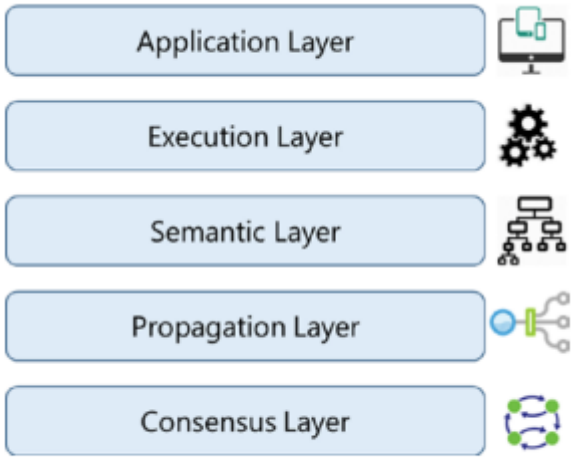
UNIT I

Differentiate Centralized and decentralized System.

Explain the importance of block chain.

Define block chain. Explain the block chain data structure.
Explain bitcoin network.
Write a short note on layers of block chain.

1. Ans:



2. *Various layers of blockchain*

Application Layer
This is the layer where you code up the desired functionalities and make an application out of it for the end users. It usually involves a traditional tech stack for software development such as client-side programming constructs, scripting, APIs, development frameworks, etc. For the applications that treat blockchain as a backend, those applications might need to be hosted on some web servers and that might require web application development, server-side programming, and APIs, etc. Ideally, good blockchain applications do not have a client–server model, and there are no centralized servers that the clients access, which is just the way Bitcoin works.
Execution Layer The Execution Layer is where the executions of instructions ordered by the Application Layer take place on all the nodes in a blockchain network. The instructions could be simple instructions or a set of multiple instructions in the form of a smart contract. In either case, a program or a script needs to be executed to ensure the correct execution of the transaction. All the nodes in a blockchain network have to execute the programs/scripts independently. Deterministic execution of programs/scripts on the same set of inputs and conditions always produces the same output on all the nodes, which helps avoid inconsistencies.
Semantic Layer The Semantic Layer is a logical layer because there is an orderliness in the transactions and blocks. A transaction, whether valid or invalid, has a set of instructions that gets through the Execution Layer but gets validated in the Semantic

Layer. If it is Bitcoin, then whether one is spending a legitimate transaction, whether it is a double-spend attack, whether one is authorized to make this transaction, etc., are validated in this layer. You will learn in the following chapters that Bitcoins are actually present as transactions that represent the system state. To be able to spend a Bitcoin, you have to consume one or more previous transactions and there is no notion of Accounts. This means that when someone makes a transaction,they use one of the previous transactions where they had received at least the amount they are spending now. This transaction must be validated by all the nodes by traversing previous transactions to see if it is a legitimate transaction.

Propagation Layer

When a transaction is made, we know that it gets broadcast to the entire network. Similarly, when a node wants to propose a valid block, it gets immediately propagated to the entire network so that other nodes could build on it, considering it as the latest block. So, transaction/block propagation in the network is defined in this layer, which ensures stability of the whole network. By design, most of the blockchains are designed such that they forward a transaction/block immediately to all the nodes they are directly connected to, when they get to know of a new transaction/block. In the asynchronous Internet network, there are often latency issues for transaction or block propagation. Some propagations occur within seconds and some take more time, depending on the capacity of the nodes, network bandwidth, and a few more factors

Consensus Layer The Consensus Layer is usually the base layer for most of the blockchain systems. The primary purpose of this layer is to get all the nodes to agree on one consistent state of the ledger. There could be different ways of achieving consensus among the nodes, depending on the use case. Safety and security of the blockchain is accertained in this layer.

2. Explain about block chain use cases.

Ans:Any type of property or asset, whether physical or digital, such as laptops, mobile phones, diamonds, automobiles, real estate, e-registrations, digital files, etc. can be registered on blockchain. This can enable these asset transactions from one person to another, maintain the transaction log, and check validity or ownerships. Also, notary services, proof of existence, tailored insurance schemes, and many more such use cases can be developed.

