You are required to design a system that allows users to create a private blockchain consisting of 5 validators. The nodes must run the geth client to run the private blockchain node.

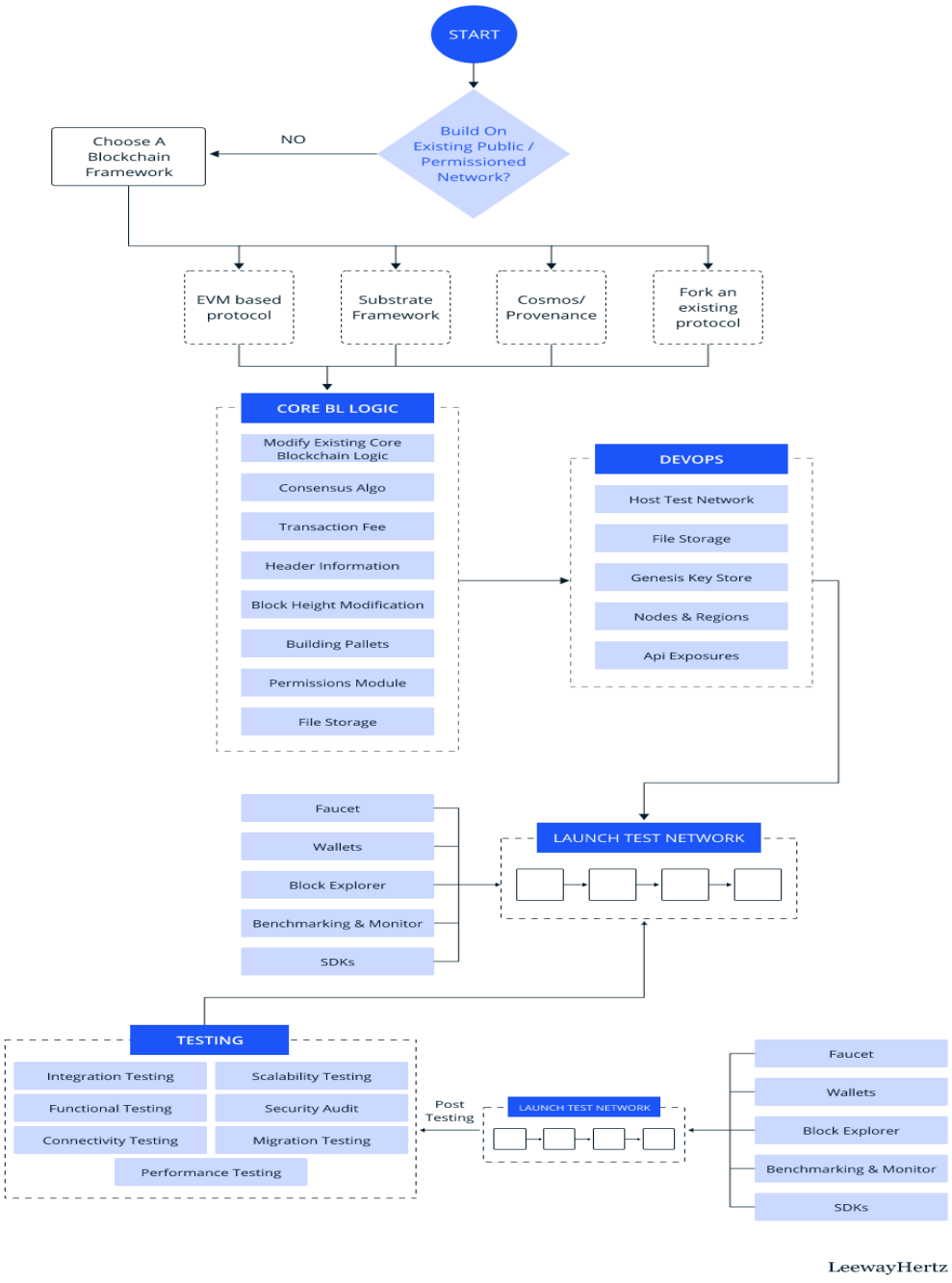
Private BlockChain

- These are similar to public blockchains, but they are managed centrally.
- This authority determines who can participate in the network, verify transactions, and maintain the shared ledger.
- These networks are not fully decentralized, as the public has limited access to them. While some entities see the merits of blockchain technology, others could live without its transparency and public nature.
- This is why private blockchains with a secure and closed database make more sense to organizations.

Important features of a private blockchain

- Only a few individuals or entities can see the network's nodes.
- The mining opportunity is not open to the public. Some cryptocurrencies can even be pre-mined.
- Only a few people have the right to review and audit.
- P2P transactions are possible because it is decentralized.
- They are mostly immutable.

Core Logic of Creating Private BlockChain



Steps involved are:

1. Select the protocol
2. Build the core logic
3. Development of the logic
4. Testing
5. Launch the main network

6. Network integration

Different ways to set up a Private Blockchain Network for development & testing

1. Geth
2. Docker and Docker Compose
3. Terraform and Amazon ECS
4. Amazon Managed Blockchain
5. Polygon

Geth:

Go Ethereum (abbreviated as Geth) is an implementation of the Ethereum blockchain, which is written in GoLang.

There are different ways for creating a private blockchain locally

- Use GETH github repo to create a private ethereum blockchain
- Setup a private ethereum blockchain using Geth and Brew Commands.

Using GETH github:

Environment

- Linux mint 19.1
- Go (version 1.10 or later)
- geth (go-ethereum)

Download and install geth

- First clone geth from go-ethereum repository to GOPATH directory
- my GOPATH is /home/ubuntu/go so i cloned geth to

  /home/ubuntu/go/src/github.com
- git clone https://github.com/ethereum/go-ethereum.git

Then geth will be downloaded. after downloading, move to go-ethereum directory and open terminal and run below command

- make all

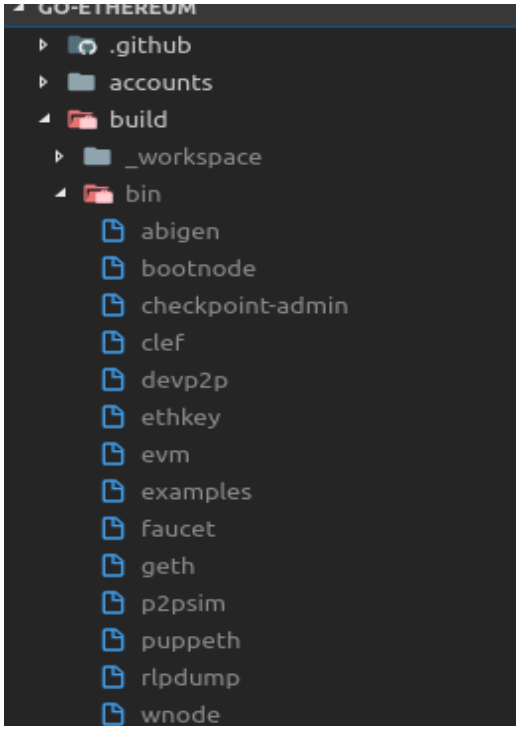
If we want to use only geth console, then

