12
Further Ethereum

This chapter is a continuation of the previous chapter, in which we will examine more Ethereum-based concepts in more detail.

We will cover both a practical and theoretical in-depth introduction to wallet software, mining, and setting up Ethereum nodes. Material relating to various challenges, such as security and scalability faced by Ethereum, will also be introduced. Moreover, prominent advanced supporting protocols, such as Swarm and Whisper, will also be introduced later in the chapter. Finally, Ethereum has several programming languages built in to support smart contract development. We will conclude with an overview of these programming languages.

This chapter continues the discussion of the Ethereum blockchain network elements that we started in the previous chapter, *Chapter 11*, *Ethereum 101*. These elements include:

- Blocks and blockchain
- Wallets and client software
- Nodes and miners
- APIs and tools
- Supporting protocols
- Programming languages

First, we will introduce blocks and blockchain, in continuation of the discussion from the previous chapter.

Blocks and blockchain

Blocks are the main building structure of a blockchain. Ethereum blocks consist of various elements, which are described as follows:

- The block header
- The transactions list
- The list of headers of ommers or uncles

An uncle block is a block that is the child of a parent but does not have any child block. Ommers or uncles are valid, but stale, blocks that are not part of the main chain but contribute to security of the chain. They also earn a reward for their participation but do not become part of the canonical truth.

The transaction list is simply a list of all transactions included in the block. Also, the list of headers of uncles is also included in the block.

Block header: Block headers are the most critical and detailed components of an Ethereum block. The header contains various elements, which are described in detail here:

- **Parent hash**: This is the Keccak 256-bit hash of the parent (previous) block's header.
- **Ommers hash**: This is the Keccak 256-bit hash of the list of ommers (or uncles) blocks included in the block.
- **The beneficiary**: The beneficiary field contains the 160-bit address of the recipient that will receive the mining reward once the block is successfully mined.
- **State root**: The state root field contains the Keccak 256-bit hash of the root node of the state trie. It is calculated once all transactions have been processed and finalized.
- **Transactions root**: The transaction root is the Keccak 256-bit hash of the root node of the transaction trie. The transaction trie represents the list of transactions included in the block.
- **Receipts root**: The receipts root is the Keccak 256-bit hash of the root node of the transaction receipt trie. This trie is composed of receipts of all transactions included in the block. Transaction receipts are generated after each transaction is processed and contain useful post-transaction information. More details on transaction receipts are provided in the next section.
- **Logs bloom**: The logs bloom is a bloom filter that is composed of the logger address and log topics from the log entry of each transaction receipt of the included transaction list in the block. Logging is explained in detail in the next section.
- **Difficulty**: The difficulty level of the current block.
- **Number**: The total number of all previous blocks; the genesis block is block zero.
- **Gas limit**: This field contains the value that represents the limit set on the gas consumption per block.
- **Gas used**: This field contains the total gas consumed by the transactions included in the block.
- **Timestamp**: The timestamp is the epoch Unix time of the time of block initialization.

- **Extra data**: The extra data field can be used to store arbitrary data related to the block. Only up to 32 bytes are allowed in this field.
- **Mixhash**: The mixhash field contains a 256-bit hash that, once combined with the nonce, is used to prove that adequate computational effort (**Proof of Work**, or **PoW**) has been spent in order to create this block.
- **Nonce**: Nonce is a 64-bit hash (a number) that is used to prove, in combination with the *mixhash* field, that adequate computational effort (PoW) has been spent in order to create this block.

The following diagram shows the detailed structure of the block and block header:

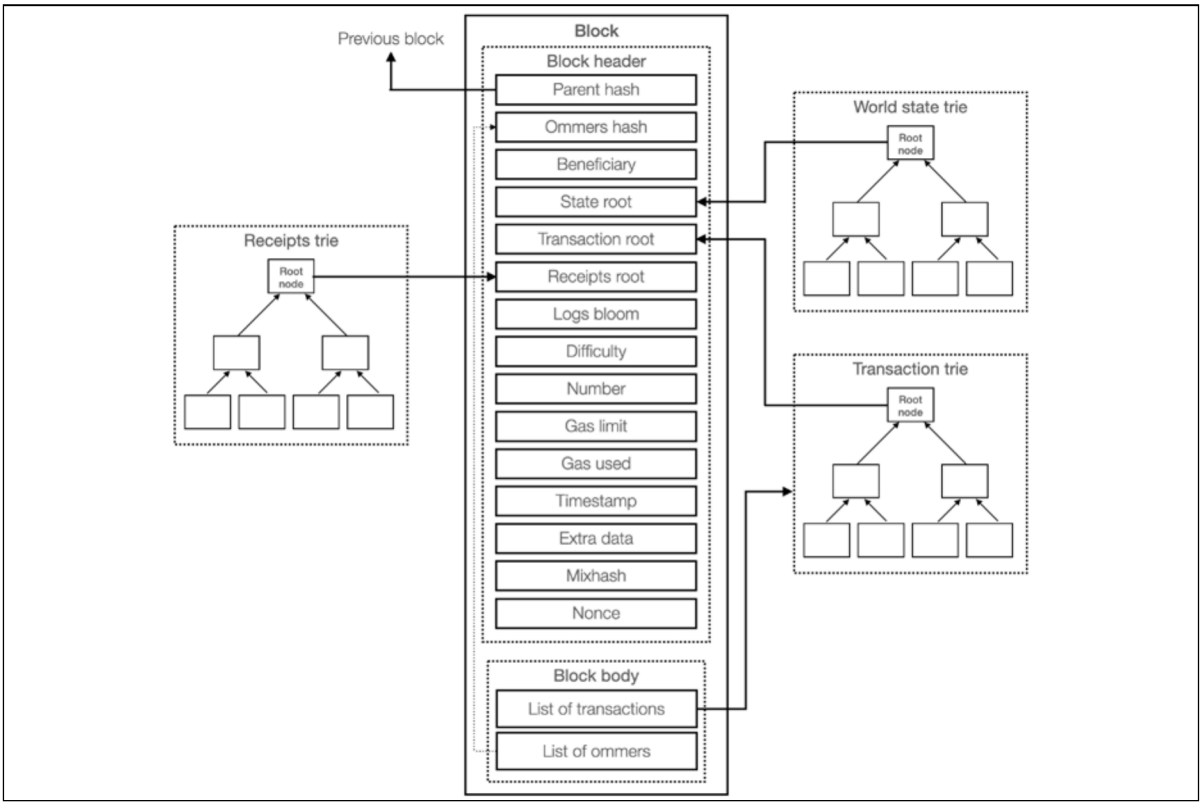


Figure 12.1: A detailed diagram of the block structure with a block header and relationship with tries

The genesis block

The genesis block is the first block in a blockchain network. It varies slightly from normal blocks due to the data it contains and the way it has been created. It contains 15 items that are described here.

From `https://etherscan.io/`, the actual version is shown as follows:

Element	**Description**
Timestamp	(Jul-30-2015 03:26:13 PM +UTC)
Transactions	8893 transactions and 0 contract internal transactions in this block
Hash	0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3
Parent hash	0x0000000000000000000000000000000000000000000000000000000000000000
SHA-3 uncles	0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
Mined by	0x0000000000000000000000000000000000000000 in 15 seconds
Difficulty	17,179,869,184
Total difficulty	17,179,869,184
Size	540 bytes
Gas used	0
Nonce	0x0000000000000042
Block reward	5 ETH
Uncles reward	0
Extra data	In hex (0x11bbe8db4e347b4e8c937c1c8370e4b5ed33adb3db69cbdb7a38e1e50b1b82fa)
Gas limit	5,000

The block validation mechanism

An Ethereum block is considered valid if it passes the following checks:

- If it is consistent with uncles and transactions. This means that all ommers satisfy the property that they are indeed uncles and also if the PoW for uncles is valid.
- If the previous block (parent) exists and is valid.
- If the timestamp of the block is valid. This means that the current block's timestamp must be higher than the parent block's timestamp. Also, it should be less than 15 minutes into the future. All block times are calculated in epoch time (Unix time).
- If any of these checks fails, the block will be rejected. A list of errors for which the block can be rejected is presented here:
    - The timestamp is older than the parent
    - There are too many uncles
    - There is a duplicate uncle
    - The uncle is an ancestor
    - The uncle's parent is not an ancestor
    - There is non-positive difficulty
    - There is an invalid mix digest
    - There is an invalid PoW

Block finalization

Block finalization is a process that is run by miners to validate the contents of the block and apply rewards. It results in four steps being executed. These steps are described here in detail:

1. **Ommers validation**. In the case of mining, determine ommers. The validation process of the headers of stale blocks checks whether the header is valid and whether the relationship between the uncle and the current block satisfies the maximum depth of six blocks. A block can contain a maximum of two uncles.
2. **Transaction validation**. In the case of mining, determine transactions. This process involves checking whether the total gas used in the block is equal to the final gas consumption after the final transaction, in other words, the cumulative gas used by the transactions included in the block.
3. **Reward application**. Apply rewards, which means updating the beneficiary's account with a reward balance. In Ethereum, a reward is also given to miners for stale blocks, which is 1/32 of the block reward. Uncles that are included in the blocks also receive 7/8 of the total block reward. The current block reward is 2 ether. It was reduced first from 5 ether to 3 with the Byzantium release of Ethereum. Later, in the Constantinople release (`https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/`), it was reduced further to 2 ether. A block can have a maximum of two uncles.
4. **State and nonce validation**. Verify the state and block nonce. In the case of mining, compute a valid state and block nonce.

After the high-level view of the block validation mechanism, we now look into how a block is received and processed by a node. We will also see how it is updated in the local blockchain:

1. When an Ethereum full node receives a newly mined block, the header and the body of the block are detached from each other. Now remember, in the last chapter, when we introduced the fact that there are three **Merkle Patricia tries** (**MPTs**) in an Ethereum blockchain. The roots of those MPTs or tries are present in each block header as a state trie root node, a transaction trie root node, and a receipt trie root node. We will now learn how these tries are used to validate the blocks.
2. A new MPT is constructed that comprises all transactions from the block.
3. All transactions from this new MPT are executed one by one in a sequence. This execution occurs locally on the node within the **Ethereum Virtual Machine** (**EVM**). As a result of this execution, new transaction receipts are generated that are organized in a new receipts MPT. Also, the global state is modified accordingly, which updates the state MPT (trie).
4. The root nodes of each respective trie, in other words, the state root, transaction root, and receipts root are compared with the header of the block that was split in the first step. If both the roots of the newly constructed tries and the trie roots that already exist in the header are equal, then the block is verified and valid.
5. Once the block is validated, new transaction, receipt, and state tries are written into the local blockchain database.

Block difficulty mechanism

Block difficulty is increased if the time between two blocks decreases, whereas it increases if the block time between two blocks decreases. This is required to maintain a roughly consistent block generation time. The difficulty adjustment algorithm in Ethereum's **Homestead** release is as follows:

$$block\_diff = parent\_diff + parent\_diff // 2048 *$$

$$max(1 - (block\_timestamp - parent\_timestamp) // 10, -99) +$$

$$int(2**((block.number // 100000) - 2))$$

Note that // is the integer division operator.

The preceding algorithm means that, if the time difference between the generation of the parent block and the current block is less than 10 seconds, the difficulty goes up by *parent_diff // 2048 * 1*. If the time difference is between 10 and 19 seconds, the difficulty level remains the same. Finally, if the time difference is 20 seconds or more, the difficulty level decreases. This decrease is proportional to the time difference from *parent_diff // 2048 * -1* to a maximum decrease of *parent_diff // 2048 * -99*.

In addition to timestamp difference-based difficulty adjustment, there is also another part (shown in the last line of the preceding algorithm) that increases the difficulty exponentially after every 100,000 blocks. This is the so-called **difficulty time bomb**, or **ice age**, introduced in the Ethereum network, which will make it very hard to mine on the Ethereum blockchain at some point in the future. This will encourage users to switch to **Proof of Stake** (**PoS**), since mining on the PoW chain will eventually become prohibitively difficult.

The difficulty time bomb was delayed via EIP-649 (`https://github.com/ethereum/EIPs/pull/669`) for around 18 months and no clear time frame has yet been suggested.

According to the original estimates based on the algorithm, the block generation time would have become significantly higher during the second half of 2017, and in 2021, it would become so high that it would be virtually impossible to mine on the PoW chain, even for dedicated mining centers. This way, miners will have no choice but to switch to the PoS scheme proposed by Ethereum, called **Casper**.

More information about Casper is available here: `https://github.com/ethereum/research/blob/master/papers/casper-basics/casper_basics.pdf`.

This ice age proposal has been postponed with the release of **Byzantium**. Instead, the mining reward has been reduced from 5 ETH to 3 ETH, in preparation for PoS implementation in Serenity.

In the Byzantium release, the difficulty adjustment formula has been changed to take uncles into account for difficulty calculation. This new formula is shown here:

adj_factor = max((2 if len(parent.uncles) else 1) - ((timestamp - parent.timestamp) // 9), -99)

Soon after the **Istanbul** upgrade, the difficulty bomb was delayed once again, under the **Muir Glacier** network upgrade with a hard fork, `https://eips.ethereum.org/EIPS/eip-2384`, for roughly another 611 days. The change was activated at block number 9,200,000 on January 2, 2020.

We know that blocks are composed of transactions and that there are various operations associated with transaction creation, validation, and finalization. Each of these operations on the Ethereum network costs some amount of ETH and is charged using a fee mechanism. This fee is also called gas. We will now introduce this mechanism in detail.

Gas

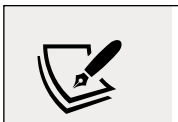
Gas is required to be paid for every operation performed on the Ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the whole blockchain to stall due to the Turing-complete nature of the EVM. A transaction fee is charged as an amount of Ether and is taken from the account balance of the transaction originator.

A fee is paid for transactions to be included by miners for mining. If this fee is too low, the transaction may never be picked up; the more the fee, the higher are the chances that the transactions will be picked up by the miners for inclusion in the block. Conversely, if the transaction that has an appropriate fee paid is included in the block by miners but has too many complex operations to perform, it can result in an out-of-gas exception if the gas cost is not enough. In this case, the transaction will fail but will still be made part of the block, and the transaction originator will not get any refund.

Transaction costs can be estimated using the following formula:

*Total cost = gasUsed * gasPrice*

Here, *gasUsed* is the total gas that is supposed to be used by the transaction during the execution, and *gasPrice* is specified by the transaction originator as an incentive to the miners to include the transaction in the next block. This is specified in ETH. Each EVM opcode has a fee assigned to it. It is an estimate because the gas used can be more or less than the value specified by the transaction originator originally. For example, if computation takes too long or the behavior of the smart contract changes in response to some other factors, then the transaction execution may perform more or fewer operations than intended initially and can result in consuming more or less gas. If the execution runs out of gas, everything is immediately rolled back; otherwise, if the execution is successful and some gas remains, then it is returned to the transaction originator.



A website that keeps track of the latest gas price and provides other valuable statistics and calculators is available at `https://ethgasstation.info/index.php`.

Each operation costs some gas; a high-level fee schedule of a few operations is shown as an example here:

Operation name	Gas cost
Stop	0
SHA3	30
SLOAD	800
Transaction	21000
Contract creation	32000

Based on the preceding fee schedule and the formula discussed earlier, an example calculation of the SHA-3 operation can be calculated as follows:

- SHA-3 costs 30 gas.
- Assume that the current gas price is 25 GWei, and convert it into ETH, which is 0.000000025 ETH. After multiplying both, 0.000000025 * 30, we get 0.00000075 ETH.
- In total, 0.00000075 ETH is the total gas that will be charged.