There are many financial use cases being developed on blockchain such as cross-border payments, share trading, loyalty and rewards system, Know Your Customer (KYC) among banks, etc. Initial Coin Offering (ICO) is one of the most trending use cases as of this writing. ICO is the best way of crowdsourcing today by using cryptocurrency as digital assets. A coin in an ICO can be thought of as a digital stock in an enterprise, which is very easy to buy and trade

Blockchain can be used to enable “The Wisdom of Crowds” to take the lead and shape businesses, economies, and various other national phenomena by using collective wisdom! Financial and economic forecasts based on the wisdom of

crowds, decentralized prediction markets, decentralized voting, as well as stocks trading can be possible on blockchain

This is the IoT era, with billions of IoT devices everywhere and many more to join the pool. A whole bunch of different makes, models, and communication protocols makes it difficult to have a centralized system to control the devices and provide a common data exchange platform. This is also an area where blockchain can be used to build a decentralized peerto-peer system for the IoT devices to communicate with each other.

In the government sectors as well, blockchain has gained momentum. There are use cases where technical decentralization is necessary, but politically should be governed by governments: land registration, vehicle registration and management, e-Voting, etc. are some of the active use cases.

5) What is Cryptography.ExplanKerckhoff's Principle and XOR Function

cryptography refers to secure information and communication techniques derived from mathematical concepts and a set of rule-based calculations called algorithms, to transform messages in ways that are hard to decipher. These deterministic algorithms are used for cryptographic key generation, digital signing, verification to protect data privacy, web browsing on the internet and confidential communications such as credit card transactions and email.

Cryptography is the most important component of blockchain. It is certainly a research field in itself and is based on advanced mathematical techniques that are quite complex to understand.

Kerckhoff's principle states that a cryptosystem should be secured even if everything about the system is publicly known, except the key. Also, the general assumption is that the message transmission channel is never secure, and messages could easily be intercepted during transmission. This means that even if the encryption algorithm E and decryption algorithm D are public, and there is a chance that the message could be intercepted during transmission, the message is still secure due to a shared secret. So, the keys must be kept secret in a symmetric cryptosystem

The XOR function is the basic building block for many encryption and decryption algorithms. Let us take a look at it to understand how it enables cryptography. The XOR, otherwise known as "Exclusive OR" and denoted by the symbol $\oplus$ can be represented by the following truth table

A	B	$A \oplus B$
0	0	0
1	0	1
0	1	1
1	1	0

The XOR function has the following properties, which are important to understand the math behindcryptography:

• Associative: A ⊕(B ⊕C) = (A ⊕B) ⊕C

• Commutative: A ⊕B = B ⊕A

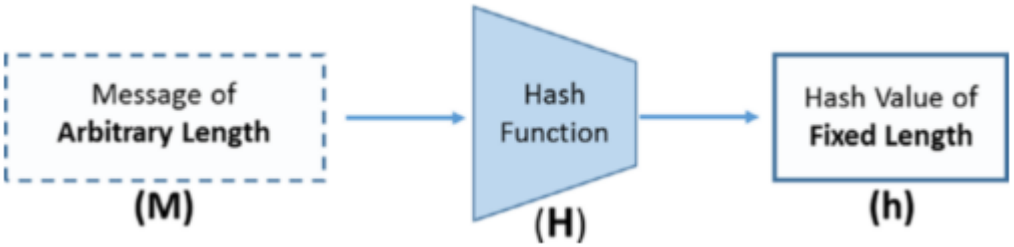
• Negation: A ⊕1 = Ā

• Identity: A ⊕A = 0

Using these properties, it would now make sense how to compute the ciphertext "c" using plaintext "m" and the key "k," and then decrypt the ciphertext "c" with the same key "k" to get the plaintext "m." The same XOR function is used for both encryption and decryption. m ⊕k = c and c ⊕k = m

6) Write a short note on cryptographic hash function.