- make geth

.However, If we want to test with multi-nodes someday, then we have to do

- make all

If build is successful then we can see geth in /go-ethereum/build/bin



wecan see geth file

Build private network

First of all we have to make a genesis.json. wecan make it in any folder, any directory except a directory which need permission, but for convenience, I recommend make it in same folder with geth (/go-ethereum/build/bin)

```
{
"config":{ "chainId": 15,
      "homesteadBlock": 0,
      "eip155Block": 0,
      "eip158Block": 0
   },
   "difficulty": "400",
   "gasLimit": "2100000",
   "alloc": {
   }
}
```

We can change chainId. chainId is the most important part of this .json Because, geth can recognize a network we want to access. In the Ethereum network, There is already defined chainID.

```
0: Olympic; Ethereum public pre-release testnet
1: Frontier; Homestead, Metropolis, the Ethereum public main network
1: Classic; The (un)forked public Ethereum Classic main network, chain ID 61
1: Expanse; An alternative Ethereum implementation, chain ID 2
2: Morden; The public Ethereum testnet, now Ethereum Classic testnet
3: Ropsten; The public cross-client Ethereum testnet
4: Rinkeby: The public Geth Ethereum testnet
42: Kovan; The public Parity Ethereum testnet
77: Sokol; The public POA testnet
99: POA; The public Proof of Authority Ethereum network
7762959: Musicoin; The music blockchain
```

Therefore, except these, we can use any chainId like 15 what i did.Second, I changed the difficulty for easy mining.Third, I set alloc to empty. It is used for funding ether to a already generated accounts For example, If we generate the account 7df9a875a174b3bc565e6424a0050ebc1b2d1d82 and .json is

```
{

  "config": {

      "chainId": 15,
```

```
    "homesteadBlock": 0,

    "eip155Block": 0,

    "eip158Block": 0

  },

  "difficulty": "400",

  "gasLimit": "2100000",

  "alloc": {

    "7df9a875a174b3bc565e6424a0050ebc1b2d1d82": { "balance": "300000" },

  }

}
```

then 300000 will be charged after initiation. Don’t need to worry about initiation.

Now we have to make a directory to store information. We made a geth-test folder. We can make it anywhere. my directory is /home/ubuntu/Documents/geth-test

Finally, We are ready to start. Move to /go-ethereum/build/bin, open terminal and follow below command

```
$ ./geth --datadir your_own_storage init genesis.json
```

For example, in our case

```
$ ./geth --datadir /home/ubuntu/Documents/geth-test init genesis.json
```

After initiation, check own storage folder

```
INFO [08-06|20:49:41.266] Bumping default cache on mainnet         provided=1024 updated=4096
INFO [08-06|20:49:41.268] Maximum peer count                       ETH=50 LES=0 total=50
INFO [08-06|20:49:41.268] Smart Card socket not found, disabling    err="stat /run/pcscd/pcscd.comm: no such file or directory"
INFO [08-06|20:49:41.271] Allocated cache and file handles         database=/home/ubuntu/Documents/geth-test/geth/chaindata cache=16.00MiB handles=16
INFO [08-06|20:49:41.284] Writing custom genesis block
INFO [08-06|20:49:41.284] Persisted trie from memory database      nodes=0 size=0.00B time=3.725µs gcnodes=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [08-06|20:49:41.285] Successfully wrote genesis state         database=chaindata hash=ab944c…55600c
INFO [08-06|20:49:41.285] Allocated cache and file handles
```

```
database=/home/ubuntu/Documents/geth-test/geth/light chain data cache=16.00MiB
handles=16
INFO [08-06|20:49:41.302] Writing custom genesis block
INFO [08-06|20:49:41.302] Persisted trie from memory database      nodes=0
size=0.00B time=3.632µs gcnodes=0 gcsize=0.00B gctime=0s livenodes=1
livesize=0.00B
INFO [08-06|20:49:41.303] Successfully wrote genesis state
database=lightchaindata hash=ab944c…55600c
```

If initiation is done well, then we can see geth and keystore folder in your storage

- Run the ls command

The above command will list two directories geth and keystore

To run our private network!

open terminal in /go-ethereum/build/bin and run below command

```
$ ./geth --datadir Your_own_storage --networkid 15 console
```

We set the chainid as 15 in genesis.json Therefore our networkid is 15 too In our case,

```
$ ./geth --datadir /home/ubuntu/Documents/geth-test --networkid 15 console
```