Fee schedule

Gas is charged in three scenarios as a prerequisite to the execution of an operation:

- The computation of an operation
- For contract creation or message calls
- An increase in the use of memory

A list of instructions and various operations with the gas values has been provided in the previous section.

Now let's move onto another topic, clients and wallets, which are responsible for mining, payments, and management functions, such as account creation on an Ethereum network.

Wallets and client software

As Ethereum is under heavy development and evolution, there are many components, clients, and tools that have been developed and introduced over the last few years.

Wallets

A wallet is a generic program that can store private keys and, based on the addresses stored within it, it can compute the existing balance of ether associated with the addresses by querying the blockchain. It can also be used to deploy smart contracts. In this chapter, we will introduce the **MetaMask** wallet, which has become the tool of choice for developers.

Having discussed the role of wallets within Ethereum, let's now discuss a number of common clients. The following is a non-exhaustive list of the client software and wallets that are available with Ethereum.

Geth

This is the official **Go** implementation of the Ethereum client.

The latest version is available at the following link: `https://geth.ethereum.org/downloads/`.

Eth

This is the **C++** implementation of the Ethereum client.

Eth is available at the GitHub repository: `https://github.com/ethereum/aleth`.

Parity

This implementation is built using **Rust** and developed by **Parity** technologies.

Parity can be downloaded from the following link: `https://www.parity.io/`

Note that Parity is now OpenEthereum. Henceforth, we will use the name OpenEthereum.

Trinity

Trinity is the implementation of the Ethereum protocol. It is written in **Python**.

Trinity can be downloaded from `https://github.com/ethereum/trinity`. The official website of Trinity client is `https://trinity.ethereum.org`.

Light clients

Simple Payment Verification (**SPV**) clients download only a small subset of the blockchain. This allows low resource devices, such as mobile phones, embedded devices, or tablets, to be able to verify the transactions.

A complete Ethereum blockchain and node are not required in this case, and SPV clients can still validate the execution of transactions. SPV clients are also called light clients. This idea is similar to Bitcoin SPV clients.

There is a wallet available from Jaxx (`https://jaxx.io/`), which can be installed on iOS and Android, which provides the SPV functionality.

The critical difference between clients and wallets is that clients are full implementations of the Ethereum protocol, which support mining, account management, and wallet functions. In contrast, wallets only store the public and private keys, provide essential account management, and interact with the blockchain for usually only payment (transfer of funds) purposes.

Next, we discuss the installation procedure and usage of some of these clients.

Installation and usage

First, we describe Geth and explore various operations that can be performed using this client.

Geth

The following installation procedure describes the installation of Ethereum clients on macOS and Linux.

Instructions for other operating systems are available on the official Ethereum documentation site at `https://geth.ethereum.org/docs/install-and-build/installing-geth`.

The Geth client can be installed by using the following command on an Ubuntu system:

```
$ 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
```

On a macOS using homebrew, Geth can be installed by executing the following commands:

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

If an older version of Ethereum `geth` client is already installed, it can be upgraded by issuing the following command:

```
$ brew upgrade ethereum
```

Once installation is complete, Geth can be launched simply by issuing the `geth` command at the terminal. It comes preconfigured with all the required parameters to connect to the live Ethereum network (mainnet):

```
$ geth
```

This will produce output similar to the following:

```
INFO [01-18|21:00:06.590] Bumping default cache on mainnet         provided=1024 updated=4096
WARN [01-18|21:00:06.590] Sanitizing cache to Go's GC limits       provided=4096 updated=1314
INFO [01-18|21:00:06.592] Maximum peer count                       ETH=50 LES=0 total=50
INFO [01-18|21:00:06.595] Starting peer-to-peer node               instance=Geth/v1.9.9-stable-01744997/linux-amd64/go1.13.4
INFO [01-18|21:00:06.597] Allocated trie memory caches             clean=328.00MiB dirty=328.00MiB
INFO [01-18|21:00:06.598] Allocated cache and file handles         database=/home/vagrant/.ethereum/geth/chaindata cache=657.00MiB handles=524288
INFO [01-18|21:00:06.749] Opened ancient database                  database=/home/vagrant/.ethereum/geth/chaindata/ancient
INFO [01-18|21:00:06.793] Initialised chain configuration          config="{ChainID: 1 Homestead: 1150000 DAO: 1920000 DAOSupport: true EIP150: 2463000 EIP155
: 2675000 EIP158: 2675000 Byzantium: 4370000 Constantinople: 7280000 Petersburg: 7280000 Istanbul: 9069000, Muir Glacier: 9200000, Engine: ethash}"
INFO [01-18|21:00:06.794] Disk storage enabled for ethash caches   dir=/home/vagrant/.ethereum/geth/ethash count=3
INFO [01-18|21:00:06.794] Disk storage enabled for ethash DAGs     dir=/home/vagrant/.ethash          count=2
INFO [01-18|21:00:06.795] Initialising Ethereum protocol           versions="[64 63]" network=1 dbversion=7
INFO [01-18|21:00:06.801] Loaded most recent local header          number=2496 hash=2f3bf5…246460 td=83203260865481 age=4y6mo1w
INFO [01-18|21:00:06.803] Loaded most recent local full block      number=0    hash=d4e567…cb8fa3 td=17179869184    age=50y9mo1w
INFO [01-18|21:00:06.804] Loaded most recent local fast block      number=1541 hash=90afd6…ca6647 td=39237270173210 age=4y6mo1w
INFO [01-18|21:00:06.806] Loaded local transaction journal         transactions=0 dropped=0
INFO [01-18|21:00:06.808] Regenerated local transaction journal    transactions=0 accounts=0
INFO [01-18|21:00:06.839] Allocated fast sync bloom                size=656.00MiB
INFO [01-18|21:00:07.154] New local node record                    seq=7 id=bd0df49fbe92d6a9 ip=127.0.0.1 udp=30303 tcp=30303
INFO [01-18|21:00:07.171] Started P2P networking                   self=enode://8d8a401311d88e7016edda12fa73a0b8ea8e563c0272e66ba1a3de055bfb4ec5a61bae6c054802
ca9c3a3314ec7f81155820ea99d8985989b51ee1426260dd17@127.0.0.1:30303
INFO [01-18|21:00:07.173] IPC endpoint opened                      url=/home/vagrant/.ethereum/geth.ipc
INFO [01-18|21:00:07.962] Initialized fast sync bloom              items=13046 errorrate=0.000 elapsed=1.120s
WARN [01-18|21:00:11.009] Dropping unsynced node during fast sync  id=04471e22004d2e5d conn=dyndial addr=211.228.238.136:30303 type=Geth/v1.9.3-stable-cfbb969
d/linux-amd64/go1.11.5
INFO [01-18|21:00:11.633] New local node record                    seq=8 id=bd0df49fbe92d6a9 ip=127.0.0.1 udp=30303 tcp=30303
INFO [01-18|21:00:13.137] New local node record                    seq=9 id=bd0df49fbe92d6a9 ip=127.0.0.1 udp=30303 tcp=30303
INFO [01-18|21:00:15.371] New local node record                    seq=10 id=bd0df49fbe92d6a9 ip=82.2.27.41 udp=54057 tcp=30303
INFO [01-18|21:00:27.175] Block synchronisation started
INFO [01-18|21:00:40.927] Imported new block headers               count=0 elapsed=8.796ms number=1733 hash=3ddbe0…378d04 age=4y6mo1w  ignored=192
INFO [01-18|21:00:41.391] Imported new block headers               count=0 elapsed=11.186ms number=1925 hash=052de6…33e43d age=4y6mo1w  ignored=192
INFO [01-18|21:00:41.403] Imported new block receipts              count=7 elapsed=29.573ms number=1548 hash=3e6adc…4a9bb5 age=4y6mo1w  size=5.54KiB
```

Figure 12.2: Geth

When the Ethereum client starts up, it starts to synchronize with the rest of the network. There are three types of synchronization mechanisms available, namely, full, fast, and light.

- **Full**: In this synchronization mode, the Geth client downloads the complete blockchain to its local node. This means that it gets all the block headers and block bodies and validates all transaction and blocks since the genesis block. Currently (as at early 2020), the Ethereum blockchain size is roughly 210 GB, and downloading and maintaining that could be a problem. Usually, SSDs are recommended for a full node, so that disk latency cannot cause any processing delays.
- **Fast**: The client downloads the full blockchain, but it retrieves and verifies only the previous 64 blocks from the current block. After this, it verifies the new blocks in full. It does not replay and verify all historic transactions since the genesis block; instead it only does the state downloads. This also reduces the on-disc size of the blockchain database quite significantly. This is the default mode of Geth client synchronization.

- **Light**: This is the quickest mode and only downloads and stores the current state trie. In this mode, the client does not download any historic blocks and only processes newer blocks.

Synchronization mode is configurable on the `geth` client via the flag:

```
--syncmode value
```

Here, the value can be either `fast`, `full`, or `light`.

Ethereum account management using Geth

New accounts can be created via the command line using Geth or OpenEthereum (formerly Parity Ethereum) command-line interface. This process is shown in the next section for `geth`, which we installed in the previous *Installation and usage* section.

Creating a Geth new account

Execute the following command to add a new account:

```
$ geth account new
```

This command will produce output similar to the following:

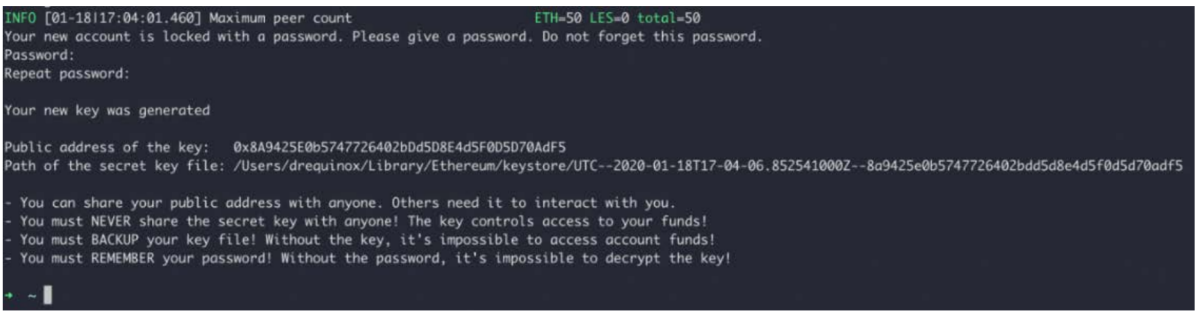


Figure 12.3: Geth account generation

The list of accounts can be displayed using the `geth` client by issuing the following command:

```
$ geth account list
```

This command will produce output similar to the following:

```
INFO [01-18|17:59:25.595] Maximum peer count        ETH=50 LES=0 total=50
Account #0: {07668e548be1e17f3dcfa2c4263a0f5f88aca402}
keystore:///home/vagrant/.ethereum/keystore/UTC--2020-01-18T17-33-
32.099124768Z--07668e548be1e17f3dcfa2c4263a0f5f88aca402
Account #1: {ba94fb1f306e4d53587fcdcd7eab8109a2e183c4}
keystore:///home/vagrant/.ethereum/keystore/UTC--2020-01-18T17-46-
46.604174215Z--ba94fb1f306e4d53587fcdcd7eab8109a2e183c4
```

```
Account #2: {1df5ae40636d6a1a4f8db8a0c65addce5a992a14}
keystore:///home/vagrant/.ethereum/keystore/UTC--2020-01-18T17-56-
12.517958461Z--1df5ae40636d6a1a4f8db8a0c65addce5a992a14
```

Note that you will see different addresses and directory paths when you run this on your computer.

In this example, we saw how to create a new account and obtain a list of accounts. Next, we'll see different methods to interact with the blockchain.

How to query the blockchain using Geth

There are different methods available to query with the blockchain. First, in order to connect to the running instance of the client, either a local IPC or RPC API can be used.

There are three methods of interacting with the blockchain using Geth:

1. Geth console
2. Geth attach
3. Geth JSON RPC

Geth console and Geth attach are used to interact with the blockchain using an REPL JavaScript environment. Geth JSON RPC will be discussed in the *APIs, tools, and DApps* section.

Geth console

Geth can be started in console mode by running the following command:

```
$ geth console
```

This will also start the interactive JavaScript environment in which JavaScript commands can be run to interact with the Ethereum blockchain.

Geth attach

When a `geth` client is already running, the interactive JavaScript console can be invoked by attaching to that instance. This is possible by running the `geth attach` command.

The Geth JavaScript console can be used to perform various functions. For example, an account can be created by attaching Geth.

Geth can be attached with the running daemon, as shown in the following screenshot:

```
$ geth attach
```

This command will produce output similar to the following:

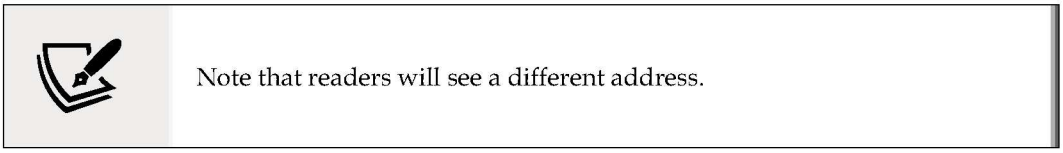
```
Welcome to the Geth JavaScript console!

instance: Geth/v1.9.9-stable-01744997/linux-amd64/go1.13.4
coinbase: 0x07668e548be1e17f3dcfa2c4263a0f5f88aca402
at block: 0 (Thu, 01 Jan 1970 00:00:00 UTC)
 datadir: /home/vagrant/.ethereum
 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

>
```

Figure 12.4: Geth client

Once Geth is successfully attached with the running instance of the Ethereum client, it will display the command prompt, >, which provides an interactive command-line interface to interact with the Ethereum client using JavaScript notations.

For example, a new account can be added using the following command in the Geth console:

```
> personal.newAccount()
Password:
Repeat password:
"0x76bcb051e3cedfcc9c7ff0b25a57383dacbf833b"
```



Note that readers will see a different address.

The list of accounts can also be displayed similarly:

```
> eth.accounts
["0x07668e548be1e17f3dcfa2c4263a0f5f88aca402",
"0xba94fb1f306e4d53587fcdcd7eab8109a2e183c4",
"0x1df5ae40636d6a1a4f8db8a0c65addce5a992a14",
"0x76bcb051e3cedfcc9c7ff0b25a57383dacbf833b"]
```

Now that we understand how accounts are created using Geth, a question arises: when a new account is created, where does Geth store the related public and private key information?

The answer to this question is that when an account in Ethereum is created, it stores the generated public and private key pair on disk in a **keystore**. It is important to understand what this keystore contains and where it is created. We explain this in the next section.

Ethereum keystore

As we saw in the previous chapter, accounts are represented by private keys. The public and private key pair generated by the new account creation process is stored in key files located locally on the disk. These key files are stored in the `keystore` directory present in the relevant path according to the OS in use.