Hash functions are the mathematical functions that are the most important cryptographic primitives and are an integral part of blockchain data structure. They are widely used in many cryptographic protocols, information security applications such as Digital Signatures and message authentication codes (MACs). Since it is used in asymmetric key cryptography, we will discuss it here prior to getting into asymmetriccryptography. Please note that the concepts covered in this section may not be in accordance with the academic text books, and a little biased toward the blockchain ecosystem. Cryptographic hash functions are a special class of hash functions that are apt for cryptography, and we will limit our discussion pertaining to it only. So, a cryptographic hash function is a one-way function that converts input data of arbitrary length and produces a fixed-length output.



For the hash functions to serve their design purpose and be usable, they should have the following core properties:

• Input can be any string of any size, but the output is of fixed length, say, a 256-bit output or a 512-bit output as examples.

• The hash value should be efficiently computable for any given message.

• It is deterministic, in the sense that the same input when provided to the same hash function produces the same hash value every time.

• It is infeasible (though not impossible!) to invert and generate the message from its hash value, except trying for all possible messages

• Any small change in the message should greatly influence the output hash, just so no one can correlate the new hash value with the old one after a small change

UNIT II

WEB2 VS WEB3

> Web2 refers to the version of the internet most of us know today. An internet dominated by companies that provide services in exchange for your personal data. Web3, in the context of Ethereum, refers to decentralized apps that run on the blockchain. These are apps that allow anyone to participate without monetising their personal data.

WEB3 BENEFITS

Many Web3 developers have chosen to build dapps because of Ethereum's inherent decentralization:

- Anyone who is on the network has permission to use the service – or in other words, permission isn't required.
- No one can block you or deny you access to the service.
- Payments are built in via the native token, ether (ETH).
- Ethereum is turing-complete, meaning you can pretty much program anything.

PRACTICAL COMPARISONS

Web2	Web3
Twitter can censor any account or tweet	Web3 tweets would be uncensorable because control is decentralized
Payment service may decide to not allow payments for certain types of work	Web3 payment apps require no personal data and can't prevent payments
Servers for gig-economy apps could go down and affect worker income	Web3 servers can't go down - they use Ethereum, a decentralized network of 1000s of computers as their backend

This doesn't mean that all services need to be turned into a dapp. These examples are illustrative of the main differences between web2 and web3 services.

WEB3 LIMITATIONS

Web3 has some limitations right now:

- Scalability – transactions are slower on web3 because they're decentralized. Changes to state, like a payment, need to be processed by a miner and propagated throughout the network.
- UX – interacting with web3 applications can require extra steps, software, and education. This can be a hurdle to adoption.
- Accessibility – the lack of integration in modern web browsers makes web3 less accessible to most users.

- Cost – most successful dapps put very small portions of their code on the blockchain as it's expensive.

CENTRALIZATION VS DECENTRALIZATION

In the table below, we list some of the broad-strokes advantages and disadvantages of centralized and decentralized digital networks.

Centralized Systems	Decentralized Systems
Low network diameter (all participants are connected to a central authority); information propagates quickly, as propagation is handled by a central authority with lots of computational resources.	The furthest participants on the network may potentially be many edges away from each other. Information broadcast from one side of the network may take a long time to reach the other side.
Usually higher performance (higher throughput, fewer total computational resources expended) and easier to implement.	Usually lower performance (lower throughput, more total computational resources expended) and more complex to implement.
In the event of conflicting data, resolution is clear and easy: the ultimate source of truth is the central authority.	A protocol (often complex) is needed for dispute resolution, if peers make conflicting claims about the state of data which participants are meant to be synchronized on.
Single point of failure: malicious actors may be able to take down the network by targeting the central authority.	No single point of failure: network can still function even if a large proportion of participants are attacked/taken out.
Coordination among network participants is much easier, and is handled by a central authority. Central authority can compel network participants to adopt upgrades, protocol updates,	Coordination is often difficult, as no single agent has the final say in network-level decisions, protocol upgrades, etc. In the worst case, network is prone to fracturing when there are disagreements about protocol

Centralized Systems	Decentralized Systems
etc., with very little friction.	changes.
Central authority can censor data, potentially cutting off parts of the network from interacting with the rest of the network.	Censorship is much harder, as information has many ways to propagate across the network.
Participation in the network is controlled by the central authority.	Anyone can participate in the network; there are no “gatekeepers.” Ideally, the cost of participation is very low.