then we can see

```
INFO [08-06|21:27:43.867] Maximum peer count                       ETH=50 LES=0
total=50
INFO [08-06|21:27:43.867] Smart Card socket not found, disabling    err="stat
/run/pcscd/pcscd.comm: no such file or directory"
INFO [08-06|21:27:43.870] Starting peer-to-peer node
instance=Geth/v1.9.2-unstable-aa6005b4-20190805/linux-amd64/go1.12.7
INFO [08-06|21:27:43.870] Allocated trie memory caches             clean=256.00MiB
dirty=256.00MiB
INFO [08-06|21:27:43.870] Allocated cache and file handles
database=/home/ubuntu/Documents/geth-test/geth/chaindata cache=512.00MiB
handles=524288
INFO [08-06|21:27:43.904] Opened ancient database
database=/home/ubuntu/Documents/geth-test/geth/chaindata/ancient
INFO [08-06|21:27:43.904] Initialised chain configuration          config="{ChainID: 15
Homestead: 0 DAO: <nil> DAOSupport: false EIP150: <nil> EIP155: 0 EIP158: 0
Byzantium: <nil> Constantinople: <nil>  Petersburg: <nil> Engine: unknown}"
INFO [08-06|21:27:43.904] Disk storage enabled for ethash caches
dir=/home/ubuntu/Documents/geth-test/geth/ethash count=3
INFO [08-06|21:27:43.904] Disk storage enabled for ethash DAGs
dir=/home/ubuntu/.ethash count=2
INFO [08-06|21:27:43.904] Initialising Ethereum protocol           versions=[63]
network=15 dbversion=7
INFO [08-06|21:27:43.944] Loaded most recent local header          number=0
hash=ab944c…55600c td=400 age=50y3mo3w
INFO [08-06|21:27:43.944] Loaded most recent local full block      number=0
hash=ab944c…55600c td=400 age=50y3mo3w
INFO [08-06|21:27:43.944] Loaded most recent local fast block      number=0
hash=ab944c…55600c td=400 age=50y3mo3w
INFO [08-06|21:27:43.945] Loaded local transaction journal         transactions=0
dropped=0
INFO [08-06|21:27:43.945] Regenerated local transaction journal    transactions=0
accounts=0
INFO [08-06|21:27:43.951] Allocated fast sync bloom                size=512.00MiB
INFO [08-06|21:27:43.951] Initialized fast sync bloom              items=0 error
rate=0.000 elapsed=37.353µs
INFO [08-06|21:27:43.997] New local node record                    seq=3
```

```
id=65c5b16ab4aa9e9f ip=127.0.0.1 udp=30303 tcp=30303
INFO [08-06|21:27:43.998] Started P2P networking
self=enode://3e6e6cc9fd56954e02f3807813e086827ddf0576d0c969f67a915691ec3f8
79867332ba4911048fd513672856c63a2746063706005c6d777f670ae16c2c4a384@1
27.0.0.1:30303
INFO [08-06|21:27:43.999] IPC endpoint opened
url=/home/ubuntu/Documents/geth-test/geth.ipc
WARN [08-06|21:27:44.088] Served eth_coinbase                    reqid=3 t=16.874µs
err="etherbase must be explicitly specified"
Welcome to the Geth JavaScript console!

instance: Geth/v1.9.2-unstable-aa6005b4-20190805/linux-amd64/go1.12.7
at block: 0 (Thu, 01 Jan 1970 09:00:00 KST)
datadir: /home/ubuntu/Documents/geth-test
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0
rpc:1.0 txpool:1.0 web3:1.0
>
```

At the last line, wecan see the console pointer(>). there are some lines we need to

check

```
INFO [08-06|21:27:43.904] Initialised chain configuration          config="{ChainID: 15
Homestead: 0 DAO: <nil> DAOSupport: false EIP150: <nil> EIP155: 0 EIP158: 0
Byzantium: <nil> Constantinople: <nil>  Petersburg: <nil> Engine: unknown}"
```

In this line, we can check ChainID is 15 like we expected. Now our private network is

running!

Test private network

Now let’s test our private network

```
> eth.blockNumber
```

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

eth.blockNumber checks the number of blocks. We just generate this network.

Therefore there is 0 block. eth.account check accounts of the network. There are no

accounts. Let’s generate account

```
> personal.newAccount("Alice")
INFO [08-06|21:33:36.241] Your new key was generated
address=0xb8C941069cC2B71B1a00dB15E6E00A200d387039
WARN [08-06|21:33:36.241] Please backup your key file!
path=/home/ubuntu/Documents/geth-test/keystore/UTC--2019-08-06T12-33-34.44282
3142Z--b8c941069cc2b71b1a00db15e6e00a200d387039
WARN [08-06|21:33:36.241] Please remember your password!
"0xb8c941069cc2b71b1a00db15e6e00a200d387039"
```

We just generated the address of

Alice:0xb8C941069cC2B71B1a00dB15E6E00A200d387039. We can see it by using

geth

```
> eth.accounts
["0xb8c941069cc2b71b1a00db15e6e00a200d387039"]
```

we will use it as miner’s address so block generation reward will be sent to Alice’s

address Before mining, let’s check Alice’s balance by any one below command

```
> eth.getBalance("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
0
> eth.getBalance(eth.accounts[0])
0
```

We will use 3 commands

- miner.setEtherbase(address): It sets the miner's address. Mining reward will be sent to this account
- miner.start(number of threads): Start mining. wecan set how many threads we will use. I will use 1 thread
- miner.stop(): Stop mining

```
> miner.setEtherbase("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
true
> miner.start(1)
null
INFO [08-06|21:42:38.198] Updated mining threads                   threads=1
INFO [08-06|21:42:38.198] Transaction pool price threshold updated
price=1000000000
null
> INFO [08-06|21:42:38.198] Commit new mining work                   number=1
sealhash=4bb421…3f463a uncles=0 txs=0 gas=0 fees=0 elapsed=325.066µs
INFO [08-06|21:42:40.752] Successfully sealed new block            number=1
sealhash=4bb421…3f463a hash=4b2b78…4808f6 elapsed=2.554s
INFO [08-06|21:42:40.752] 🔨 mined potential block                 number=1
hash=4b2b78…4808f6
.
.
.
INFO [08-06|21:42:56.174] 🔨 mined potential block                 number=9
hash=2faebb…8be693
INFO [08-06|21:42:56.174] Commit new mining work                   number=10
sealhash=384aa6…cb0596 uncles=0 txs=0 gas=0 fees=0 elapsed=179.463µs
> miner.stop()
null
```

We finished mining. Now let’s check it worked well.

```
> eth.blockNumber
9
```

In our case, we mined 9 blocks until we stopped it.

```
> eth.getBalance("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
45000000000000000000
```

1 ether = 10^18 wei

We can convert it to ether by command

```
>
web3.fromWei(eth.getBalance("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
, "ether")
45
```

Make a transaction for testing private network

Generate new account

```
> personal.newAccout("Bob")
INFO [08-06|22:00:23.416] Your new key was generated
address=0xf39Cf42Cd233261cd2b45ADf8fb1E5A1e61A6f90
WARN [08-06|22:00:23.416] Please backup your key file!
path=/home/ubuntu/Documents/geth-test/keystore/UTC--2019-08-06T13-00-21.62117
2635Z--f39cf42cd233261cd2b45adf8fb1e5a1e61a6f90
WARN [08-06|22:00:23.416] Please remember your password!
"0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90"
> eth.getBalance("0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90")
0
```

I got the account of Bob:0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90 Alice will

send ether to Bob's account

```
> eth.sendTransaction({from: "0xb8c941069cc2b71b1a00db15e6e00a200d387039",
to: "0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90", value: web3.toWei(5, "ether")})
```

Let’s send 5 ether to Bob’s account

- Alice’s account : 0xb8c941069cc2b71b1a00db15e6e00a200d387039
- Bob’s account : 0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90

Or we can initialize these using variable

```
> from = "0xb8c941069cc2b71b1a00db15e6e00a200d387039"
> to = "0xb8c941069cc2b71b1a00db15e6e00a200d387039"
> eth.sendTransaction({from: from, to: to, value: web3.toWei(5, "ether")})
```

