call
> eth.getBlockNumber(function (err, number) { console.log(err, number); })
null 7
undefined

// As 2 synchronous calls
> eth.getBalance(eth.accounts[0])
35000000000000000000
```

```
// As a synchronous call and an asynchronous one
> eth.getBalance(eth.accounts[0], function (err, balance) { console.log(err, bala
null 35000000000000000000
undefined

// As nested asynchronous calls
> eth.getAccounts(function(err, accounts) { eth.getBalance(accounts[0], function
null 35000000000000000000
undefined

> eth.getBalance(eth.coinbase)
35000000000000000000

> web3.fromWei(eth.getBalance(eth.accounts[0]), "ether")
35
```

Here, we have mined 7 blocks. Each mined block awards 5 Ethers. Your account collects every mining award because you are alone on this network. Which is why you already collected 35 ethers.


Mining

Transactions

So far all the blocks you have mined were empty:

```
// As a synchronous call
> eth.getBlock(0).transactions.length
0

// As an asynchronous call
> eth.getBlock(0, function(err, block) { console.log(err, block.transactions.leng
null 0
undefined

> eth.getBlock(7).transactions.length
0

> eth.getBlock(7)
{
  difficulty: 131072,
  extraData: "0xd783010607846765746887676f312e382e31856c696e7578",
  gasLimit: 4990242,
  gasUsed: 0,
  hash: "0xd75f72bd016e0b82d5ac78975702537390553ebc2e39c266ca91a578bc5f5e87",
  logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000000
  miner: "0x6c503786685f23abae76976fe0aa843f75ea9e71",
  mixHash: "0xa5f2d12171b18a5fed5f6b07b42f9cb8d0db38ec85aa6c927b62092ecb7ed575",
  nonce: "0x10381021e4c07035",
  number: 7,
```

```
  parentHash: "0x2646ce7e83c6186d020660e1873a6d2d9f26e721da5fd7dda44d975939a653d7
  receiptsRoot: "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b4
  sha3Uncles: "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
  size: 535,
  stateRoot: "0xf0b25d6fb01ee92c74ecfd47db366f75bd539fa36bf5d0610cbc5dc400ae5ad9"
  timestamp: 1501707482,
  totalDifficulty: 263168,
  transactions: [],
  transactionsRoot: "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e3
  uncles: []
}
```

Notice how `transactionsRoot` and `receiptsRoot` have not changed, i.e. they are still representing empty Merkle trees. On the other hand, `stateRoot` has changed and for a good reason: it has stored your account's balance.

Let's use a command that will no longer work when blocks are too big:

```
> debug.dumpBlock("0x7")
{
  accounts: {
    6c503786685f23abae76976fe0aa843f75ea9e71: {
      balance: "35000000000000000000",
      code: "",
      codeHash: "c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
      nonce: 0,
      root: "56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
      storage: {}
    }
  },
  root: "f0b25d6fb01ee92c74ecfd47db366f75bd539fa36bf5d0610cbc5dc400ae5ad9"
}
```

Here is the full state of this Ethereum blockchain at the moment:

- a single known address, for which:
- `balance` is as expected.
- (transaction) `nonce` to 0 because this account has not sent any transaction.
- no `code` because this an externally owned account, not a smart contract.
- no `storage` because, at leat for now, such account cannot update its storage.
- `root` is the Merkle root of the storage tree.

Let's change that and launch a transaction which will send Ethers from our first account to another one. To start this part of the exercise, ensure you have run `miner.stop()`. First let's create that other account:

```
> eth.mining
false

> personal.newAccount()
Passphrase:
Repeat passphrase:
"0xaa2d08a053b8f007cc685e72975e31bcd336a028"
> INFO [08-02|22:13:33] New wallet appeared                          url=keystore:///

> eth.accounts
["0x6c503786685f23abae76976fe0aa843f75ea9e71", "0xaa2d08a053b8f007cc685e72975e31b

> eth.getBalance(eth.accounts[1])
0
```

Let's send some Ether to this second account:

```
> web3.toWei(5, "ether")
"5000000000000000000"

> eth.sendTransaction({ from: eth.accounts[0], to: eth.accounts[1], value: web3.t
account is locked
```

Locked? Oh!, of course, we unlocked it some time ago and now it is locked again.

> The `{ from: ... }` JSON object is the **transaction parameters object**. With it, you can define all or part of the transaction parameters. At this stage you should remember that *each and every transaction* needs this object, however sparse. Remember this particularly when we learn how to interact with smart contracts. And that we invoke functions through transactions.

We see the `from` key among the transaction parameters. Indeed, Geth needs the private key of the sender's account to sign the transaction. However, accounts are locked by default for your ~~annoyance~~ safety, and so you need to unlock them with the passphrase prior any transaction. So, as we have seen, to unlock the account interactively for a limited amount of time:

```
> personal.unlockAccount(eth.accounts[0])
Unlock account 0x6c503786685f23abae76976fe0aa843f75ea9e71
Passphrase:
true

// Try again as a synchronous call
> var txHash = eth.sendTransaction({ from: eth.accounts[0], to: eth.accounts[1],
INFO [08-02|22:17:23] Submitted transaction                    fullhash= 0xbe2125
undefined

> txHash
"0xbe212587a808dea6fb635197937dbbc3f19634d69933079e6e6f2c00271e6e5d"
```

It went through, a transaction was sent and the result hash is that of the transaction. Notice that `accounts[1]`, i.e. the recipient, did not need to be unlocked. This recipient could well be an account whose key you do not possess; that would amount to a payment to a third party. This recipient couls also well be a smart contract; that would amount to sending Ether to this account.

Let's see what the transaction represents:

```
// As a synchronous call
> eth.getTransaction(txHash)
{
  blockHash: "0x0000000000000000000000000000000000000000000000000000000000000000"
  blockNumber: null,
  from: "0x6c503786685f23abae76976fe0aa843f75ea9e71",
  gas: 90000,
  gasPrice: 20304857463,
  hash: "0xbe212587a808dea6fb635197937dbbc3f19634d69933079e6e6f2c00271e6e5d",
  input: "0x",
  nonce: 0,
  r: "0xd035d58d3bbf0a9ec0efff89e72ef14111b919f982b64e0131e4f5a6b9c11cbc",
  s: "0x1dbeaa0c3ecb82a18f5cb30a3af9d0ba9956f974a0dd9659ec721992ff2ffa1b",
  to: "0xaa2d08a053b8f007cc685e72975e31bcd336a028",
  transactionIndex: null,
  v: "0x1b",
  value: 5000000000000000000
}

// Try your hand at the asynchronous call equivalent
```

Notice how `blockHash` is `0x0`, `blockNumber` is `null` and `transactionIndex` is `null`; these are indicators of a *pending* transaction, i.e. it has not been included in any block, i.e. it has not been mined. Let's confirm that:

```
> txpool.content.pending
{
  0x6c503786685f23abae76976fe0aa843f75ea9e71: {
```

```
    0: {
      blockHash: "0x0000000000000000000000000000000000000000000000000000000000000000
      blockNumber: null,
      from: "0x6c503786685f23abae76976fe0aa843f75ea9e71",
      gas: "0x15f90",
      gasPrice: "0x4a817c800",
      hash: "0xbe212587a808dea6fb635197937dbbc3f19634d69933079e6e6f2c00271e6e5d",
      input: "0x",
      nonce: "0x0",
      r: "0xd035d58d3bbf0a9ec0efff89e72ef14111b919f982b64e0131e4f5a6b9c11cbc",
      s: "0x1dbeaa0c3ecb82a18f5cb30a3af9d0ba9956f974a0dd9659ec721992ff2ffa1b",
      to: "0xaa2d08a053b8f007cc685e72975e31bcd336a028",
      transactionIndex: null,
      v: "0x1b",
      value: "0x4563918244f40000"
    }
  }
}
```

It is the same transaction with values displayed as hexadecimals.

Notice also the `gas` field, at `90000`, which:

- is the total gas sent along the transaction,
- acts as the upper limit of consumable gas for this transaction.

Notice also the `nonce: 0`. This transaction is your account's first ever transaction. All transactions are sequentially numbered. Your account's next transaction will have `nonce: 1`. If you were to force your first transaction to, say, `nonce: 1`, miners would then keep this transaction in their cache, on a best effort basis, waiting for your actual first transaction. This also assists miners in ordering your transactions, should you submit many at the same time, and should they be transmitted between peers in a different order.

Let's check the balance of the recipient:

```
> eth.getBalance(eth.accounts[1])
0
```

Still `0` indeed because the transaction has not been mined yet.

Let's fix that and start the miner:

```
> miner.start(1)
true
```

Wait for it to mine 5 blocks, then:

```
> miner.stop()
true

> eth.getBalance(eth.accounts[1])
5000000000000000000

> eth.getTransaction(txHash)
{
  blockHash: "0xf8e7cee207ce8bd16b6e57a21357389fc99e9e221b08c96e42cfab127fe8c5a8"
  blockNumber: 8,
  from: "0x6c503786685f23abae76976fe0aa843f75ea9e71",
  gas: 90000,
  gasPrice: 20304857463,
  hash: "0xbe212587a808dea6fb635197937dbbc3f19634d69933079e6e6f2c00271e6e5d",
  input: "0x",
  nonce: 0,
  to: "0xaa2d08a053b8f007cc685e72975e31bcd336a028",
  transactionIndex: 0,
  value: 5000000000000000000
}
```

We now see how fields have been populated.

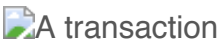
Just to experiment with additional commands, let's access the same transaction via
its `blockNumber : 8`,
its `blockHash: "0xf8e7cee207ce8bd16b6e57a21357389fc99e9e221b08c96e42cfab127fe8c5a8"`,
and its `transactionIndex: 0`:

```
// As a synchronous call
> eth.getTransactionFromBlock(8, 0)
{
  blockHash: "0xf8e7cee207ce8bd16b6e57a21357389fc99e9e221b08c96e42cfab127fe8c5a8"
  blockNumber: 8,
  from: "0x6c503786685f23abae76976fe0aa843f75ea9e71",
  gas: 90000,
  gasPrice: 20304857463,
  hash: "0xbe212587a808dea6fb635197937dbbc3f19634d69933079e6e6f2c00271e6e5d",
  input: "0x",
  nonce: 0,
  to: "0xaa2d08a053b8f007cc685e72975e31bcd336a028",
  transactionIndex: 0,
  value: 5000000000000000000
}

> eth.getTransactionFromBlock("0xf8e7cee207ce8bd16b6e57a21357389fc99e9e221b08c96e
{
  blockHash: "0xf8e7cee207ce8bd16b6e57a21357389fc99e9e221b08c96e42cfab127fe8c5a8"
  blockNumber: 8,
  from: "0x6c503786685f23abae76976fe0aa843f75ea9e71",
  gas: 90000,
```

```
  gasPrice: 20304857463,
  hash: "0xbe212587a808dea6fb635197937dbbc3f19634d69933079e6e6f2c00271e6e5d",
  input: "0x",
  nonce: 0,
  to: "0xaa2d08a053b8f007cc685e72975e31bcd336a028",
  transactionIndex: 0,
  value: 5000000000000000000
}
```

> If you want to have a look at how much gas was actually consumed during the transaction, you can run `> eth.getTransactionReceipt(txHash)`. The receipt of a transaction sent to a smart contract usually also contains information on contract events.



Interconnect

We now assume that you are sufficiently confident with Geth and its console, and that you have a network id and genesis file that is common to your cohort.

Anyone starting Geth with the parameters seen above will create a private blockchain, unless, that is, they connect to each other to exchange and ensure they are on the same network. And we have indeed seen cases where students connect to each other by chance on the "net42" network.

Interconnection can take place randomly over an indeterminate, and possibly very long, period of time, or we can instruct Geth to look for a specific computer. In particular, your Geth node opens a port to receive potential peer connection requests. When you start Geth, you can tell it to listen for peers on a specific port, by default `--port 30303`.

Keep in mind that if 2 different Geth nodes have been running, and mining, with the same parameters but isolated from each other, they have in effect created 2 different forks. When they eventually interconnect, they will decide, according to the protocol, which fork they will both keep working on. The "losing" party may thus appear to lose Ethers.

Node info

Every running Geth is identifiable by an *enode* id. To find yours, in Geth console, run:

```
> admin.nodeInfo.enode
"enode://4955a02eb11a66c8a24d678567e66b3918ce4357ff10dec695c7384332132e0e0f66c3a0
```

- The long hexadecimal part is an encoded public key uniquely identifying the node. In particular, when connecting to a remote node, it is used by Geth to ascertain that it is not looping back to itself. So, when you copy your data directory to another computer, make sure you do not copy the `geth/nodekey` file as it will prevent your 2 nodes from connecting to each other. Instead, let Geth automatically create a new `nodekey` file when starting the other computer.
- When you submit the enode information to another computer, you need to replace `[::]` with the accessible IP address of the original computer.

You can also run `> admin.nodeInfo` and see more information.

Static node

There may be a static node that will be accessible by all. This increases centralisation but allows for a quick setup. When that is the case, collect the enode id of the static node(s) and save it in the file `<datadir>/static-nodes.json`, where `datadir` is the same data directory you pass in your Geth command line, in the form:

```
[
    "enode://4955a02eb11a66c8a24d678567e66b3918ce4357ff10dec695c7384332132e0e0f66
]
```

Where you have replaced `PUBLIC_IP_ADDRESS` with the IP address of the remote computer you connect to.

To let others mine for Ether, you may want to have this central node mine as little as possible, if at all. All the more so that you may want this static node to have a small attack surface, and so to hold no account.

> Actually, it is possible to send the mining rewards to an account you do not own. Even to a smart contract. Just use `miner.setEtherbase`. We use this trick to refill our Ropsten faucet.

Dynamic node

When the `<datadir>/static-nodes.json` file does not work, or in the easy case where both your nodes are on the same LAN, after every start, in the Geth console, you run:

```
> admin.addPeer("enode://4955a02eb11a66c8a24d678567e66b3918ce4357ff10dec695c73843
```

Where you have replaced `PUBLIC_IP_ADDRESS` with the proper value of the other node.

Verification

How to verify that 2 computers are on the same blockchain?

With a transaction

You may send a transaction to an account and cross-check the values between nodes.

Alternatively, you can *burn* ethers. This is morally ok on this private blockchain. For instance, in the Geth console:

```
> eth.sendTransaction({from: "0x8c1e6c4a4d610e33ba765de341fd12361c2fae2f", to: "0
```

Wait for it to mine, then, on both sides, you should see:

```
> eth.getBalance("0x0000000000000000000000000000000000000000")
1
```

With the latest block

It comes in two parts:

- Check that they both share the network id:

```
> net.version
"42"
```

- Check that they have the same latest block:

```
// On the 1st computer, as a synchronous call:
> eth.getBlock("latest")
{
  ...
  hash: "0x84c5099b83cd3c400ebbda85aee10e6976328f6c13bf5be4a75e8a8d0278365a",
  ...
  nonce: "0x0c40e19fc02dcb11",
  number: 388,
  ...
}

// On the 2nd computer, replace "latest" with the number, or the hash:
> eth.getBlock(388)
{
```

```
  ...
  hash: "0x84c5099b83cd3c400ebbda85aee10e6976328f6c13bf5be4a75e8a8d0278365a",
  ...
  nonce: "0x0c40e19fc02dcb11",
  number: 388,
  ...
}
```

Confirm that at the same height of `388`, blocks have the same hash.