On the Linux OS, its location is as follows: `~/.ethereum/keystore`

The content of a `keystore` file is shown here as an example. Can you correlate some of the names with what we have already learned in *Chapter 3*, *Symmetric Cryptography*, and *Chapter 4*, *Public Key Cryptography*? It is a JSON file. In the following example, it has been formatted for better visibility:

```
{
"address":"ba94fb1f306e4d53587fcdcd7eab8109a2e183c4",
"crypto":
{
    "cipher":"aes-128-ctr",
    "ciphertext":
        "b2ab4f94f5f44ce98e61d99641cd28eb00fd794129be25beb8a5fae89ef93241",
    "cipherparams":
        {"iv":"a0fdfe0a6d314a62ba6a370f438faa57"},
    "kdf":"scrypt",
    "kdfparams":
        {"dklen":32,"n":262144,"p":1,"r":8,
"salt":"be3e99203c24ffcb71a6be2823fc7a211c8cc10d66bc6b448fef420fa0669068"},
"mac":"1b0a42d4bf7a8e96d308179e9714718e902727ead7041b97a646ef1c9d6f9ad7"},
"id":"7e0772e0-965e-4a05-ad93-fc5d11245ba3",
"version":3
}
```

The private key is stored in an encrypted format in the `keystore` file. It is generated when a new account is created using the password and private key. The `keystore` file is also referred to as a UTC file as the naming format of it starts with UTC with the date timestamp therein. A sample name of the `keystore` file is shown here:

```
UTC--2020-01-18T17-46-46.604174215Z--ba94fb1f306e4d53587fcdcd7eab8109a2e183c4
```

On macOS, the `keystore` file is stored at the following location: `~/Library/Ethereum/keystore/`

It is imperative to keep safe the associated passwords created at the time of creating the accounts and when the key files are produced.

As the `keystore` file is present on the disk, it is very important to keep it safe. It is recommended that it is backed up as well. If the `keystore` files are lost or overwritten or somehow corrupted, there is no way to recover them. This means that any ether associated with the private key will be irrecoverable, too.

Now we have described the elements of the password `keystore` file and what they represent. The key purpose of this file is to store the configuration, which, when provided with the account password, generated a decrypted account private key. This private key is then used to sign transactions.

- `Address`: This is the public address of the account that is used to identify the sender or receiver.
- `Crypto`: This field contains the cryptography parameters.
- `Cipher`: This is the cipher used to encrypt the private key. In the following diagram, **AES-128-CTR** indicates the Advanced Encryption Standard – 128 bit – in counter mode. Remember that we covered this in *Chapter 3*, *Symmetric Key Cryptography*.
- `Ciphertext`: This is the encrypted private key.
- `Cipherparams`: This represents the parameters required for the encryption algorithm, **AES-128-CTR**.
- `IV`: This is the 128-bit initialization vector for the encryption algorithm.
- `KDF`: This is the key derivation function. It is `scrypt` in this case.
- `KDFParams`: These are the parameters for the **key derivation function** (**KDF**).
- `Dklen`: This is the derived key length. It is 32 in our example.
- `N`: This is the iteration count.
- `P`: This is the parallelization factor; the default value is 1.
- `R`: This is the block size of the underlying hash function. It is set to 8, which is the default setting.
- `Salt`: This is the random value of salt for the key derivation function, KDF.
- `Mac`: This is the Keccak-256 hash output obtained following concatenation of the second leftmost 16 bytes of the derived key together with the ciphertext.
- `ID`: This is a random identification number.
- `Version`: The version number of the file format, currently version 3.

Now we will explore how all these elements work together. This whole process can be visualized in the following diagram:

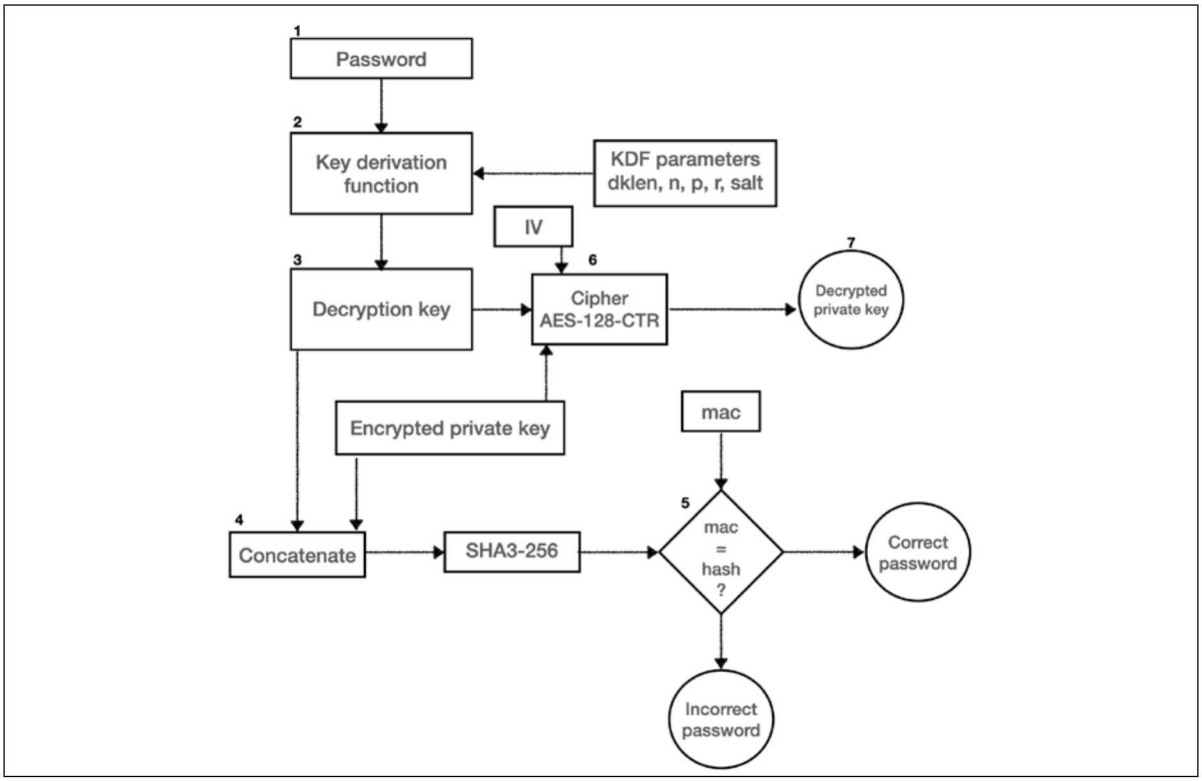


Figure 12.5: Private key decryption process

As shown in the preceding diagram, we can divide the process into different steps:

1. First, the password is fed into the **KDF**.
2. KDF takes several parameters, namely, **dklen**, **n**, **p**, **r**, and **salt**, and produces a decryption key.
3. The decryption key is concatenated with an **encrypted private key** (ciphertext). The decryption key is also fed into the cipher algorithm, **AES-128-CTR**.
4. The concatenated **decryption key** and ciphertext are hashed using the **SHA3-256** hash function.
5. **mac** is fed into the checking function where the hash produced from step 4 is compared with the **mac**. If both match, then the password is valid, otherwise the password is incorrect.
6. The cipher function, which is fed with the initialization vector (**IV**), encrypted private key (ciphertext), and decryption key, decrypts the encrypted private key.
7. The **decrypted private key** is produced.

This decrypted private key is then used to sign transactions on the Ethereum network.

With this, we have completed the introduction to Geth. Next, we will briefly touch on Eth and OpenEthereum before looking at MetaMask in detail. We'll start with installation.

Eth installation

Eth is the C++ implementation of the Ethereum client and can be installed using the following command on Ubuntu:

```
$ sudo apt-get install cpp-ethereum
```

OpenEthereum installation

OpenEthereum is another implementation of the Ethereum protocol. It has been written using the **Rust** programming language. The key aims behind the development of OpenEthereum are high performance, a small footprint, modularization, and reliability.

It can be downloaded from Open Ethereum GitHub at `https://github.com/openethereum/openethereum`.

OpenEthereum is the rebranded version of Parity Ethereum, which was a famous Ethereum client developed by Parity technologies. Parity Ethereum has been transitioned into a DAO-based ownership and maintainer model that is expected to provide a decentralized platform for collaboration, governance, and further development of the Parity Ethereum code base.

More information on this transition is available here: `https://www.parity.io/parity-ethereum-openethereum-dao/`.

For all major operating systems, the latest release of the software is available here: `https://github.com/openethereum/openethereum/releases/latest`.

OpenEthereum can be installed using the `brew` package manager on a macOS system:

```
$ brew tap paritytech/paritytech
$ brew install parity
```

If Parity is already installed, it can be upgraded using the following command.

```
$ brew upgrade parity
```

If you are on an OS other than macOS, the software for that OS can be downloaded from the preceding GitHub link.

Once installation has completed successfully, Parity can be run by using the following command:

```
$ parity
```

This command will produce output similar to the following:

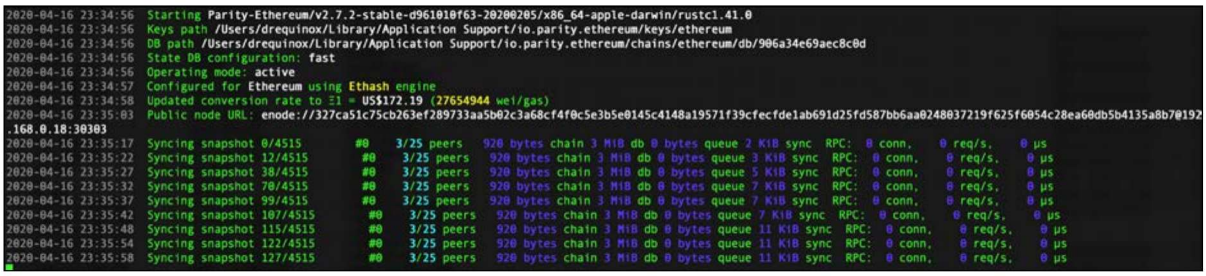


Figure 12.6: Parity startup

Creating accounts using the Parity command line

The following command can be used to create a new account using Parity:

```
$ parity account new
Please note that password is NOT RECOVERABLE.
Type password:
Repeat password:
0x398aa52c557a6deed04d0e5dd486bc40dd374d00
```

Now we move to a different type of software used for interaction with the Ethereum blockchain: wallets.

Various wallets are available for Ethereum for desktop, mobile, and web platforms. A popular wallet that we will use for examples in the next chapter is named **MetaMask**, which is a tool of choice when it comes to development for Ethereum.

MetaMask

MetaMask runs as a plugin or add-on in the web browser. It is available for the Chrome, Firefox, Opera, and Brave browsers. The key idea behind the development of MetaMask is to provide an interface with the Ethereum blockchain. It allows efficient account management and connectivity to the Ethereum blockchain without running the Ethereum node software locally. MetaMask allows connectivity to the Ethereum blockchain through the infrastructure available at Infura (`https://infura.io`). This allows users to interact with the blockchain without having to host any node locally.

Installation

MetaMask is available for download at `https://metamask.io`. Here, we'll go over the installation process.

Browse to `https://metamask.io/`, where links to the relevant source are available to download the extension for your browser:

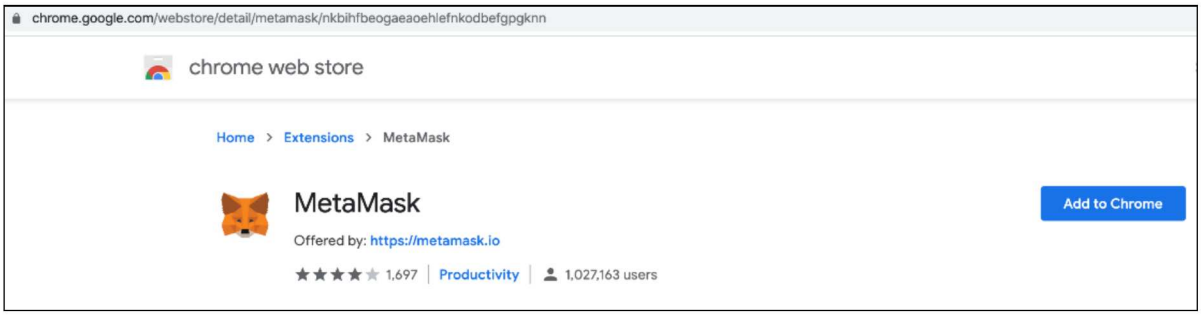


Figure 12.7: MetaMask download

The preceding image shows the MetaMask plugin on Chrome Web Store.

1. Click the **Add to Chrome** button and it will install the extension for the browser. It will install quickly, and if all goes well, you will see the following window:

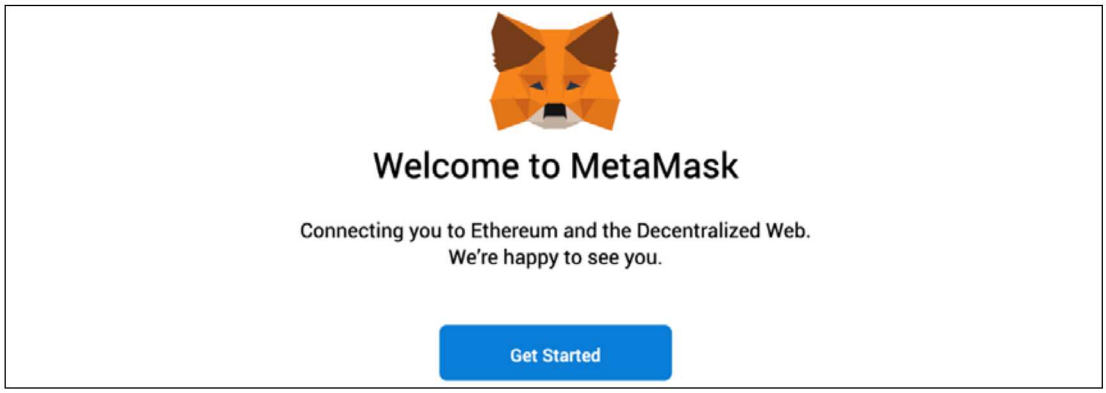


Figure 12.8: MetaMask welcome

2. Click on **Get Started**, and it will show two options, either to import an already existing wallet using the seed, or to create a new wallet. We will select **Create a Wallet** here:

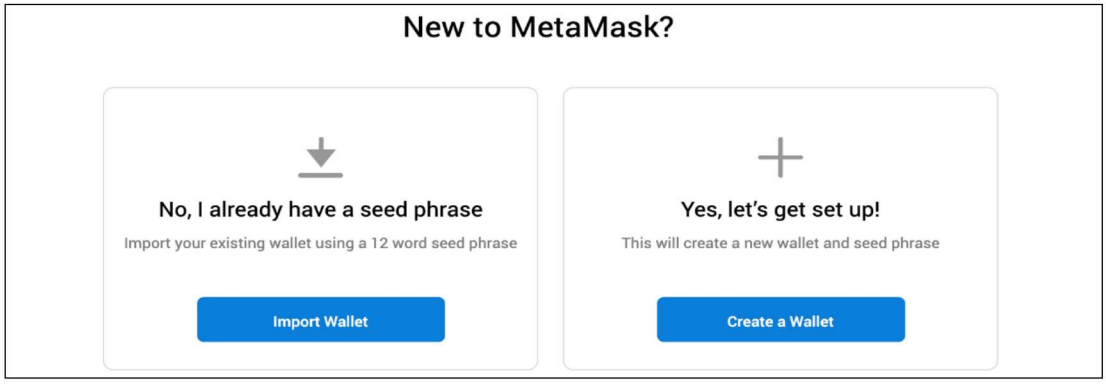


Figure 12.9: MetaMask wallet management

3. Next, create a **Password**:

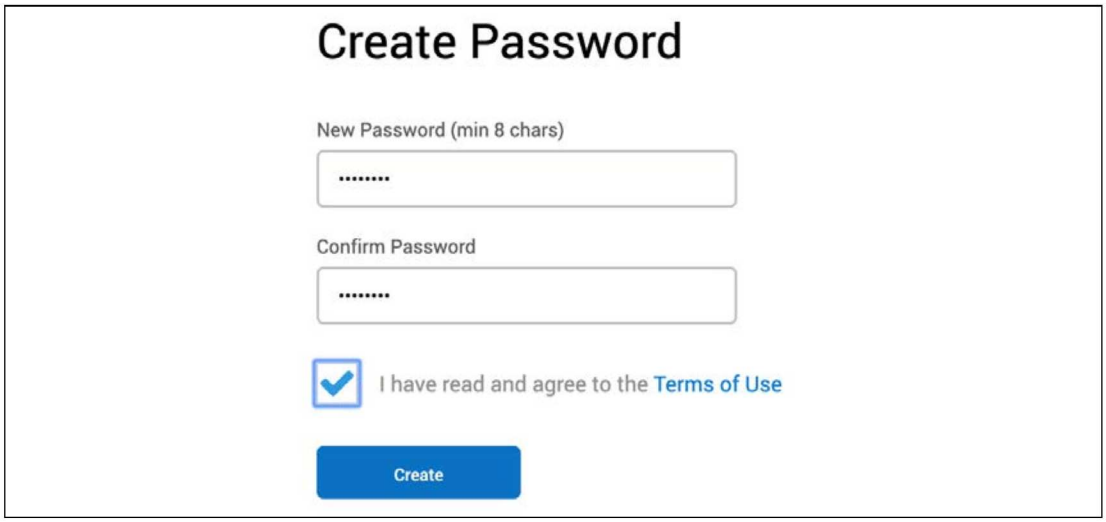


Figure 12.10: Create password

4. Optionally, set up a **Secret Backup Phrase**:

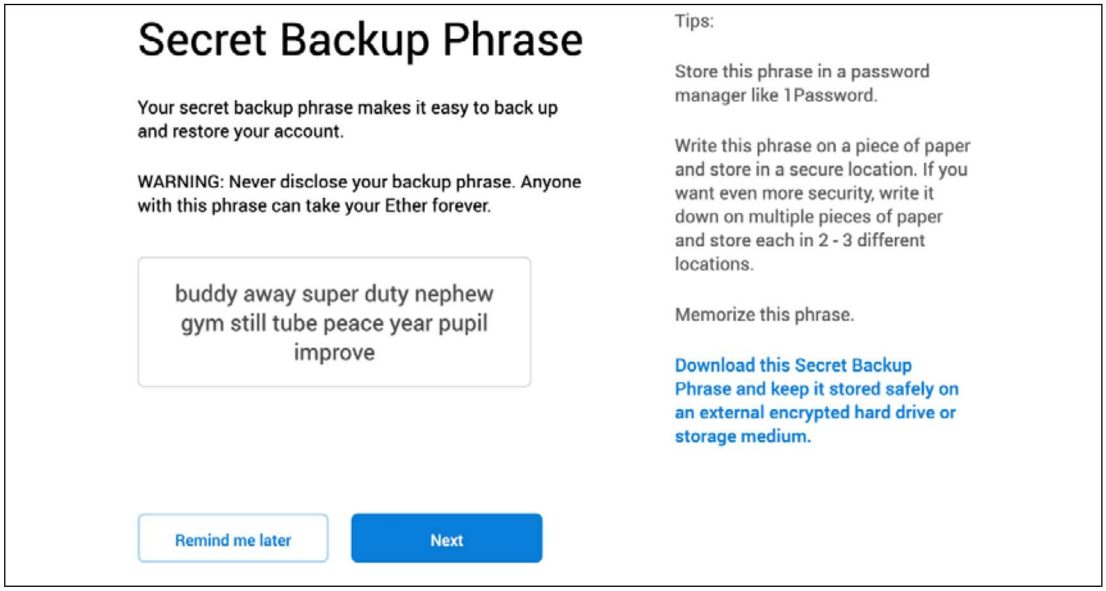


Figure 12.11: Secret backup phrase

5. Once everything has been completed, you will see the following message:

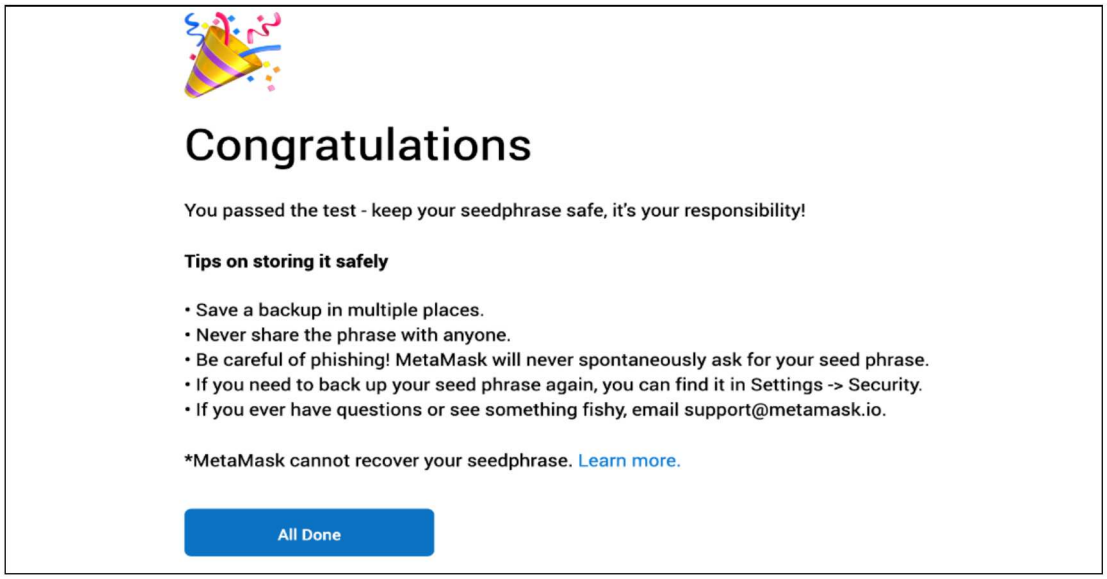


Figure 12.12: Congratulations in MetaMask

6. After clicking on the **All Done** button, the main MetaMask main view will open:

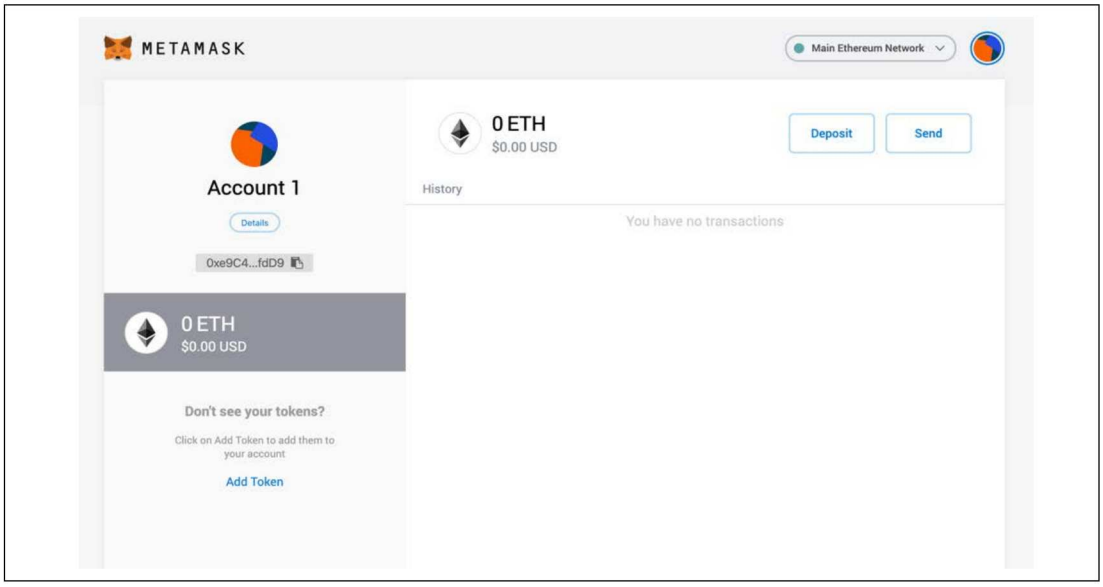


Figure 12.13: MetaMask main view

7. Also, in the top right-hand corner, you will see a small fox icon. Click that and you will see the following view:

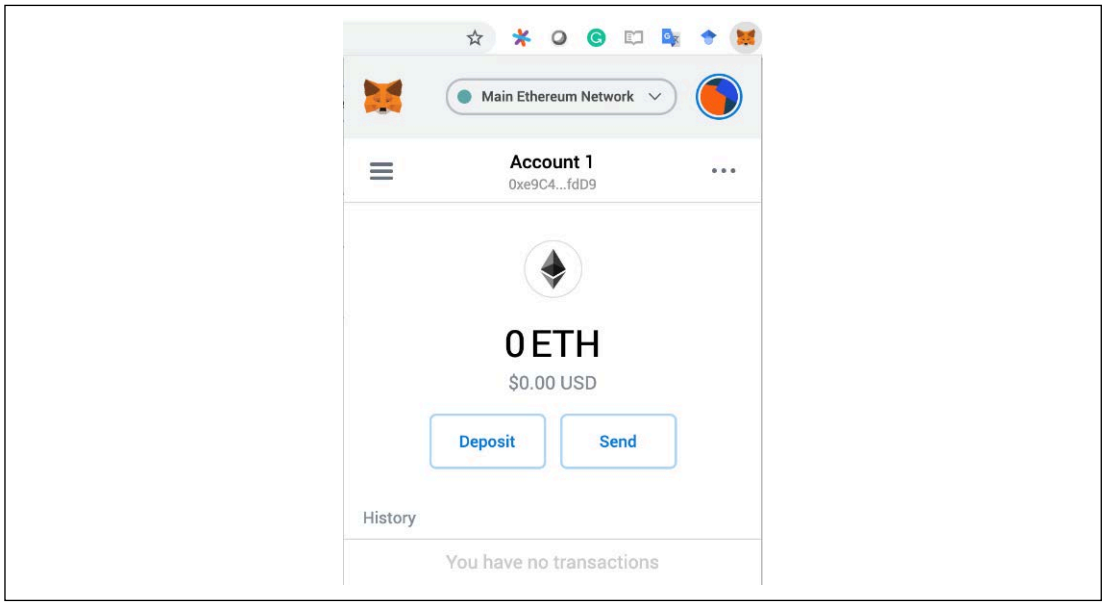


Figure 12.14: MetaMask view

Once installed, the Ethereum provider API will be available in the browser context, which can be used to interact with the blockchain. MetaMask injects a global API into websites at `window.ethereum`.

An example of the Ethereum API is shown here using the JavaScript console in Chrome:

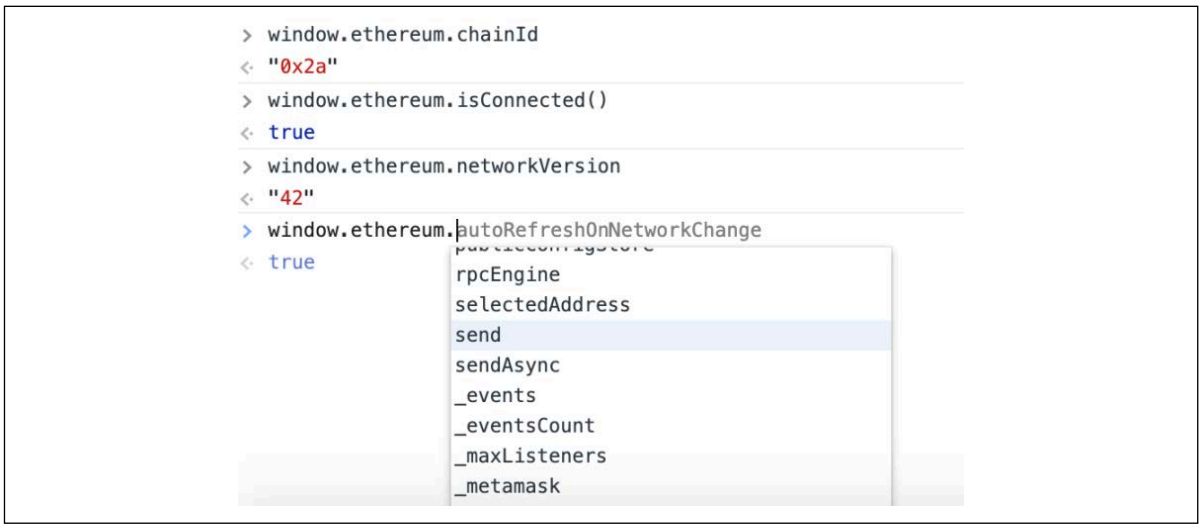


Figure 12.15: Ethereum object

MetaMask can connect to various networks. By default, it connects to the Ethereum mainnet.

Other included networks include, but are not limited to, the **Ropsten** test network, the **Kovan** test network, the **Rinkeby** test network and the **Goerli** test network.

In addition, it can connect to various testnets and local nodes via `Localhost 8545`, and to *Custom RPC*, which means that it can connect to any network as long as RPC connection information is available.

Now, let's see how account management works in MetaMask.