We sent ether, however, ***don’t forget we never use private key of Alice.*** Therefore we

met Error

```
WARN [08-06|22:04:55.407] Served eth_sendTransaction              reqid=25
t=3.461656ms err="authentication needed: password or unlock"
Error: authentication needed: password or unlock
  at web3.js:3143:20
  at web3.js:6347:15
  at web3.js:5081:36
  at <anonymous>:1:1
```

We have to unlock Alice’s account. Let’s see the status of Alice’s account.

```
> personal.listWallets[0].status
```

```
"Locked"
```

It is locked So, we have to unlock it to send ether from Alice to Bob

```
>
web3.personal.unlockAccount("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
Unlock account 0xb8c941069cc2b71b1a00db15e6e00a200d387039
```

Alice’s address is 0xb8c941069cc2b71b1a00db15e6e00a200d387039. However we have to type a Passphrase of Alice. passphrase is Alice because we generate this address using Alice

```
> personal.newAccount(“Alice”)
Passphrase: Alice
true
```

Alice’s account is unlocked. Let’s go back to the transaction. We can see pending transactions using below command

```
> eth.pendingTransactions
[]
```

Until now, there have not been any transactions. We just unlocked Alice’s account. Let’s make a transaction again.

```
> eth.sendTransaction({from: "0xb8c941069cc2b71b1a00db15e6e00a200d387039",
to: "0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90", value: web3.toWei(5, "ether")})
INFO [08-06|22:16:09.274] Setting new local account
address=0xb8C941069cC2B71B1a00dB15E6E00A200d387039
INFO [08-06|22:16:09.275] Submitted transaction
fullhash=0x926f1bb71d5b48a306e6cde2d45c01f8af2107febf94b166a7e5f8e025dc8a
dc recipient=0xf39Cf42Cd233261cd2b45ADf8fb1E5A1e61A6f90
```

```
"0x926f1bb71d5b48a306e6cde2d45c01f8af2107febf94b166a7e5f8e025dc8adc"
```

There is no error. Let's see pending transactions

```
> eth.pendingTransactions
[{
  blockHash: null,
  blockNumber: null,
  from: "0xb8c941069cc2b71b1a00db15e6e00a200d387039",
  gas: 21000,
  gasPrice: 1000000000,
  hash:
"0x926f1bb71d5b48a306e6cde2d45c01f8af2107febf94b166a7e5f8e025dc8adc",
  input: "0x",
  nonce: 0,
  r: "0x70484271bdc85f7233e715423d8d0be5c669a323385b5ec0ff080a52cf3c654c",
  s:
"0x1b55a792995f61128c10a48ce1e0869893c863d38489f574d84ae3a96b031cef",
  to: "0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90",
  transactionIndex: null,
  v: "0x42",
  value: 5000000000000000000
}]
```

There is a transaction.

```
> eth.getBalance("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
45000000000000000000
> eth.getBalance("0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90")
0
```

We didn't mine any block. so There is no change of balance yet. Let's mine again!

```
> miner.start(1)
INFO [08-06|22:19:53.061] Updated mining threads                   threads=1
INFO [08-06|22:19:53.061] Transaction pool price threshold updated
price=1000000000
null
> INFO [08-06|22:19:53.062] Commit new mining work                   number=10
```

```
sealhash=f69cfb…273c0d uncles=0 txs=0 gas=0 fees=0 elapsed=265.557µs
INFO [08-06|22:19:53.062] Commit new mining work                   number=10
sealhash=a018f5…65f494 uncles=0 txs=1 gas=21000 fees=2.1e-05
elapsed=1.022ms
INFO [08-06|22:19:54.718] Successfully sealed new block            number=10
sealhash=a018f5…
.
.
.
INFO [08-06|22:20:05.086] 🔨 mined potential block                 number=16
hash=e7688a…09ed64
INFO [08-06|22:20:05.086] Commit new mining work                   number=17
sealhash=6b297d…b76b19 uncles=0 txs=0 gas=0     fees=0       elapsed=252.945µs
> miner.stop()
null
> eth.blockNumber
16
```

Last time Alice mined 9 blocks and this time mined 7 blocks more. So We can expect

Alice to have 75 ether (80 ether block reward — 5 ether sent to Bob = 75 ether). Let’s

check pending transactions

```
> eth.pendingTransactions
[]
```

There is no pending transaction. Alice and Bob’s transaction is done!Let’s see balance

of them

```
> eth.getBalance("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
75000000000000000000
> eth.getBalance("0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90")
5000000000000000000
```

Setup a private ethereum blockchain using Geth and Brew Commands:
Installing Geth

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install ethereum
```

Now Wehave Geth available to you on the command-line.

Creating an Account
We Need to create a private/public keypair first so We can write transactions to the blockchain. We do this in Geth with the following command (note: do not forget the passphrase We choose).

```
ubuntu> geth account new
Your new account is locked with a password. Please give a password. Do not forget this password.
Passphrase:
Repeat Passphrase:
Address: {<xxxxxx>}
```

What We see next to Address is your wallet address. our wallet and encrypted private key are stored in a file in the following locations based on operating system:

- OSX: ~/Library/Ethereum
- Linux: ~/.ethereum
- Windows: %APPDATA%\Ethereum

If We look inside that file, We will only see our encrypted key, never the unencrypted key, and some other metadata about the key. Geth does not support storing these private keys unencrypted.

When We want to use that key to create a transaction using Geth, We will need to enter the passphrase We created so Geth can decrypt our private key in that file.

Wecan see our account by running:

```
ubuntu> geth account list
```

The genesis.json file:

The genesis file determines two things:

- what will take place in the genesis block, or the first block of our blockchain.
- The configuration rules our blockchain will follow.

Store the following into a genesis.json file in whatever directory Weare running your command-line commands. We will use this for our private blockchain:

```
// genesis.json
{
 "config": {
   "chainID": 1234,
   "homesteadBlock": 0,
   "eip155Block": 0,
   "eip158Block": 0
 },
 "alloc": {
    "0x<YOUR WALLET'S ADDRESS>": {
      "balance": "1000000000000000000000000000000"
    }
 },
 "difficulty": "0x4000",
 "gasLimit": "0xffffffff",
 "nonce": "0x0000000000000000",
 "coinbase": "0x0000000000000000000000000000000000000000",
 "mixhash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
 "parentHash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
 "extraData":
"0x123458db4e347b1234537c1c8370e4b5ed33adb3db69cbdb7a38e1e50b1b82fa",
 "timestamp": "0x00"
}
```

Note: Remember the address that creating the new account gave us? Be sure to substitute that address in under "alloc" where it says <our wallet's address>. This is us telling Geth we want the first block in our blockchain to give 100 billion ether to that address.

We can do this because it's the first block in the entire chain and we are the creators of this chain. We have the power to decide who starts out with what. That's what the first block of any cryptocurrency does, including the main Ethereum blockchain.

Creating and storing the blockchain
Just as the mainnet Ethereum blockchain resides on multiple computers, we will simulate having multiple computers storing our blockchain.

We will create two nodes, which act like two different computers hosting and interacting with the same blockchain. Any node that wants to interact with a blockchain will want to store the blockchain on their computer somewhere (it can be too large to keep in memory). Using the –datadir flag, we specify where we want to store the blockchain data as we create it and it continues to grow.

Since we're going to create two different nodes, we will run Geth from two different Terminal windows to simulate two different computers. We will also use two different datadir directories so each node has a separate place to store their local copy of our blockchain.

As mentioned, the three major operating systems by default store their blockchains in the following locations:

- OSX: ~/Library/Ethereum
- Linux: ~/.ethereum
- Windows %APPDATA%\Ethereum.

We're creating a new, completely private blockchain, so let's store it elsewhere.

Run the following to store Node #1's copy of our private blockchain in a "LocalNode1" folder (it will get created with this command):

```
geth --datadir "~/Library/LocalNode1" init genesis.json
```

(This assumes the genesis.json file We Created is in the same directory from which you're running these commands.)

Run the following to store Node #2's copy of our private blockchain in a "LocalNode2" folder:

```
geth --datadir "~/Library/LocalNode2" init genesis.json
```

We can look inside the directories Geth just created at ~/Library/LocalNode1 and ~/Library/LocalNode2. You'll soon notice how the contents of the "geth" directory grows as we add to our chain.

Interacting with the blockchain
Now that we have the first block written and our configuration variables set, we can launch the geth console to interact with the blockchain from our first node.

```
geth --datadir "~/Library/LocalNode1" --networkid 1234 --port 11111 --nodiscover console
```

The networkid here has to match the chainID we set in the genesis.json file.
--nodiscover means that even though we're running an Ethereum client, we don't want other people in the world trying to connect to our chain. Finally, port is the port number our node will communicate with other peers over.

Once We run that command, Weshould see a message Welcome to the Geth JavaScript console! appear towards the bottom of your terminal.

We have just opened our console to interact with our first node's copy of our blockchain. Now, open another tab in Terminal. We'll now set up a console for our second Node to interact with the same blockchain.

```
geth --datadir "~/Library/LocalNode2" --networkid 1234 --port 11112 --nodiscover console
```

Notice that the datadir corresponds to the directory where we are storing the blockchain for our second node (LocalNode2) and that we changed port to 11112 so our two nodes are not colliding on the same port.

List wallets
From node 1, run personal.listWallets. This will list everything inside the _keystore_ file in the datadir directory for that node.

```
> personal.listWallets
[]
```

Right now, this yields an empty set [].

We created a wallet at the beginning of this tutorial, and then specified in the genesis.json file that we want to allocate an initial balance of ether to that wallet in the genesis block. When we created that wallet, Geth by default places the encrypted file in ~/Library/Ethereum/keystore/.

We are using a custom datadir since we're not interacting with the mainnet blockchain, so we need to copy that file into the keystore directory we're using. Copying that file over will give our node access to that wallet file:

```
ubuntu> cp ~/Library/Ethereum/keystore/UTC--<rest of wallet file's name>
~/Library/LocalNode1/keystore/
```

Alternatively, when we launched the Geth console, we could have explicitly used the --keystore flag to tell Geth where our wallet files were.

After copying over the file, go back to Node 1 and run personal.listWallets.

```
> personal.listWallets
[{
    accounts: [{
        address: "0x<your wallet address>",
        url: "keystore:///Users/<Your
username>Library/LocalNode1/keystore/UTC--<datecreated>Z--<your wallet
address>"
    }],
    status: "Locked",
    url: "keystore:///Users/<Your
username>/Library/LocalNode1/keystore/UTC--<datecreated>Z--<your wallet
address>"
}]
```

To see just the wallet address, Wecan also run:

```
> personal.listAccounts
["<your wallet address>"]
```

As mentioned, we allocated some ether to this wallet in the genesis block via our genesis.json file. Let's see if it worked:

This genesis block we created recognizes the wallet we control as the owner of 100 billion ether on this private chain. If We want to be rich, it's our job to convince the world to use our chain instead of the one of the mainnet.

Connecting to Peers
Nodes 1 and 2 have the same genesis state files and configurations, so if they could communicate with one-another, they would build on the same blockchain.

We set the --nodiscover flag earlier to ensure these nodes don't automatically interact with other nodes. We'll have to manually tell one of the nodes about the other's existence. Currently, entering net.peerCount or admin.peers produces 0 and the empty set respectively.

Tab over to Node 2 and enter admin.nodeInfo.enode. What gets returned is the following:

```
> admin.nodeInfo.enode
"enode://<enode ID>@<ip>:<TCP discovery port>?discport=<UDP discovery port>"
```

The enode ID is a unique hexadecimal value. Immediately after it is the "@" sign and then the IP address of the node. In our case, since everything is running locally, this will return [::].

After the IP address is a colon and the TCP discovery port number that the enode is listening on. If We recall, we set Node 2 to listen on port 1112, so that's what Wewill see here.

Finally, after we have discport=. This tells us the UDP discovery port for that node. Since we did not set this, it returns 0.

Copy this entire NodeUrl and tab back over to Node 1. Copy it all into the admin.addPeer function like so:

```
admin.addPeer("enode://341793ade41a145c1987d2ade...")
true
```

Now, running admin.peers from either node will give We detailed information about the other node.

If We want to automate this process in the future, We can set up static nodes, which allows nodes to automatically connect with known enodes.

Coinbase/Etherbase
The coinbase or etherbase we set for a node is the address that will collect the rewards of any mining that takes place on that node.

Right now, checking the coinbase on either of our nodes yields the following error:

```
> eth.coinbase
Error: etherbase address must be explicitly specified
    at web3.js:3143:20
    at web3.js:6347:15
    at get (web3.js:6247:38)
    at <unknown>
```

We need to explicitly set it.
If we wanted to set our pre-existing account as the coinbase, we can do so with the following:

```
> miner.setEtherbase(eth.accounts[0])
```

But, let's create a separate address for our mining rewards. We can create an account through the geth console like so:

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
["<new wallet address"]
```

We will see this new encrypted key appear in the keystore directory in the datadir for this node.
Now, entering eth.accounts produces an array of two addresses: the original we had at index 0, and the new one we created at index 1. We can set the new wallet address as the coinbase like so:

```
miner.setEtherbase(eth.accounts[1])
```

We can verify this value by then entering eth.coinbase, which should return the wallet address we just set.