TRANSACTIONS

> Transactions are cryptographically signed instructions from accounts. An account will initiate a transaction to update the state of the Ethereum network. The simplest transaction is transferring ETH from one account to anothe

WHAT'S A TRANSACTION?

An Ethereum transaction refers to an action initiated by an externally-owned account, in other words an account managed by a human, not a contract. For example, if Bob sends Alice 1 ETH, Bob's account must be debited and Alice's must be credited. This state-changing action takes place within a transaction.

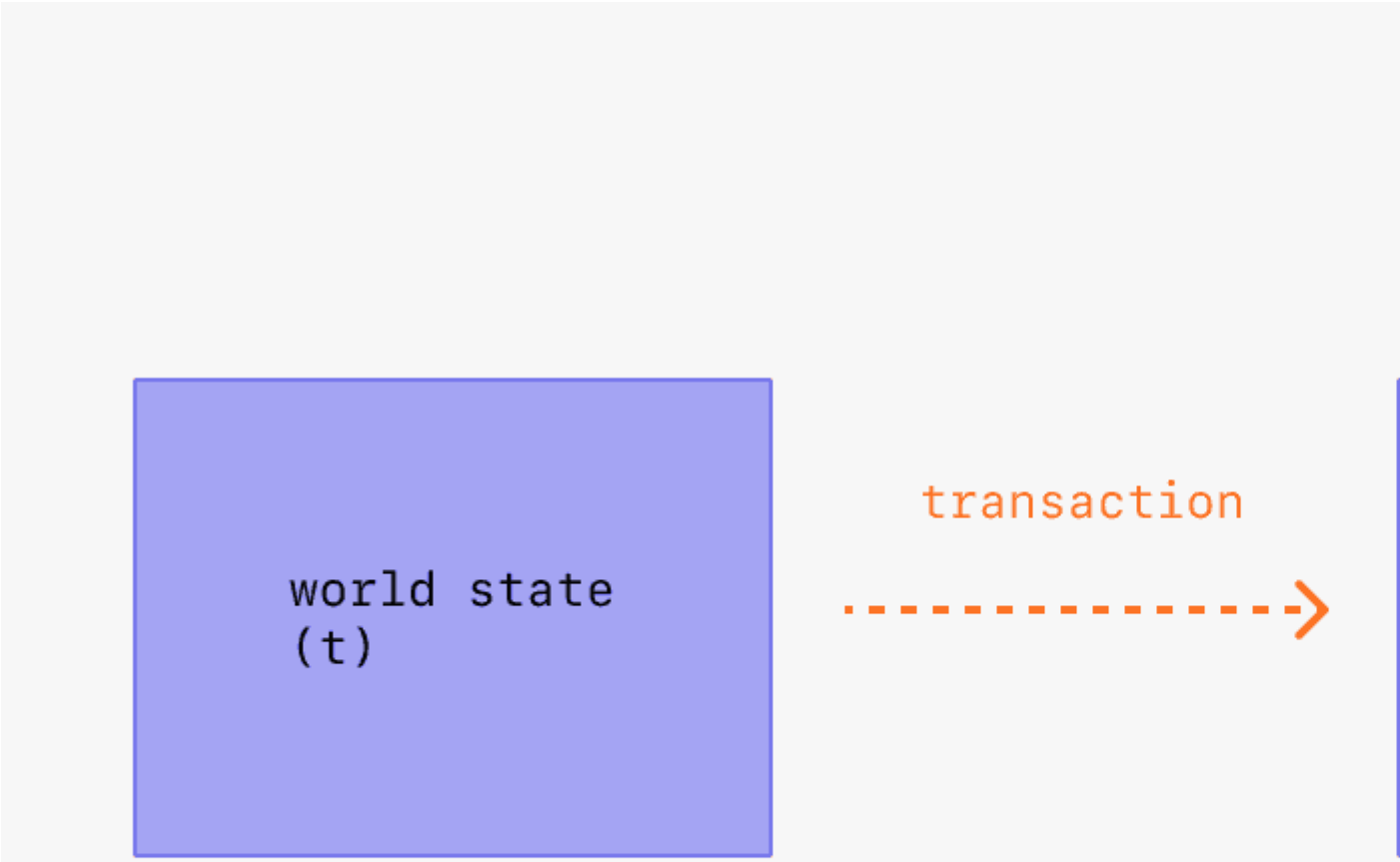


Diagram adapted from [Ethereum EVM illustrated]

Transactions, which change the state of the EVM, need to be broadcast to the whole network. Any node can broadcast a request for a transaction to be executed on the EVM; after this happens, a miner will execute the transaction and propagate the resulting state change to the rest of the network.