Creating and funding an account using MetaMask

Let's now create an account and fund it with some ETH, all using MetaMask. Note that we are connected to the Kovan test network. Open MetaMask and ensure that it is connected to the Kovan test network, as shown here:

1. Click on **Create Account**:

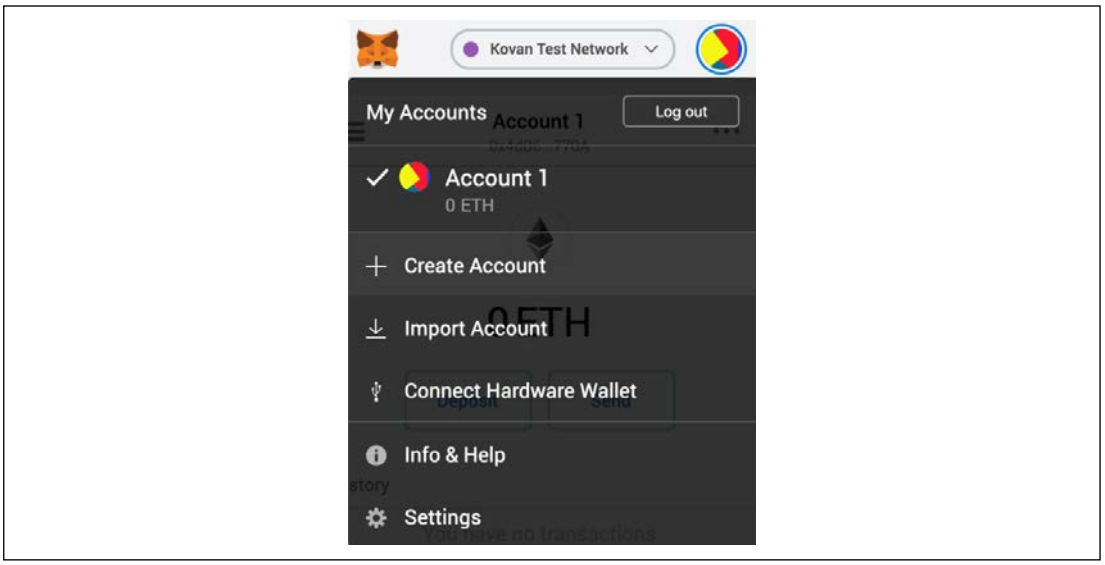


Figure 12.16: Creating an account

2. Enter the new account name and click on **Create**, which will immediately create a new account:

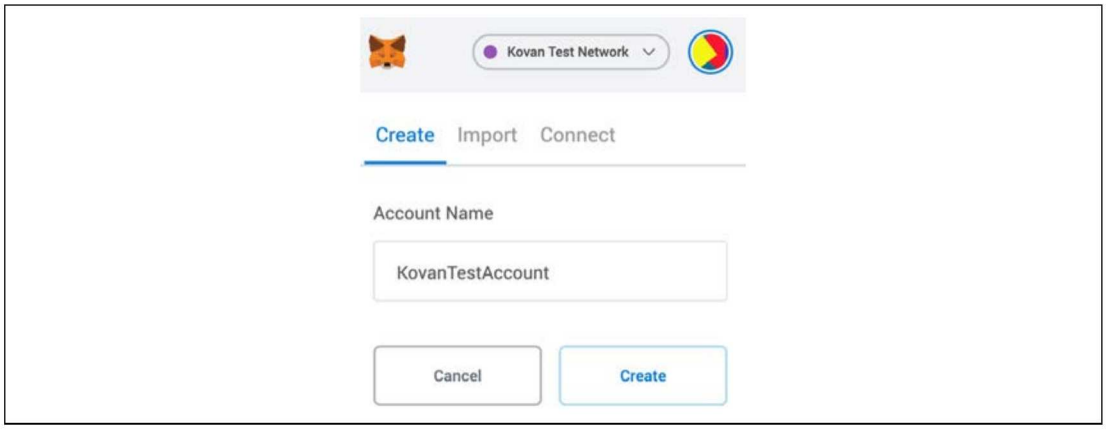


Figure 12.17: Creating a Kovan test account

Once created, we can fund the account with the following steps:

3. Copy the account address to the clipboard:

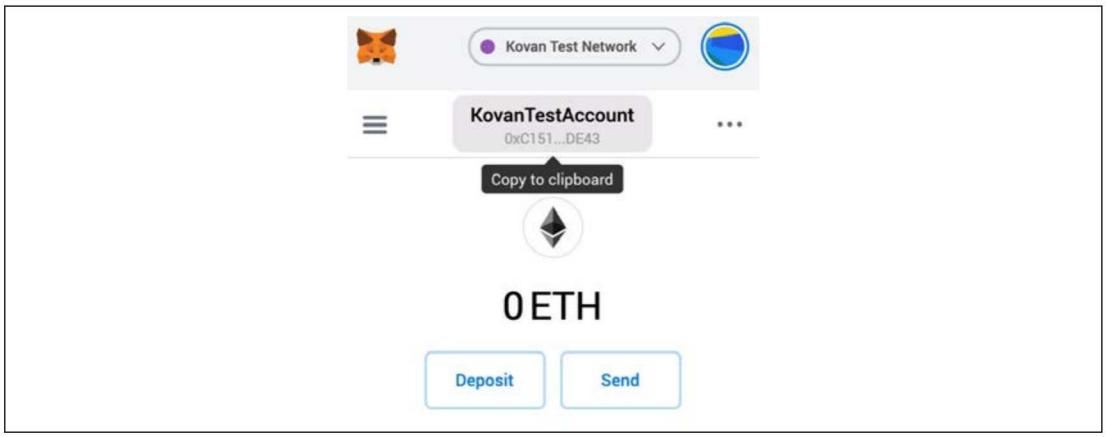


Figure 12.18: Kovan test account

4. Go to `https://gitter.im/kovan-testnet/faucet` and enter the **KovanTestAccount** address. You will receive some ETH in your account:

Figure 12.19: Funding using Kovan faucet

Now, notice in MetaMask that 3 ETH are available:

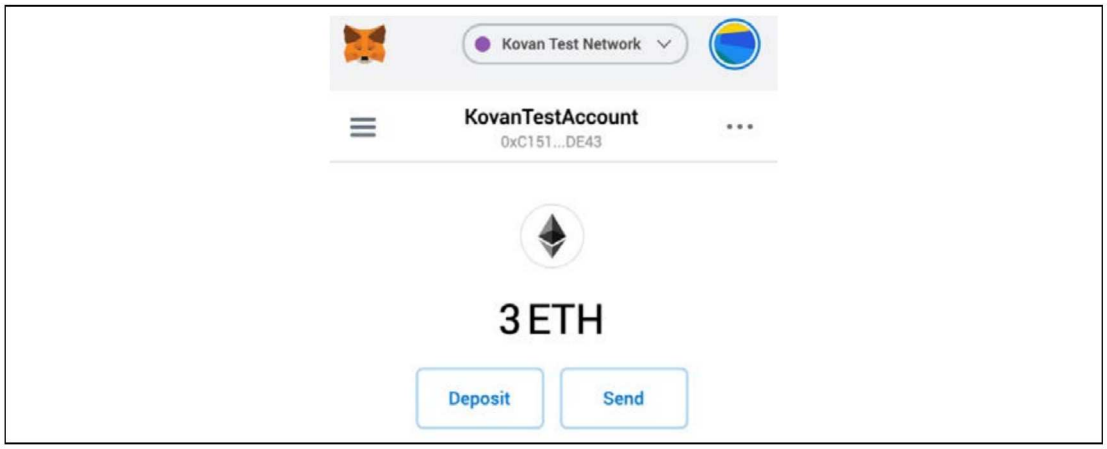


Figure 12.20: Kovan 3 ether

5. Now create another account and transfer some ETH from **KovanTestAccount** to the new account:

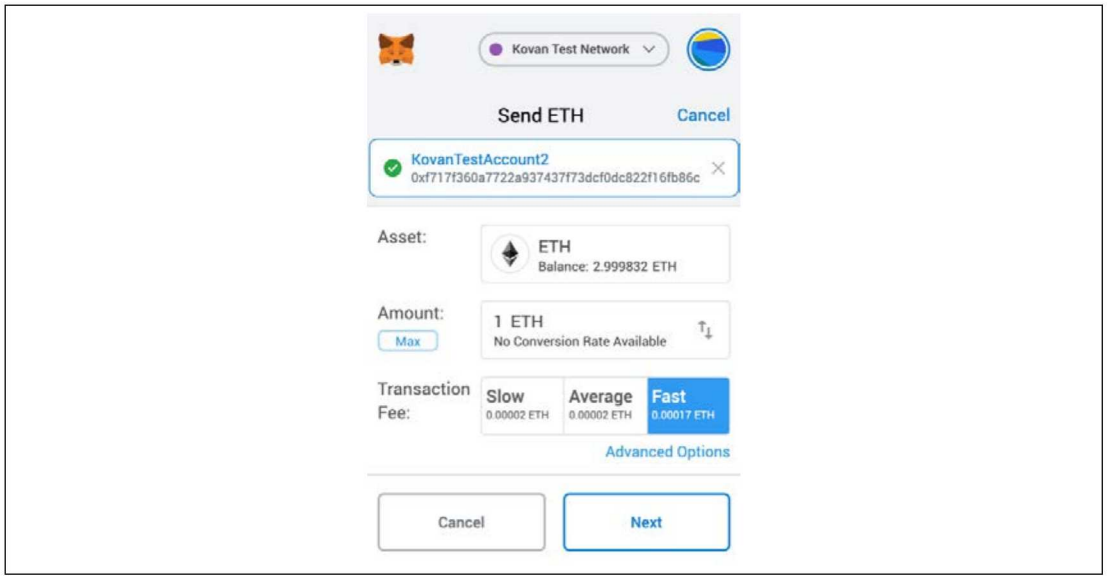


Figure 12.21: Sending ether

6. You will observe that there is an **Advanced Options** button. Click it to customize gas.

   We can now see several options related to gas and transaction fees. We usually don't need to do that, but in order to control the transaction fee and transaction speed, these options are adjusted accordingly. You can select your transaction to be fast or slow. This depends on the amount of gas you are willing to pay. The higher the gas, the greater the chances are that miners will pick it up because of higher transaction fees. Remember that we discussed gas concepts earlier. Here, these concepts will become clearer.

Notice the gas price and gas limit in the following screenshot:

- The **Gas Price** is **8**, which is the amount of ether one individual is willing to pay for each unit of gas.
- The **Gas Limit** is **21,000**, which is the maximum amount of gas the transaction sender is willing to spend.

Now, can we calculate the gas using the preceding information?

Note that the gas price is in GWei, which is also called nano-ether and represents the ninth power of the ETH in fractions. This means that 0.000000001 ETH is, in fact, 1 GWei. Notice that there are 8 zeros after the decimal point.

Now, if the gas price is 8 and the gas limit is 21,000 GWei, we use the following simple formula of (21000 * 8) * 0.000000001, to get 0.000168, which is exactly what is shown in the following screenshot:

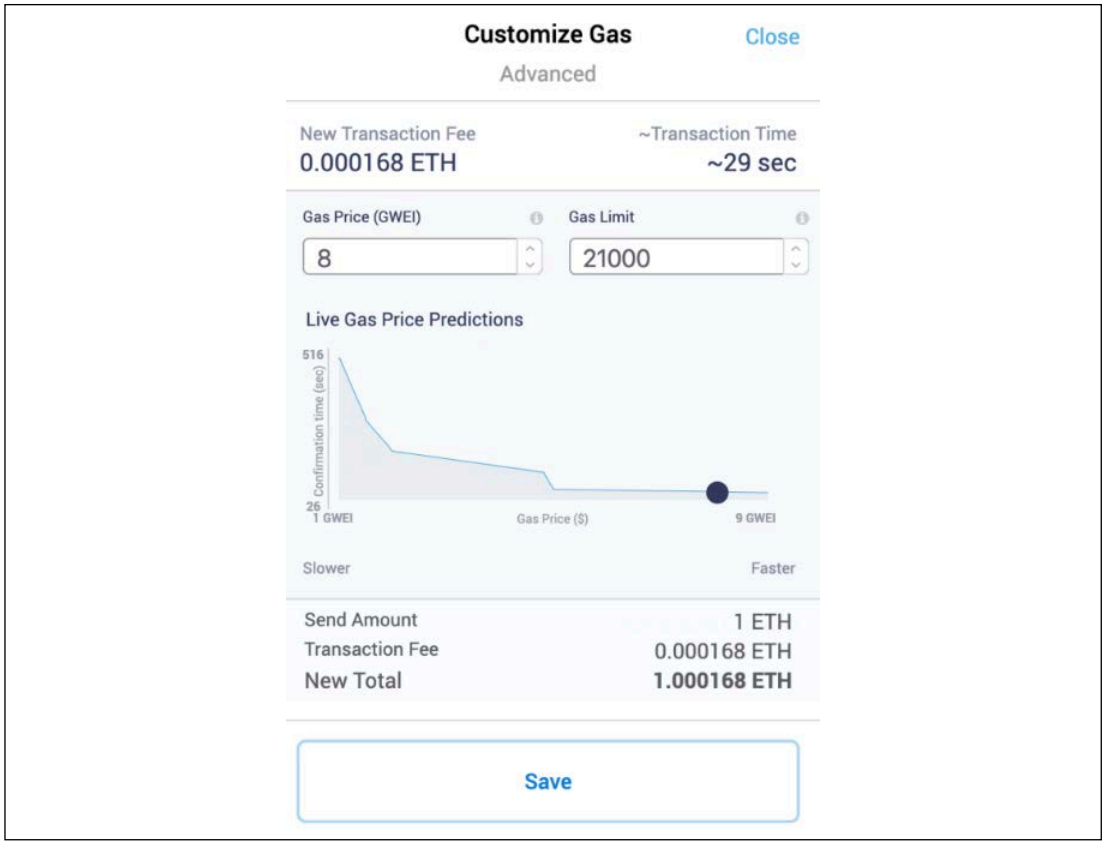


Figure 12.22: Customize Gas

7. Confirm the transaction:

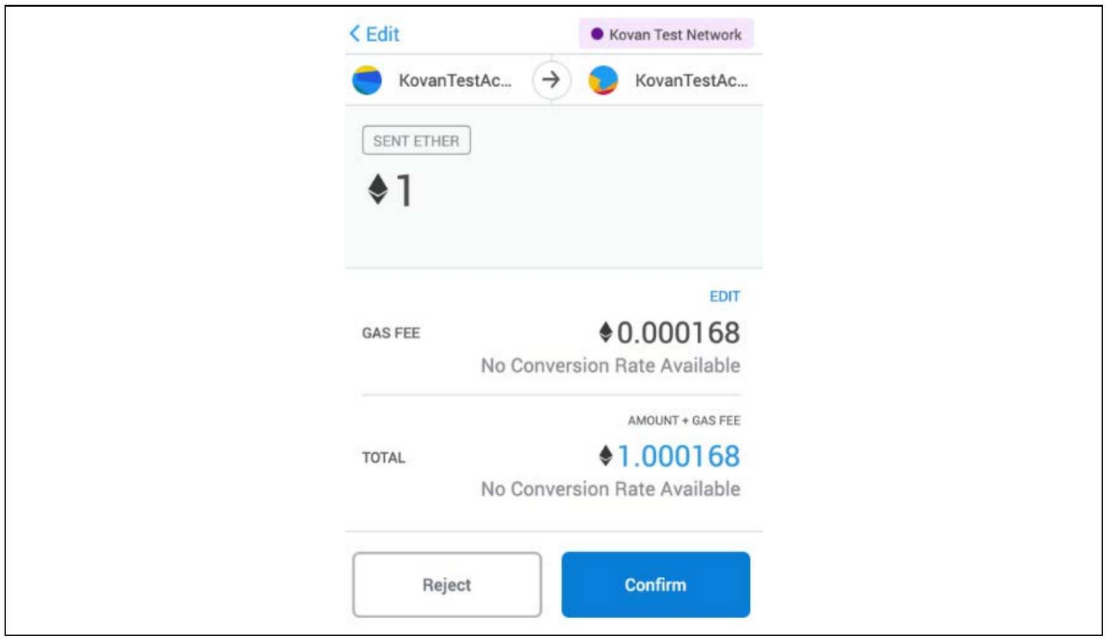


Figure 12.23: Confirming the transaction

Now, notice that the ether has been sent to the other account:

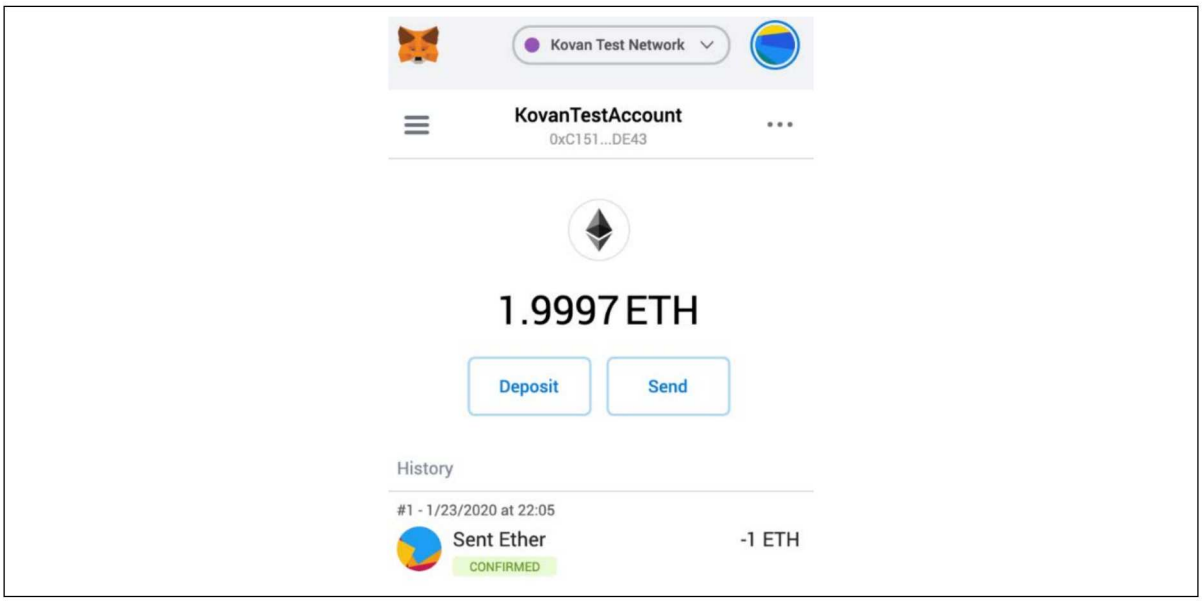


Figure 12.24: Ether sent in Kovan

Also note the detailed logs and information pertaining to the transaction:

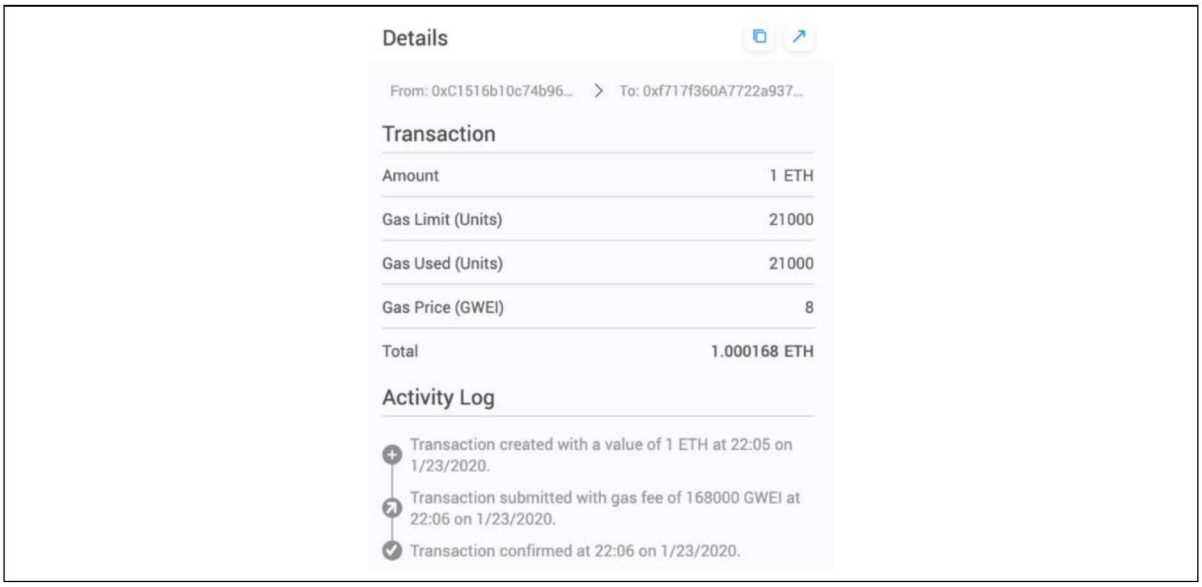


Figure 12.25: Log of the transfer

This information can also be viewed in the block explorer:

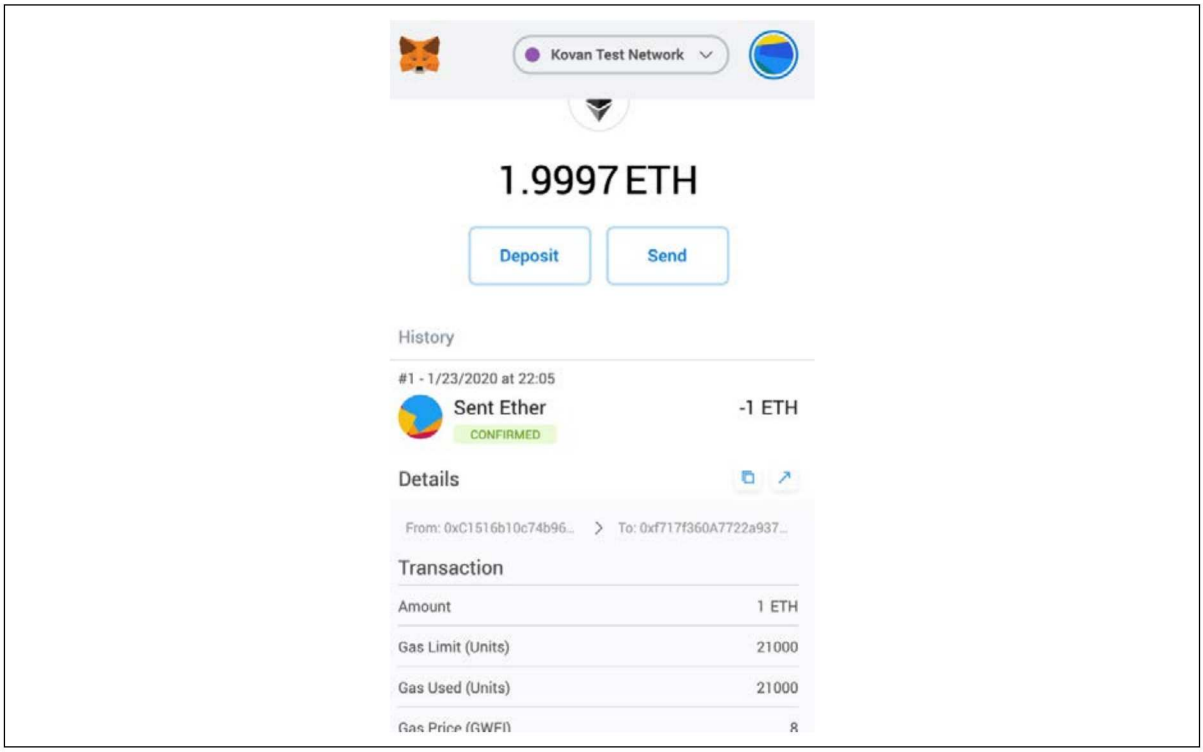


Figure 12.26: Block explorer in Kovan

The explorer can be opened by clicking on the blue-colored right arrow pointing at 45 degrees, as shown in the preceding diagram, or by following this link to the transaction summary: `https://kovan.etherscan.io/tx/0x884cc20881017aa4d49afa7c31a8db6f2ade8911c2959100aab1d10e5fcbdf72`.

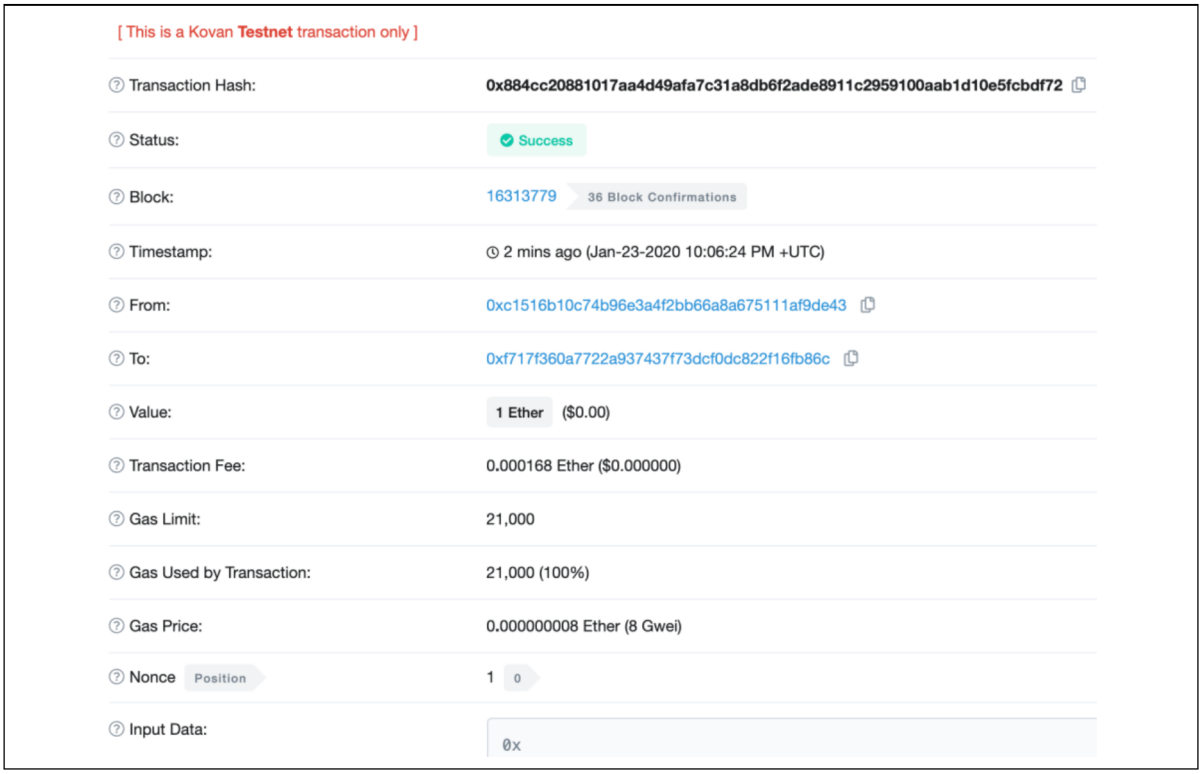


Figure 12.27: Kovan block explorer

With this, we have completed our introduction to MetaMask. We will now focus on account management using the Geth client. MetaMask and its use in development will be discussed at length in the next chapter, *Chapter 13*, *Ethereum Development Environment*.

We have now covered some popular Ethereum clients, wallets, and related usage concepts.

Next, we will introduce two of the most important elements of the Ethereum blockchain, nodes and miners. Miners perform the most important operation of the Ethereum blockchain, called mining.

Nodes and miners

The Ethereum network contains different nodes. Some nodes act only as wallets, some are light clients, and a few are full clients running the full blockchain. One of the most important types of nodes are mining nodes. We will see what constitutes mining in this section.

Mining is the process by which new blocks are selected via a consensus mechanism and added to the blockchain.

As a result of the mining operation, currency (ether) is awarded to the nodes that perform mining operations. These mining nodes are known as **miners**. Miners are paid in ether as an incentive for them to validate and verify blocks made up of transactions. The mining process helps secure the network by verifying computations.

At a theoretical level, a miner node performs the following functions:

- It listens for the transactions broadcasted on the Ethereum network and determines the transactions to be processed.
- It determines stale ommer blocks and includes them in the blockchain.
- It updates the account balance with the reward earned from successfully mining the block.
- Finally, a valid state is computed, and the block is finalized, which defines the result of all state transitions.

The current method of mining is based on PoW, which is similar to that of Bitcoin. When a block is deemed valid, it has to satisfy not only the general consistency requirements, but it must also contain the PoW for a given difficulty.

The PoW algorithm is due to be replaced by the PoS algorithm with the release of **Serenity**. There is no set date for the release of Serenity, as this will be the final version of Ethereum. Considerable research work has been carried out to build the PoS algorithm, which is suitable for the Ethereum network.

More information on PoS research work is available at `https://ethresear.ch/t/initial-explorations-on-full-pos-proposal-mechanisms/925`.

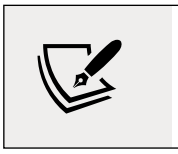
An algorithm named **Casper** has been developed that will replace the existing PoW algorithm in Ethereum. This is a security deposit based on the economic protocol where nodes are required to place a security deposit before they can produce blocks. Nodes have been named *bonded validators* in Casper, whereas the act of placing the security deposit is named *bonding*.

More information about Casper can be found here: `https://github.com/ethereum/research/tree/master/casper4`.

Miners play a vital role in reaching a consensus regarding the canonical state of the blockchain. The consensus mechanism that they contribute to is explained in the next section.

The consensus mechanism

The consensus mechanism in Ethereum is based on the **Greedy Heaviest Observed Subtree** (**GHOST**) protocol proposed initially by Zohar and Sompolinsky in December 2013.



Readers interested in this topic can find more information in the original paper at `http://eprint.iacr.org/2013/881.pdf`.

Ethereum uses a simpler version of this protocol, where the chain that has the most computational effort spent on it to build it is identified as the definite version. Another way of looking at it is to find the longest chain, as the longest chain must have been built by consuming adequate mining efforts. The GHOST protocol was first introduced as a mechanism to alleviate the issues arising out of fast block generation times that led to stale or orphaned blocks.

In GHOST, stale blocks, or ommers, are added in calculations to figure out the longest and heaviest chain of blocks.

The following diagram shows a quick comparison between the longest and heaviest chains:

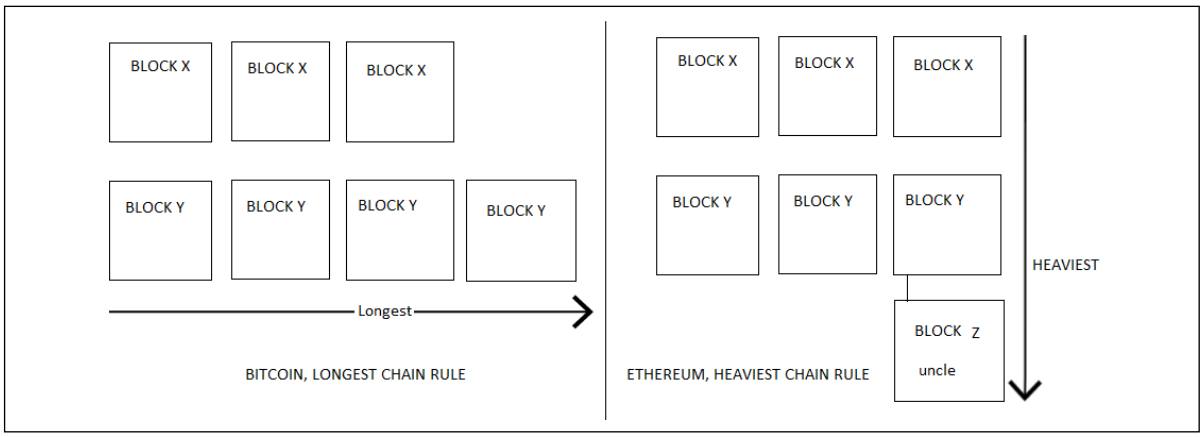


Figure 12.28: Longest versus the heaviest chain

The preceding diagram shows two rules of figuring out which blockchain is the canonical version of truth. In the case of Bitcoin, shown on the left-hand side in the diagram, the longest chain rule is applied, which means that the active chain (true chain) is the one that has the most amount of PoW done. In the case of Ethereum, the concept is similar from the point of view of the longest chain, but it also includes ommers, the *orphaned* blocks, which means that it also rewards those blocks that were competing with other blocks during mining to be selected and performed significant PoW, or were mined exactly at the same time as others but did not make it to the main chain. This makes the chain the *heaviest* instead of the *longest* because it also contains the *orphaned* blocks. This is shown on the right-hand side of the diagram.