Transactions

We will now send a transaction from a wallet address on node 1 to a wallet address on node 2. We have yet to create a wallet on node 2, so go to that geth tab and do the same process from earlier:

```
// On node 2
> personal.newAccount()
Passphrase:
Repeat passphrase:
["<new wallet address on node 2"]
```

This encrypted private key will be in the keystore file in the datadir for node 2.

Now, tab back to Node 1. We will send 20 ether to the wallet address on node 2 like so:

```
> eth.sendTransaction({from: eth.accounts[0], to: "<wallet address on node 2>", value: 20})
```

This will yield an error, like so:

```
Error: authentication needed: password or unlock
    at web3.js:3143:20
    at web3.js:6347:15
    at web3.js:5081:36
    at <anonymous>:1:1
```

The reason we get this error is because, at this point, our private key file is still encrypted for our first wallet address. This protects an untrusted third party with access to the file in the keystore directory from being able to issue transactions without knowing the passphrase.

First, we unlock the account (decrypt the private key for geth's use) by running personal.unlockAccount(eth.accounts[0]). After entering the passphrase, the key will be decrypted in memory and ready to sign transactions for geth.

Now, try the same sendTransaction command from earlier, and Wewill get a response that the transaction has been submitted.

```
> eth.sendTransaction({from: eth.accounts[0], to: "<wallet address on node 2>", value: 20})
INFO [MM-DDD|HH:MM:SS] Submitted transaction
fullhash=<transaction hash> recipient=<wallet address on node 2>
"<transaction hash>"
```

The transaction has been submitted. But, if we go to Node 2 and check the balance of the address to which we sent the ether, we see the following:

```
// Checking the balance of the destination address on node 2
> web3.fromWei(eth.getBalance(eth.accounts[0]), "ether")
0
```

Since we have submitted the transaction, it has yet to be _mined_ and therefore it has not been added to the blockchain.
Going back to node 1, we simply run:

```
> miner.start()
```

Because our difficulty is so low, we'll see blocks mined very quickly. After a few seconds, enter miner.stop().

Now, go back to Node 2 and enter the same command from above to check accounts[0]'s balance:

```
> web3.fromWei(eth.getBalance(eth.accounts[0]), "ether");
20
```

The transaction has successfully been written to our private blockchain, and the wallet owned by node 2 now has 20 ether!

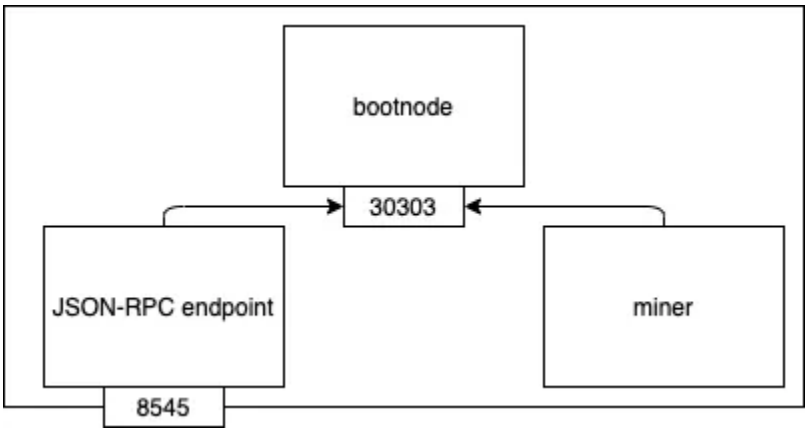
We'll see that the coinbase account we created on Node 1 will also have some ether now:

```
> eth.getBalance(eth.coinbase)
<amount in Wei>
```

Set up a Private Blockchain Network for development & testing using Docker
The private Ethereum network nodes

Ethereum network is a peer-to-peer network consisting of multiple nodes running the Ethereum client such as Geth or OpenEthereum. we are setting up a private network with 3 nodes as follows:

1. Bootnode — the bootstrap node that is used for peer discovery purposes. It listens on port *30303*, the other nodes joining the network connect to this bootnode first.
2. JSON-RPC endpoint — this node exposes JSON-RPC API over HTTP endpoint on port *8545*. We will publish the port *8545* of this node container to the host machine to allow external interaction with this private blockchain.
3. Miner — this node is responsible for mining (the process of creating a new block in our blockchain). When the miner node successfully mines a new block, it receives the rewards into the configured account.



The genesis block

To set up a private Ethereum blockchain, the Ethereum client requires some information for creating the first block which is called the genesis block.

Below is a simple custom *genesis.json* containing information to create the genesis block.

```
{
  "config": {
    "chainId": 1214,
    "homesteadBlock": 0,
    "eip150Block": 0,
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
    "petersburgBlock": 0,
    "ethash": {}
  },
  "difficulty": "1",
  "gasLimit": "12000000",
  "alloc": {}
}
```

- config.chainId — this is the identifier to tell the nodes which blockchain they are on. The *chainId* was introduced in EIP-155 for replay protection. We set it to be `1214` to avoid conflicting with the public Ethereum networks.
- config.ethash — this indicates that our blockchain will be using *proof-of-work* as the consensus engine.