Transactions require a fee and must be mined to become valid. To make this overview simpler we'll cover gas fees and mining elsewhere.

A submitted transaction includes the following information:

- `recipient` – the receiving address (if an externally-owned account, the transaction will transfer value. If a contract account, the transaction will execute the contract code)
- `signature` – the identifier of the sender. This is generated when the sender's private key signs the transaction and confirms the sender has authorized this transaction
- `value` – amount of ETH to transfer from sender to recipient (in WEI, a denomination of ETH)
- `data` – optional field to include arbitrary data

- `gasLimit` – the maximum amount of gas units that can be consumed by the transaction. Units of gas represent computational steps
- `maxPriorityFeePerGas` - the maximum amount of gas to be included as a tip to the miner
- `maxFeePerGas` - the maximum amount of gas willing to be paid for the transaction (inclusive of `baseFeePerGas` and `maxPriorityFeePerGas`)

Gas is a reference to the computation required to process the transaction by a miner. Users have to pay a fee for this computation. The `gasLimit`, and `maxPriorityFeePerGas` determine the maximum transaction fee paid to the miner. More on Gas.

The transaction object will look a little like this:

```
{
  from: "0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8",
  to: "0xac03bb73b6a9e108530aff4df5077c2b3d481e5a",
  gasLimit: "21000",
  maxFeePerGas: "300",
  maxPriorityFeePerGas: "10",
  nonce: "0",
  value: "10000000000"
}

Show all

Copy
```

But a transaction object needs to be signed using the sender's private key. This proves that the transaction could only have come from the sender and was not sent fraudulently.

An Ethereum client like Geth will handle this signing process.

Example JSON-RPC call:

```
{
  "id": 2,
  "jsonrpc": "2.0",
  "method": "account_signTransaction",
  "params": [
    {
      "from": "0x1923f626bb8dc025849e00f99c25fe2b2f7fb0db",
      "gas": "0x55555",
      "maxFeePerGas": "0x1234",
       "maxPriorityFeePerGas": "0x1234",
       "input": "0xabcd",
       "nonce": "0x0",
       "to": "0x07a565b7ed7d7a678680a4c162885bedbb695fe0",
       "value": "0x1234"
    }
  ]
```

```
17}
18
```

Show all

Copy

Example response:

```
1{
2  "jsonrpc": "2.0",
3  "id": 2,
4  "result": {
5    "raw":
"0xf88380018203339407a565b7ed7d7a678680a4c162885bedbb695fe080a44401a
6e400000000000000000000000000000000000000000000000000000000001226a0223a7c9bcf5531c99be5ea7082183816eb20cfe0bbc322e97cc5c7f71ab8b20ea
02aadee6b34b45bb15bc42d9c09de4a6754e7000908da72d48cc7704971491663",
6    "tx": {
7      "nonce": "0x0",
8      "maxFeePerGas": "0x1234",
9      "maxPriorityFeePerGas": "0x1234",
10      "gas": "0x55555",
11      "to": "0x07a565b7ed7d7a678680a4c162885bedbb695fe0",
12      "value": "0x1234",
13      "input": "0xabcd",
14      "v": "0x26",
15      "r":
"0x223a7c9bcf5531c99be5ea7082183816eb20cfe0bbc322e97cc5c7f71ab8b20e",
16      "s":
"0x2aadee6b34b45bb15bc42d9c09de4a6754e7000908da72d48cc7704971491663",
17      "hash":
"0xeba2df809e7a612a0a0d444ccfa5c839624bdc00dd29e3340d46df3870f8a30e"
18    }
19  }
20}
21
```

Show all

Copy

- the `raw` is the signed transaction in Recursive Length Prefix (RLP) encoded form
- the `tx` is the signed transaction in JSON form

With the signature hash, the transaction can be cryptographically proven that it came from the sender and submitted to the network.