As the blockchain progresses (more blocks are added to the blockchain) governed by the consensus mechanism, on occasion, the blockchain can split into two. This phenomenon is called **forking**.

Forks in the blockchain

A fork occurs when a blockchain splits into two. This can be intentional or non-intentional. Usually, as a result of a major protocol upgrade, a hard fork is created, while an unintentional fork can be created due to bugs in the software.

It can also be temporary as discussed previously; in other words, the longest and heaviest chain. This temporary fork occurs when a block is created almost at the same time and the chain splits into two, until it finds the longest or heaviest chain to achieve eventual consistency.

The release of **Homestead** involved major protocol upgrades, which resulted in a hard fork. The protocol was upgraded at block number 1,150,000, resulting in the migration from the first version of Ethereum, known as **Frontier**, to the second version. The latest version is called **Byzantium**, which is the first part of the **Metropolis** release. This was released as a hard fork at block number 4,370,000.

An unintentional fork, which occurred on November 24, 2016, at 14:12:07 UTC, was due to a bug in Ethereum's Geth client journaling mechanism.

As a result, a network fork occurred at block number 2,686,351. This bug resulted in Geth failing to prevent empty account deletions in the case of the empty out-of-gas exception. This was not an issue in **Parity** (another popular Ethereum client). This means that from block number 2,686,351, the Ethereum blockchain is split into two, one running with the Parity clients and the other with Geth. This issue was resolved with the release of Geth version 1.5.3.

As a result of the DAO hack, which was discussed in *Chapter 10*, *Smart Contracts*, the Ethereum blockchain was also forked to recover from the attack.

To facilitate consensus, a PoW algorithm is used. In Ethereum, the algorithm used for this purpose is called **Ethash**, which will be covered in the next section.

Ethash

Ethash is the name of the PoW algorithm used in Ethereum. Originally, this was proposed as the **Dagger-Hashimoto algorithm**, but much has changed since the first implementation, and the PoW algorithm has now evolved into what's known as Ethash.

Similar to Bitcoin, the core idea behind mining is to find a nonce (a random arbitrary number), which, once concatenated with the block header and hashed, results in a number that is lower than the current network difficulty level. Initially, the difficulty was low when Ethereum was new, and even CPU and single GPU mining was profitable to a certain extent, but that is no longer the case. Now, only either pooled mining or large GPU mining farms are used for profitable mining purposes.

Ethash is a memory-hard algorithm, which makes it difficult to be implemented on specialized hardware. As in Bitcoin, ASICs have been developed, which have resulted in mining centralization over the years, but memory-hard PoW algorithms are one way of thwarting this threat, and Ethereum implements Ethash to discourage ASIC development for mining. Ethash is a **memory-hard** algorithm and developing ASICs with large and fast memories is not feasible. This algorithm requires subsets of a fixed resource called **Directed Acyclic Graph** (**DAG**) to be chosen, depending on the nonce and block headers.

DAG is a large, pseudo-randomly generated dataset. This graph is represented as a matrix in the DAG file created during the Ethereum mining process. The Ethash algorithm expects the DAG as a two-dimensional array of 32-bit unsigned integers.

Mining can only start when DAG is completely generated the first time a mining node starts. This DAG is used as a seed by the algorithm called Ethash. According to current specifications, the epoch time is defined as 30,000 blocks, or roughly 6 days.

The Ethash algorithm requires a DAG file to work. A DAG file is generated every epoch, which is 30,000 blocks. DAG grows linearly as the chain size grows. Currently, the DAG size is around 3.5 GB (as of block 9325164) and epoch number 310.

The protocol works as follows:

1. First, the header from the previous block and a 32-bit random nonce is combined using Keccak-256.
2. This produces a 128-bit structure called `mix`.
3. `mix` determines which data is to be picked up from the DAG.
4. Once the data is fetched from the DAG, it is "mixed" with the `mix` to produce the next `mix`, which is then again used to fetch data from the DAG and subsequently mixed. This process is repeated 64 times.
5. Eventually, the 64th `mix` is run through a digest function to produce a 32-byte sequence.
6. This sequence is compared with the difficulty target. If it is less than the difficulty target, the nonce is valid, and the PoW is solved. As a result, the block is mined. If not, then the algorithm repeats with a new nonce.

More technical details at code level can be found here: `https://github.com/ethereum/go-ethereum/blob/master/consensus/ethash/algorithm.go`.

The current reward scheme is 2 ETH for successfully finding a valid nonce. In addition to receiving 2 ether, the successful miner also receives the cost of the gas consumed within the block and an additional reward for including stale blocks (uncles) in the block. A maximum of two ommers are allowed per block and are rewarded with 7/8 of the normal block reward. In order to achieve a 12-second block time, block difficulty is adjusted at every block. The rewards are proportional to the miner's hash rate, which means how fast a miner can hash. You can use an ether mining calculator to calculate what hash rate is required to generate profit.

One example of such a calculator is `https://etherscan.io/ether-mining-calculator`.

Mining can be performed by simply joining the Ethereum network and running an appropriate client. The key requirement is that the node should be fully synched with the main network before mining can start.

In the next section, various methods of mining will be explored.

CPU mining

Even though not profitable on mainnet, CPU mining is still valuable on the test network or even a private network to experiment with mining and contract deployment. Private and test networks will be discussed with practical examples in the next chapter, *Chapter 13*, *Ethereum Development Environment*. A Geth example is provided here on how to start CPU mining. Geth can be started with a mine switch in order to initiate mining:

```
$ geth --mine --minerthreads <n>
```

CPU mining can also be started using the Web3 Geth console. The Geth console can be started by issuing the following command:

```
$ geth attach
```

After this, the miner can be started by issuing the following command, which will return `True` if successful, or `False` otherwise. Take a look at the following command:

```
miner.start(4)
true
```

Number `4` here represents the number of threads that will run for mining. It can be any number depending on the number of CPUs you have.

The preceding command will start the miner with four threads. Take a look at the following command:

```
miner.stop()
true
```

The preceding command will stop the miner. The command will return `True` if successful.

GPU mining

At a basic level, GPU mining can be performed easily by running two commands:

```
geth --rpc
```

Once `geth` is up and running, and the blockchain is fully downloaded, **Ethminer** can be run in order to start mining. `ethminer` is a standalone miner that can also be used in farm mode to contribute to mining pools:

You can download Ethminer from GitHub here: `https://github.com/Genoil/cpp-ethereum/tree/117/releases`.

```
$ ethminer -G
```

Running with the `-G` switch assumes that the appropriate graphics card is installed and configured correctly.

If no appropriate graphics cards are found, `ethminer` will return an error, as shown in the following screenshot:

```
drequinox@drequinox-OP7010:~$ ethminer -G
[OPENCL]:No OpenCL platforms found
No GPU device with sufficient memory was found. Can't GPU mine. Remove the -G argument
drequinox@drequinox-OP7010:~$
```

Figure 12.29: Error in the case that no appropriate GPUs can be found

GPU mining requires an AMD or NVIDIA graphics card and an applicable OpenCL SDK.

For an NVIDIA chipset, it can be downloaded from `https://developer.nvidia.com/cuda-downloads`. For AMD chipsets, it is available at `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk`.

Once the graphics cards are installed and configured correctly, the process can be started by issuing the same `ethminer -G` command.

Benchmarking

`ethminer` can also be used to run benchmarking, as shown in the following screenshot. Two modes can be invoked for benchmarking. It can either be CPU or GPU. The commands are shown here:

CPU benchmarking:

```
$ ethminer -M -C
```

GPU benchmarking:

```
$ ethminer -M -G
```

The following screenshot is shown as an example of CPU mining benchmarking:

```
drequinox@drequinox-OP7010:~$ ethminer -M -C
  ◇  22:43:30.560 ethminer  #00004000…
Benchmarking on platform: 8-thread CPU
Preparing DAG...
  □  22:43:30.561 miner0  Loading full DAG of seedhash: #00000000…
Warming up...
Trial 1... 0
Trial 2... DAG  22:43:38.310 miner0  Generating DAG file. Progress: 0 %
0
Trial 3... 0
Trial 4... DAG  22:43:45.336 miner0  Generating DAG file. Progress: 1 %
0
```

Figure 12.30: CPU benchmarking

The GPU device to be used can also be specified in the command line:

```
$ ethminer -M -G --opencl-device 1
```

As GPU mining is implemented using OpenCL AMD, chipset-based GPUs tend to work faster as compared to NVIDIA GPUs. Due to the high memory requirements (DAG creation), FPGAs and ASICs will not provide any major advantage over GPUs. This is done on purpose to discourage the development of specialized hardware for mining.

Mining rigs

As difficulty increased over time in relation to mining ether, mining rigs with multiple GPUs started to be built by the miners. A mining rig usually contains around five GPU cards with all of them working in parallel for mining, thereby improving the chances of finding a valid nonce for mining.

Mining rigs can be built with some effort and are also available commercially from various vendors. A typical mining rig configuration includes the components discussed here:

- **Motherboard**: A specialized motherboard with multiple PCI-E x1 or x16 slots, for example, BIOSTAR Hi-Fi or ASRock H81, is required.
- **SSD hard drive**: An SSD hard drive is required. The SSD drive is recommended because of its much faster performance vis-à-vis the analog equivalent. This will mainly be used to store the blockchain. It is recommended that you have roughly over 350 GB of free space on your hard disk.

You can always check chain data size here and adjust the disk space accordingly: `https://etherscan.io/chartsync/chaindefault`.

- **GPU**: The GPU is the most critical component of the rig as it is the primary workhorse that will be used for mining. For example, it can be a Sapphire AMD Radeon R9 380 with 4 GB RAM. A page that maintains these benchmark metrics is available at `https://www.miningbenchmark.net`.

- **OS**: Linux Ubuntu's latest version is usually chosen as the OS for the rig because it is more reliable and gives better performance as compared to Windows. Also, it allows you to run a bare minimum OS required for mining, and essential operations as compared to heavy graphical interfaces that another OS may have. There is also another variant of Linux available, called **EthOS** (available at `http://ethosdistro.com/`) that is specially built for Ethereum mining and supports mining operations natively.
- **Mining software**: Finally, mining software such as Ethminer and Geth are installed. Additionally, some remote monitoring and administration software is also installed so that rigs can be monitored and managed remotely if required. It is also important to put proper air conditioning or cooling mechanisms in place, since running multiple GPUs can generate a large amount of heat. This also necessitates the need to use an appropriate monitoring software that can alert users if there are any problems with the hardware, for example, if the GPUs are overheating.
- **Power supply units (PSUs)**: In a mining rig, there are multiple GPUs running in parallel. Therefore, there is a need for a constant powerful supply of electricity and it is necessary to use PSUs that can take the load and provide enough power to the GPUs in order for them to operate. Usually, 1,000 watts of power is required to be produced by PSUs. An excellent comparison of PSUs is available at `https://www.thegeekpub.com/11488/best-power-supply-mining-cryptocurrency/`:

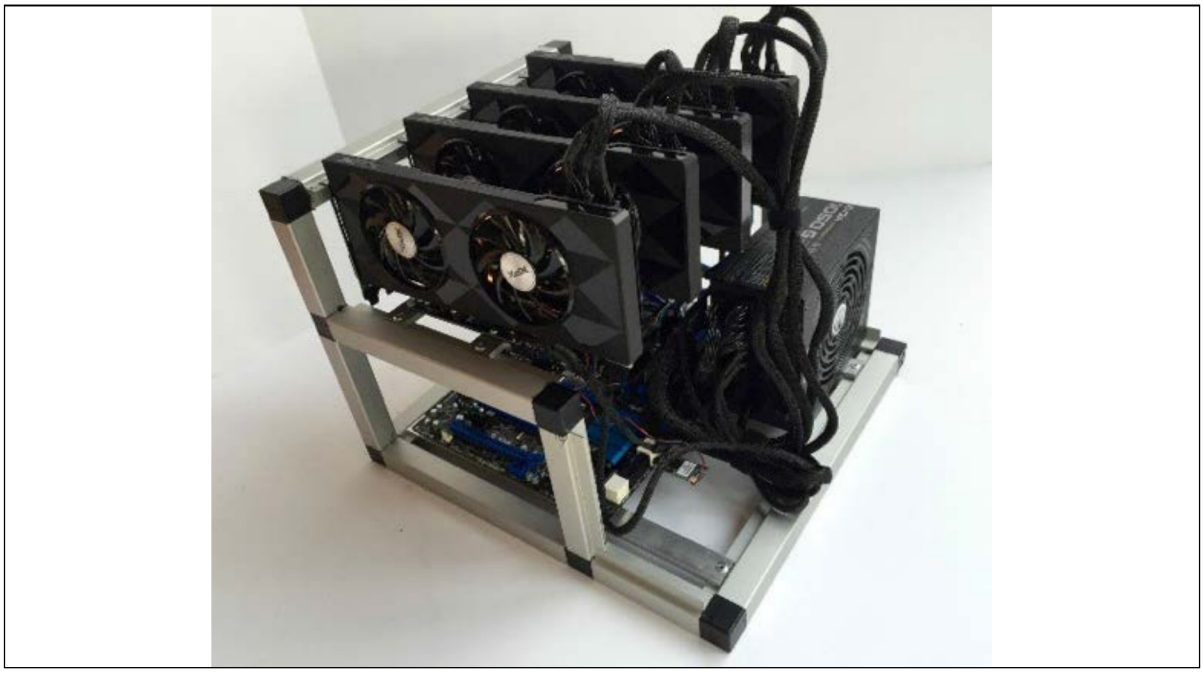


Figure 12.31: A mining rig for Ethereum

Mining pools

Many online mining pools offer Ethereum mining. Ethminer can be used to connect to a mining pool using the following command. Each pool publishes its instructions, but generally, the process of connecting to a pool is similar. An example from `http://ethereumpool.co` is shown here:

```
$ ethminer -C -F http://ethereumpool.co/?miner=0.1@0x024a20cc5feba7f3dc3776075b3e
61234eb1459 c@DrEquinox
```

This command will produce output similar to the one shown here:

```
miner 23:50:53.046 ethminer Getting work package . . .
```

This output means that the `ethminer` is running and receiving `work` from the pool to process.

In the next section, we introduce specialized mining hardware called **ASIC**. ASIC is an abbreviation of **Application Specific Integrated Circuit**, which is an integrated circuit (chip) specially developed for a particular use. In this case, the particular use is mining, for which ASICs are currently the fastest method.

ASICs

Even though Ethereum was originally considered ASIC-resistant, there are now ASICs being built for ether mining that perform significantly better than GPUs. ASIC resistance in Ethereum comes from the fact that the Ethash consensus algorithm (introduced earlier in this chapter) requires a large amount of memory to store the DAG. Since maintaining such a large memory of usually around 4 GB is quite difficult on ASICs, this feature offers some resistance against ASICs.