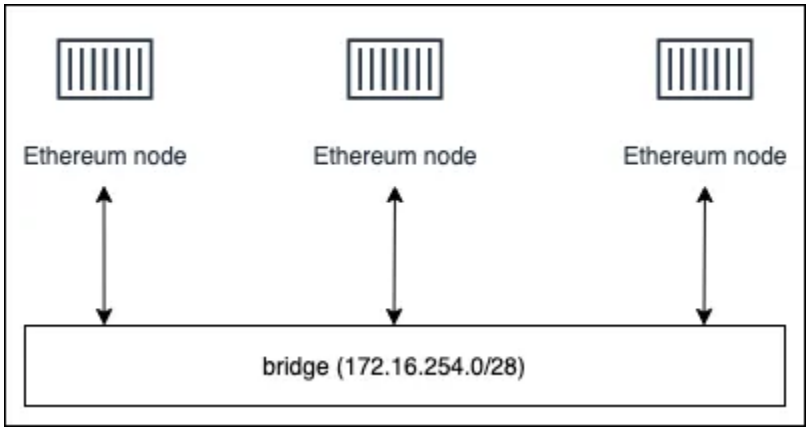
Docker image for Ethereum client

For the Ethereum nodes in our private blockchain, we will use Go Ethereum (Geth) as the client. So let's create a *Dockerfile* for building the image of our Ethereum client.

```
FROM ethereum/client-go:v1.10.1

ARG ACCOUNT_PASSWORD

COPY genesis.json /tmp

RUN geth init /tmp/genesis.json \
    && rm -f ~/.ethereum/geth/nodekey \
    && echo ${ACCOUNT_PASSWORD} > /tmp/password \
    && geth account new --password /tmp/password \
    && rm -f /tmp/password

ENTRYPOINT ["geth"]
```

Note:The `ACCOUNT_PASSWORD` will be provided as an argument while building an image from the *Dockerfile*.

Configuring Ethereum nodes in Docker Compose
As described earlier, our private Ethereum network consists of 3 nodes connected together. In order to connect them, we will put all nodes in 172.16.254.0/28 subnet in Docker's bridge network.



The cluster of Ethereum nodes in Docker
The Docker Compose file to run the nodes in the Docker environment.

```
version: '3.7'

services:
  geth-bootnode:
    hostname: geth-bootnode
```

```
    env_file:
      - .env
    image: geth-client
    build:
      context: .
      args:
        - ACCOUNT_PASSWORD=${ACCOUNT_PASSWORD}
    command:

--nodekeyhex="b0ac22adcad37213c7c565810a50f1772291e7b0ce53fb73e7ec2a3c7
5bc13b5"
      --nodiscover
      --ipcdisable
      --networkid=${NETWORK_ID}
      --netrestrict="172.16.254.0/28"
    networks:
      priv-eth-net:

  geth-rpc-endpoint:
    hostname: geth-rpc-endpoint
    env_file:
      - .env
    image: geth-client
    depends_on:
      - geth-bootnode
    command:

--bootnodes="enode://af22c29c316ad069cf48a09a4ad5cf04a251b411e45098888d11
4c6dd7f489a13786620d5953738762afa13711d4ffb3b19aa5de772d8af72f851f7e9c5b
164a@geth-bootnode:30303"
      --allow-insecure-unlock
      --http
      --http.addr="0.0.0.0"
      --http.api="eth,web3,net,admin,personal"
      --http.corsdomain="*"
      --networkid=${NETWORK_ID}
      --netrestrict="172.16.254.0/28"
    ports:
      - "8545:8545"
    networks:
      priv-eth-net:

  geth-miner:
    hostname: geth-miner
    env_file:
      - .env
```

```
    image: geth-client
    depends_on:
      - geth-bootnode
    command:

--bootnodes="enode://af22c29c316ad069cf48a09a4ad5cf04a251b411e45098888d11
4c6dd7f489a13786620d5953738762afa13711d4ffb3b19aa5de772d8af72f851f7e9c5b
164a@geth-bootnode:30303"
      --mine
      --miner.threads=1
      --networkid=${NETWORK_ID}
      --netrestrict="172.16.254.0/28"
    networks:
      priv-eth-net:

networks:
  priv-eth-net:
    driver: bridge
    ipam:
      config:
      - subnet: 172.16.254.0/28
```

As we want to have 3 nodes performing different roles in our private blockchain, therefore, each node will be configured with a different set of parameters.

1. Bootnode command parameters

- nodekeyhex — we specify the nodekey for the bootnode in order to pre-define the enode (as it is generated from the nodekey) of this bootstrap node to use it for configuring other nodes.
- nodiscover — this node does not have to discover other nodes because the others will connect to it when joining the network.
- ipcdisable — to make the node more light-weight as it is only used as bootstrap node.
- networkid — specify the identifier of the network, every node in the same Ethereum network must have the same networkid. Avoid the values which conflict with the public Ethereum network.
- netrestrict — to only accept the connection from nodes within the CIDR range.

2. JSON-RPC endpoint node command parameters

- bootnodes— specify the list of nodes to connect to for peer discovery

- allow-insecure-unlock — to allow unlocking the account we created over the HTTP. This is unsafe if the endpoint is exposed to external, consider removing this parameter in the production environment.
- http — to enable JSON-RPC over HTTP protocol.
- http.api — the list of APIs to enable via HTTP-RPC interface. In the production environment, specify only the ones which are required.
- http.addr — set to 0.0.0.0 to accept HTTP connection on all IPs of the node container.
- http.corsdomain — to allow connection from cross-origin web pages.
- networkid — specify the identifier of the network, every node in the same Ethereum network must have the same networkid. Avoid the values which conflict with the public Ethereum network.
- netrestrict — to only accept the connection from nodes within the CIDR range.

3. Miner node command parameters

- bootnodes — specify the list of nodes to connect to for peer discovery
- mine — to enable the mining for this node
- miner.threads — specify the number of CPU thread to use for mining
- miner.etherbase—specify the address of the account to receive mining rewards. We do not specify it here, so the rewards go to the primary account we created by default.
- networkid — specify the identifier of the network, every node in the same Ethereum network must have the same networkid. Avoid the values which conflict with the public Ethereum network.
- netrestrict — to only accept the connection from nodes within the CIDR range.
- Running and interacting with the private Ethereum network

To run our private Ethereum network from the docker-compose.yaml file, we have to specify 2 environment variables in the .env file

```
# The ID of Ethereum Network
NETWORK_ID=1214

# The password to create and access the primary account
ACCOUNT_PASSWORD=5uper53cr3t
```

Then start the nodes with the command:

```
docker-compose up -d
```

After the containers are up and running, we can interact with the blockchain via the Inter-process Communication (IPC) or the RPC endpoint.

We can use a software wallet, such as MetaMask or MyCrypto, to connect to our private blockchain via the HTTP-RPC endpoint. However, this post will use curl to demonstrate how we interact with our private blockchain.

As we map the HTTP-RPC port 8545 from the container to our host machine, we can then connect to it at localhost:8545 .

1. First of all, let's check the connectivity of the nodes.

```
curl --location --request POST 'localhost:8545' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "admin_peers",
    "params": []
}'
```

Since we are talking to the RPC endpoint node, we will see that the peer it is connecting to is the bootnode (recognized from the enode value is the same as we set as bootnodes parameters in the docker-compose.yml).