The data field

The vast majority of transactions access a contract from an externally-owned account. Most contracts are written in Solidity and interpret their data field in accordance with the application binary interface (ABI).

The first four bytes specify which function to call, using the hash of the function's name and arguments. You can sometimes identify the function from the selector using this database.

The rest of the calldata is the arguments, encoded as specified in the ABI specs.

For example, lets look at this transaction. Use **Click to see More** to see the calldata.

The function selector is `0xa9059cbb`. There are several known functions with this signature. In this case the contract source code has been uploaded to Etherscan, so we know the function is `transfer(address,uint256)`.

The rest of the data is:

```
1000000000000000000000004f6742badb049791cd9a37ea913f2bac38d01279
2000000000000000000000000000000000000000000000000000000003b0559f4
3
```

According to the ABI specifications, integer values (such as addresses, which are 20-byte integers) appear in the ABI as 32-byte words, padded with zeros in the front. So we know that the `to` address is `4f6742badb049791cd9a37ea913f2bac38d01279`. The `value` is 0x3b0559f4 = 990206452.

TYPES OF TRANSACTIONS

On Ethereum there are a few different types of transactions:

- Regular transactions: a transaction from one wallet to another.
- Contract deployment transactions: a transaction without a 'to' address, where the data field is used for the contract code.
- Execution of a contract: a transaction that interacts with a deployed smart contract. In this case, 'to' address is the smart contract address.

On gas

As mentioned, transactions cost gas to execute. Simple transfer transactions require 21000 units of Gas.

So for Bob to send Alice 1 ETH at a `baseFeePerGas` of 190 gwei and `maxPriorityFeePerGas` of 10 gwei, Bob will need to pay the following fee:

```
1(190 + 10) * 21000 = 4,200,000 gwei
2--or--
30.0042 ETH
4
```

Bob's account will be debited **-1.0042 ETH**

Alice's account will be credited **+1.0 ETH**

The base fee will be burned **-0.00399 ETH**

Miner keeps the tip **+0.000210 ETH**

Gas is required for any smart contract interaction too.