However, there are some companies that are building Ethereum ASICs, but these are quite difficult to acquire as they are in very high demand. A transition from the existing Ethash consensus algorithm to ProgPoW (`https://github.com/ifdefelse/ProgPOW`) has also been agreed among developers, which is also slowing down ASIC hardware adoption. However, the difficulty is increasing on the Ethereum network due the existence of these ASIC devices. Eventually, a move to a PoS system in Ethereum 2.0 will permanently address this issue.

So far, we have covered various clients, installation, relevant commands, and mining. Next, we will introduce some basic concepts related to Ethereum APIs, development, and protocols.

APIs, tools, and DApps

The Web3 JavaScript API provides an interface to the Ethereum blockchain via JavaScript. It provides an object called `web3`, which contains other objects that expose different methods to support interaction with the blockchain. This API covers methods related to administration of the blockchain, debugging, account-related operations, supporting protocol methods for Whisper, as well as storage and other network-related operations.

This API will be discussed in detail in the next chapter, *Chapter 13*, *Ethereum Development Environment*, where we will see how to interact with the Ethereum blockchain with it.

Applications (DApps and DAOs) developed on Ethereum

There are various implementations of DAOs and smart contracts in Ethereum, most notably, the DAO, which was recently misused due to a weakness in the code and required a hard fork for funds to be recovered that had been syphoned out by the attackers. The DAO was created to serve as a decentralized platform to collect and distribute investments.

Augur is another DApp that has been implemented on Ethereum, which is a decentralized prediction market.

Many other decentralized applications are listed on `https://www.stateofthedapps.com/`.

Tools

Various frameworks and tools have been developed to support decentralized application development, such as **Truffle**, **MetaMask**, **Ganache**, and many more. We will talk about these in *Chapter 14*, *Development Tools and Frameworks*.

Earlier in this chapter, we looked at different methods to connect to the blockchain, in other words, the Geth console and Geth attach. Now, we will demonstrate the use of another method that is commonly used— the Geth JSON RPC API.

Geth JSON RPC API

JSON RPC is a remote procedure call mechanism that makes use of JSON data format to encode its calls. In simpler terms, it is an RPC encoded in JSON. **JSON** stands for **JavaScript Object Notation**. It is a lightweight and easy-to-understand text format used for data transmission and storage. A remote procedure call is a distributed systems concept. It is a mechanism used to invoke a procedure in a different computer. It appears as if a local procedure call is being made because there is a requirement to write code to handle remote interactions.

Further details on RPC and JSON can be found here: `https://www.jsonrpc.org`.

For this chapter, it is sufficient to know that the JSON-RPC API is used in Ethereum extensively to allow users and DApps to interact with the blockchain.

There are a number of methods available to interact with the blockchain. One is to use the Geth console, which makes use of the Web3 API to query the blockchain. The Web3 API makes use of the JSON-RPC API. Another method is to make JSON-RPC calls directly, without using the Web3 API. In this method, direct RPC calls can be made to the Geth client over HTTP. By default, Geth RPC listens at TCP port `8545`.

Now, we will look at some examples involving utilization of the JSON RPC API. We will use a common utility curl (`https://curl.haxx.se`) for this purpose.

Examples

For these examples to work, first, the Geth client needs to be started up with appropriate flags. If there is an existing session of Geth running with other parameters, stop that instance and run the `geth` command as shown here, which will start Geth with the RPC available. The user can also control which APIs are exposed:

```
$ geth  --rpc --rpcapi "eth,net,web3,personal"
```

In this command, Geth is started up with `--rpc` and `--rpcapi` flags, along with a list of APIs that are exposed:

- The `rpc` flag enables the HTTP-RPC server.
- The `rpcapi` flag is used to define which API's are made available over the HTTP-RPC interface. This includes a number of APIs such as `eth`, `net`, `web3`, and `personal`.

For each of the following examples, run the `curl` command in the terminal, as shown in each of the following examples.

List accounts

The list of accounts can be obtained by issuing the following command:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_accounts","params":[], "id":64}'
http://localhost:8545
```

This will display the following JSON output, which lists all the Ethereum accounts owned by the client:

```
{"jsonrpc":"2.0","id":64,"result":["0x07668e548be1e17f3dcfa2c4263a0f5
f88aca402","0xba94fb1f306e4d53587fcdcd7eab8109a2e183c4","0x1df5ae4063
6d6a1a4f8db8a0c65addce5a992a14","0x76bcb051e3cedfcc9c7ff0b25a57383dac
bf833b"]}
```

Check if the network is up

We can query if the network is up by using the command shown here:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"net_listening","params":[], "id":64}'
http://localhost:8545
```

This will display the output as shown here with the result `true`, indicating that the network is up:

```
{"jsonrpc":"2.0","id":64,"result":true}
```

Providing the Geth client version

We can find the Geth client version using this command:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],
"id":64}' http://localhost:8545
```

This will display the `geth` client version:

```
{"jsonrpc":"2.0","id":64,"result":"Geth/v1.9.9-stable-01744997/linux-amd64/
go1.13.4"}
```

Synching information

To check the latest synchronization status, we can use the following command:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_syncing","params":[], "id":64}'
http://localhost:8545
```

This will display the data pertaining to the synchronization status or return `false`:

```
{"jsonrpc":"2.0","id":64,"result":{"currentBlock":"0x1d202","highest
lock":"0x8a61c8","knownStates":"0x23b9b","pulledStates":"0x2261a",
"startingBlock":"0xa5c5"}}
```

Finding the Coinbase address

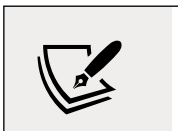
The Coinbase address can be queried using:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_coinbase","params":[], "id":64}'
http://localhost:8545
```

This will display the output shown here, indicating the client `coinbase` address:

```
{"jsonrpc":"2.0","id":64,"result":"0x07668e548be1e17f3dcfa2c4263a0f5f88aca402"}
```

These are just a few examples of extremely rich APIs that are available in Ethereum's `geth` client.



More information and official documents regarding Geth RPC APIs are available at the following link: `https://eth.wiki/json-rpc/API`.

In this section, we have covered Geth JSON RPC APIs and seen some examples on how to query the blockchain via the RPC interface. In the next section, we will introduce a number of supporting protocols that are part of the complete decentralized ecosystem based on the Ethereum blockchain.

Supporting protocols

Various supporting protocols are available to assist the complete decentralized ecosystem. This includes the Whisper and Swarm protocols. In addition to the contracts layer, which is the core blockchain layer, there are additional layers that need to be decentralized in order to achieve a complete decentralized ecosystem. This includes decentralized storage and decentralized messaging.

Whisper, which is being developed for Ethereum, is a decentralized messaging protocol, whereas **Swarm** is a decentralized storage protocol. Both of these technologies provide the basis for a fully decentralized web, and are described in the following sections.

Whisper

Whisper provides decentralized peer-to-peer messaging capabilities to the Ethereum network. In essence, Whisper is a communication protocol that DApps use to communicate with each other. The data and routing of messages are encrypted within Whisper communications. Whisper makes use of the **DEVp2p** wire protocol for exchanging messages between nodes on the network. Moreover, it is designed to be used for smaller data transfers and in scenarios where real-time communication is not required. Whisper is also designed to provide a communication layer that cannot be traced and provides *dark communication* between parties. Blockchain can be used for communication, but that is expensive, and a consensus is not really required for messages exchanged between nodes. Therefore, Whisper can be used as a protocol that allows censor-resistant communication. Whisper messages are ephemeral and have an associated **time to live** (**TTL**). Whisper is already available with Geth and can be enabled using the `-shh` option while running the Geth Ethereum client.

Official Whisper documentation is available at `https://eth.wiki/concepts/whisper/whisper`.

Swarm

Swarm has been developed as a distributed file storage platform. It is a decentralized, distributed, and peer-to-peer storage network. Files in this network are addressed by the hash of their content. This is in contrast to the traditional centralized services, where storage is available at a central location only. It has been developed as a native base layer service for the Ethereum Web3 stack. Swarm is integrated with DEVp2p, which is the multiprotocol network layer of Ethereum. Swarm is envisaged to provide a **Distributed Denial of Service** (**DDOS**)-resistant and fault-tolerant distributed storage layer for Ethereum Web 3.0. Similar to `shh` in Whisper, Swarm has a protocol called `bzz`, which is used by each Swarm node to perform various Swarm protocol operations.

Official Swarm documentation is available at: `https://swarm-guide.readthedocs.io/en/latest/`.

The following diagram gives a high-level overview of how Swarm and Whisper fit together and work with the Ethereum blockchain:

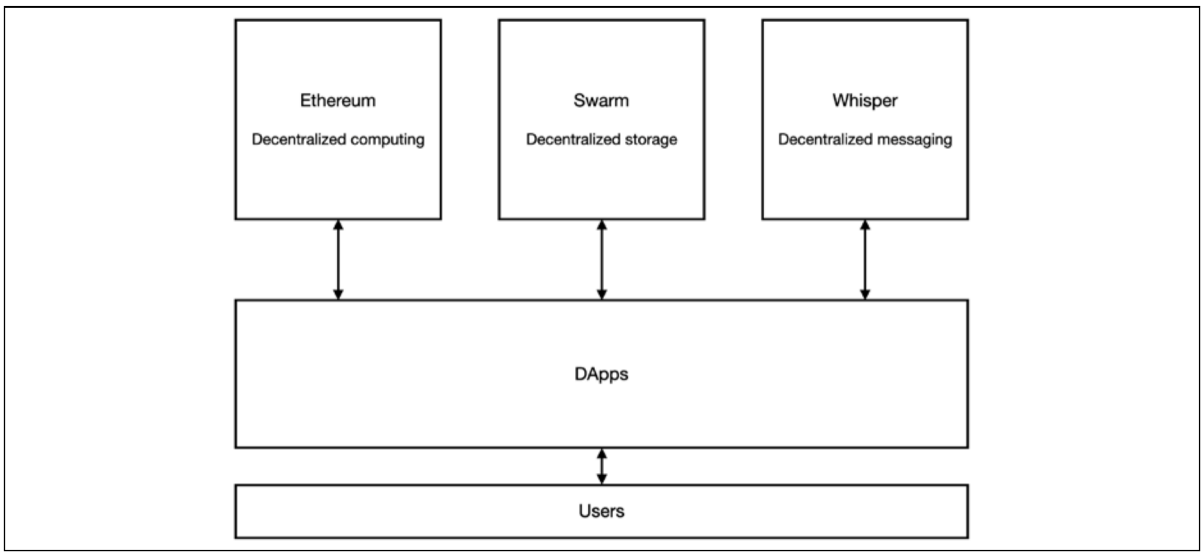


Figure 12.32: Blockchain, Whisper, and Swarm

With the development of Whisper and Swarm, a complete decentralized ecosystem emerges, where Ethereum serves as a decentralized computer, Whisper as a decentralized means of communication, and Swarm as a decentralized means of storage. In simpler words, Ethereum or more precisely, EVM, provides compute services, Whisper handles messaging and Swarm provides storage.

If you recall, in *Chapter 2*, *Decentralization*, we mentioned that decentralization of the whole ecosystem is highly desirable as opposed to decentralization of just the core computation element. The development of Whisper for decentralized communication and Swarm for decentralized storage is a step toward decentralization of the entire blockchain ecosystem.

Now, we will briefly introduce the programming languages that are being used in the development of smart contracts on the Ethereum blockchain. Also, we will cover the underlying instruction set that underpins the operation of EVM.

Programming languages

Code for smart contracts in Ethereum is written in high-level languages such as Serpent, **Low-level Lisp-like Language** (**LLL**), Solidity, or Vyper, and is converted into the bytecode that the EVM understands for it to be executed.

Solidity is one of the high-level languages that has been developed for Ethereum. It uses JavaScript-like syntax to write code for smart contracts. Once the code is written, it is compiled into bytecode that's understandable by the EVM using the Solidity compiler called **solc**.

Official Solidity documentation is available at `http://solidity.readthedocs.io/en/latest/.`

LLL is another language that is used to write smart contract code.

Serpent is a Python-like, high-level language that can be used to write smart contracts for Ethereum.

Vyper is a newer language that has been developed from scratch to achieve a secure, simple, and auditable language.

More information regarding Vyper is available at `https://github.com/ethereum/vyper`.

LLL and Serpent are no longer supported by the community and their usage has almost vanished. The most commonly used language is Solidity, which we will discuss at length in this chapter.

For example, a simple program in Solidity is shown as follows:

```
pragma solidity ^0.6.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

This program is converted into bytecode, as shown in the following subsection. Details on how to compile Solidity code with examples will be provided in *Chapter 14*, *Development Tools and Frameworks*.

Runtime bytecode

The runtime bytecode is what executes on the EVM. The smart contract code (`contract Test1`) from the previous section, is translated into binary runtime and opcodes, as follows:

Binary of the runtime (Raw hex codes):

```
608060405234801560f57600080fd5b506004361060285760003560e01c80634d3e4
6e414602d575b600080fd5b60336049565b604051808281526020019150506040518
910390f35b60006002600054190509056fea2646970667358221204b6ab44aed247
87c35f5a1942f8d3005e7a35314d03d62ced225c1ee1353653864736f6c634300060b
0033
```

Opcodes:

```
PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x2 PUSH1 0x0 SSTORE CALLVALUE
DUP1 ISZERO PUSH1 0x14 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1
0x8C DUP1 PUSH2 0x23 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN INVALID
PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH1 0xF JUMPI
PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0x4 CALLDATASIZE LT PUSH1
0x28 JUMPI PUSH1 0x0 CALLDATALOAD PUSH1 0xE0 SHR DUP1 PUSH4
0x4D3E46E4 EQ PUSH1 0x2D JUMPI JUMPDEST PUSH1 0x0 DUP1 REVERT
JUMPDEST PUSH1 0x33 PUSH1 0x49 JUMP JUMPDEST PUSH1 0x40 MLOAD DUP1
DUP3 DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP POP PUSH1 0x40 MLOAD DUP1
SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH1 0x0 PUSH1 0x2 PUSH1 0x0 SLOAD
ADD SWAP1 POP SWAP1 JUMP INVALID LOG2 PUSH5 0x6970667358 0x22 SLT
KECCAK256 0x4B PUSH11 0xB44AED24787C35F5A1942F DUP14 ADDRESS SDIV
0xE7 LOG3 MSTORE8 EQ 0xD0 RETURNDATASIZE PUSH3 0xCED225 0xC1 0xEE SGT
MSTORE8 PUSH6 0x3864736F6C63 NUMBER STOP MOD SIGNEXTEND STOP CALLER
```

Opcodes

There are many different **opcodes** that have been introduced in the EVM. Opcodes are divided into multiple categories based on the operation they perform. A list of opcodes, their meanings, and usages is available in the extra online content repository for this book here: `https://static.packt-cdn.com/downloads/Altcoins_Ethereum_Projects_and_More_Bonus_Content.pdf`.

Summary

This chapter started with some general concepts, including blocks, block structure, gas, and messages, which were introduced and discussed in detail. The later sections of the chapter introduced the practical installation and management of Ethereum wallets and clients. The two most commonly-used clients, Geth and OpenEthereum, were discussed. We also explored an introduction of programming languages for programming smart contracts in Ethereum.

Supporting protocols and topics related to challenges faced by Ethereum were also presented. Ethereum is under continuous development, and new improvements are being made by a dedicated community of developers on a regular basis. Ethereum improvement proposals, available at `https://github.com/ethereum/EIPs`, are also an indication of the magnitude of research and the keen interest exhibited by the community in this technology.

In the next chapter, we will explore Ethereum smart contract development, along with the relevant tools and frameworks.