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    {
      "enode":
"enode://af22c29c316ad069cf48a09a4ad5cf04a251b411e45098888d114c6dd7f48
9a13786620d5953738762afa13711d4ffb3b19aa5de772d8af72f851f7e9c5b164a
@172.16.254.2:30303",
...
}
```

2. Check the latest block number of the blockchain.

```
curl --location --request POST 'localhost:8545' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 2,
    "method": "eth_blockNumber",
    "params": []
}'
```

After running the nodes for a while, we should see this number greater than 0x0, which means our miner node already created other blocks after the genesis block.

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": "0x3"
}
```

Note: The miner node takes several minutes to initialize before it starts mining. However if the miner has already been running for some time but still get the 0x0 as the latest block number, we should go check the log of the miner node.

3. Next, we will get the address of our primary account created while the image is being built.

```
curl --location --request POST 'localhost:8545' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 3,
    "method": "eth_accounts",
    "params": []
}'
```

The response will contain a list of the created accounts, in our case there should be only 1 account for now.

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": [
    "0x08d1f47128f5c04d7a4aee69e90642645059acd6"
  ]
}
```

Note: The address of the account will be different as it is generated while the image is being built.

4. Check the balance

```
curl --location --request POST 'localhost:8545' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 4,
    "method": "eth_getBalance",
    "params": [
        "0x08d1f47128f5c04d7a4aee69e90642645059acd6",
        "latest"
    ]
}'
```

Note: We send the address of our created account as a parameter to check its balance.

If the latest block number of our private blockchain is greater than 0x0, we should see a non-zero balance in the account. As this first account we created is, by default, the one receives the mining rewards.

```
{
  "jsonrpc": "2.0",
```

```
  "id": 4,
  "result": "0x19b21248a3ef280000"
}
```

Note: The balance is in the hexadecimal number and in the wei unit.

5. To test transferring some funds, we will create another account to be the recipient.

```
curl --location --request POST 'http://localhost:8545' \
--header 'Content-type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 5,
    "method": "personal_newAccount",
    "params": [
        "5uper53cr3t"
    ]
}'
```

Note: To create an account, we have to specify the password as the parameter. This password is required to access the account later. Keep it safe and secret.

If successful, we will get the address of the newly created account in return.

```
{
    "jsonrpc": "2.0",
    "id": 5,
    "result": "0x2bc05c71899ecff51c80952ba8ed444796499118"
}
```

6. Before sending a transaction, we must make sure that the sender account is unlocked.

```
curl --location --request POST 'http://localhost:8545' \
--header 'Content-type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
```

```
    "id": 6,
    "method": "personal_unlockAccount",
    "params": [
        "0x08d1f47128f5c04d7a4aee69e90642645059acd6",
        "5uper53cr3t"
    ]
}'
```

Note: We will use our primary account as a sender as it receives funds from mining rewards. The password used to unlock here must be the same as the one we use to create the account.
If the account is unlocked successfully, it returns true.

```
{
    "jsonrpc": "2.0",
    "id": 6,
    "result": true
}
```

7. Now we are ready to transfer some funds.

```
curl --location --request POST 'localhost:8545' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 7,
    "method": "eth_sendTransaction",
    "params": [
        {
            "from": "0x08d1f47128f5c04d7a4aee69e90642645059acd6",
            "to": "0x2bc05c71899ecff51c80952ba8ed444796499118",
            "value": "0xf4240"
        }
    ]
}'
```

Note: We are sending 0xf4240 wei (equal to 1 ether) from the account, address 0x08d1f47128f5c04d7a4aee69e90642645059acd6 to the account 0x2bc05c71899ecff51c80952ba8ed444796499118.

If the transaction is successfully submitted, we will get the transaction hash in return.

```
{
    "jsonrpc": "2.0",
    "id": 7,
    "result":
"0xa96de080dfcb9c5f908528b92d3df55a0e230cf4e48ae178bb2
20862c2a544c7"
}
```

8. We can get the status of a transaction by its hash.

```
curl --location --request POST 'localhost:8545' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 8,
    "method": "eth_getTransactionByHash",
    "params": [
"0xa96de080dfcb9c5f908528b92d3df55a0e230cf4e48
ae178bb220862c2a544c7"
    ]
}'
```

And the response will look like:

```
{
    "jsonrpc": "2.0",
    "id": 8,
    "result": {
        "blockHash":
"0xf31b1a454f1cd35388476acead342fe93c02667ac99f136a36ef677c462ad04d",
        "blockNumber": "0x16f2",
        "from": "0x08d1f47128f5c04d7a4aee69e90642645059acd6",
        "gas": "0x5208",
```

```
        "gasPrice": "0x3b9aca00",
        "hash":
"0xa96de080dfcb9c5f908528b92d3df55a0e230cf4e48ae178bb220862c2a544c7"
,
        "input": "0x",
        "nonce": "0x0",
        "to": "0x2bc05c71899ecff51c80952ba8ed444796499118",
        "transactionIndex": "0x0",
        "value": "0xf4240",
        "type": "0x0",
        "v": "0x99b",
        "r":
"0xc32053edadd067e93480d76e24fcd03e9879335739c06804916cd292c380cad
1",
        "s":
"0x5f49f5b8c773babff4752abe6662d8f0a1b61e336b051954bd265bf13cd695e9"
    }
}
```


9. Lastly, verify if we get the fund in the recipient's account.

```
curl --location --request POST 'localhost:8545' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "id": 9,
    "method": "eth_getBalance",
    "params": [
        "0x2bc05c71899ecff51c80952ba8ed444796499118",
        "latest"
    ]
}'
```

We expect to see the same value (0xf4240) as we sent from the primary account as the result.


```
{
    "jsonrpc": "2.0",
    "id": 9,
```

```
    "result": "0xf4240"
}
```

Note: Normally, the transaction submitted to the blockchain takes some time to be processed by miners. But for our private network, it should be quick. If we still do not receive the fund after a while, we can go back to recheck the transaction status again.

To automate the setting up of the Private Blockchain:

We can automate the setup of the private blockchain on cloud with the IAC using CFT or Terraform.

Some of the resource which will be required for setting up the private blockchain are:
Considering AWS as cloud service provider

1. VPC
2. Subnets
3. Security Groups
4. IAM Roles
5. EC2 instances with Auto Scaling groups
6. Cloudwatch for monitoring
7. SNS for sending alerts

Once you have the Cloudformation available for all these resources or services we can start setting up the private blockchain manually or through automatic manner using the scripts or even we can have other option i.e while setting up the EC2 instance we can have the commands or script for setting up the private blockchain using user data property of ec2 service.