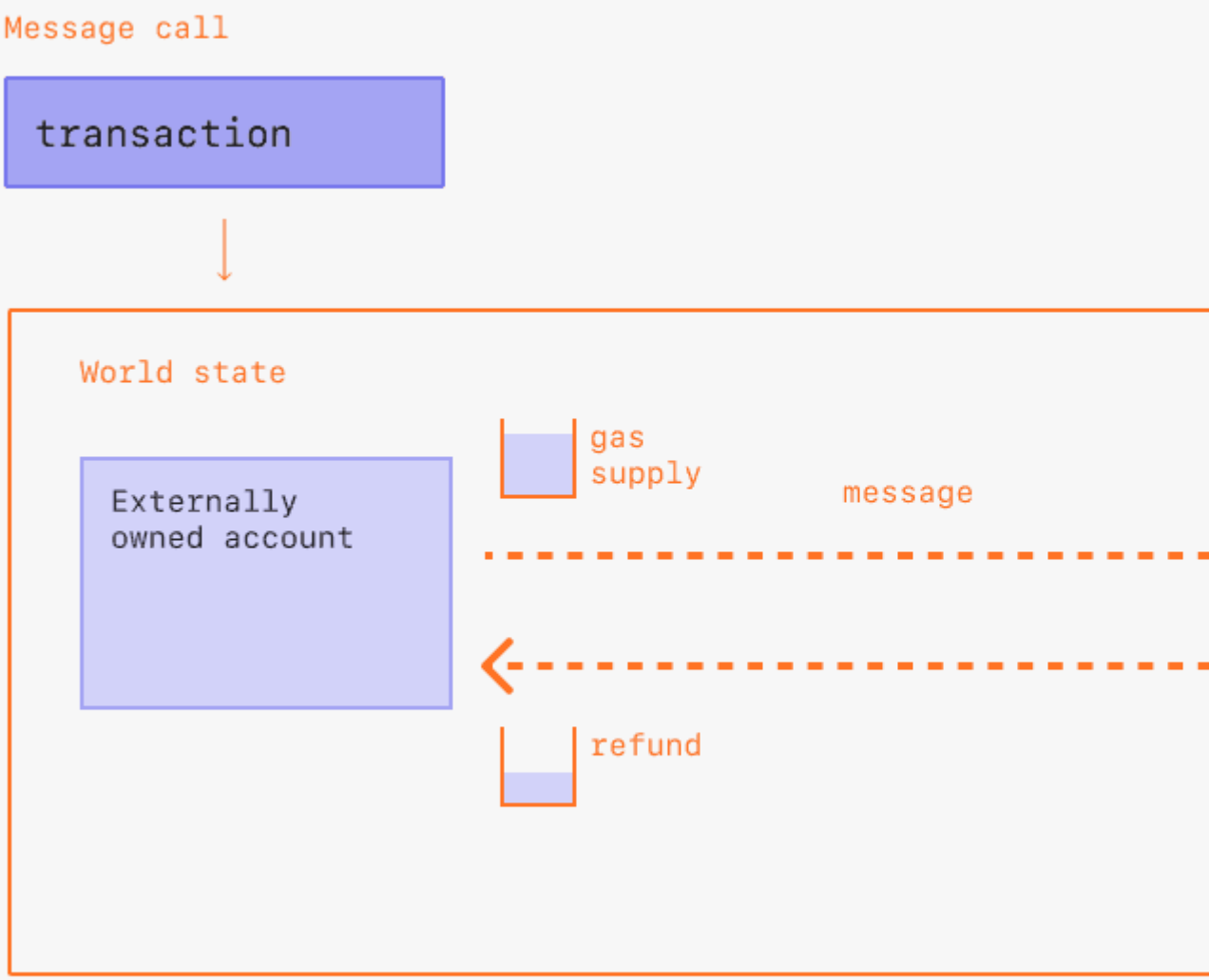


Diagram adapted from Ethereum EVM illustrated

Any gas not used in a transaction is refunded to the user account.

TRANSACTION LIFECYCLE

Once the transaction has been submitted the following happens:

1. Once you send a transaction, cryptography generates a transaction hash: `0x97d99bc7729211111a21b12c933c949d4f31684f1d6954ff477d0477538ff017`
2. The transaction is then broadcast to the network and included in a pool with lots of other transactions.

3. A miner must pick your transaction and include it in a block in order to verify the transaction and consider it "successful".
    - You may end up waiting at this stage if the network is busy and miners aren't able to keep up.
4. Your transaction will receive "confirmations". The number of confirmations is the number of blocks created since the block that included your transaction. The higher the number, the greater the certainty that the network processed and recognized the transaction.
    - Recent blocks may get re-organized, giving the impression the transaction was unsuccessful; however, the transaction may still be valid but included in a different block.
    - The probability of a re-organization diminishes with every subsequent block mined, i.e. the greater the number of confirmations, the more immutable the transaction is.

GAS AND FEES

Gas is essential to the Ethereum network. It is the fuel that allows it to operate, in the same way that a car needs gasoline to run.

WHAT IS GAS?

Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network.

Since each Ethereum transaction requires computational resources to execute, each transaction requires a fee. Gas refers to the fee required to conduct a transaction on Ethereum successfully.

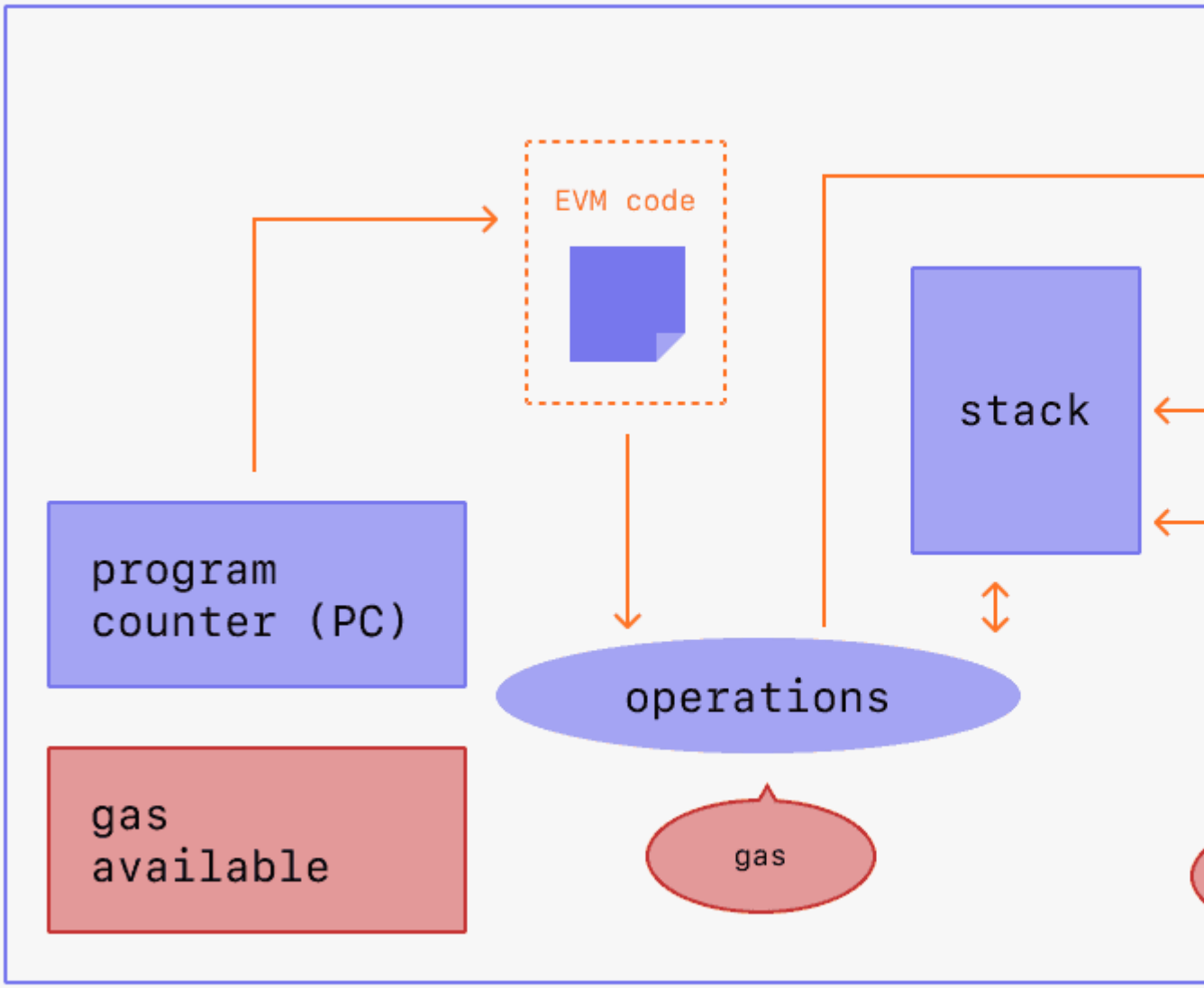


Diagram adapted from Ethereum EVM illustrated

Gas fees are paid in Ethereum's native currency, ether (ETH). Gas prices are denoted in gwei, which itself is a denomination of ETH - each gwei is equal to 0.000000001 ETH (10^{-9} ETH). For example, instead of saying that your gas costs 0.000000001 ether, you can say your gas costs 1 gwei. The word 'gwei' itself means 'giga-wei', and it is equal to 1,000,000,000 wei. Wei itself (named after Wei Dai, creator of b-money) is the smallest unit of ETH.

PRIOR TO THE LONDON UPGRADE

The way transaction fees on the Ethereum network were calculated changed with the London Upgrade of August 2021. Here is a recap of how things used to work:

Let's say Alice had to pay Bob 1 ETH. In the transaction, the gas limit is 21,000 units, and the gas price is 200 gwei.

Total fee would have been: `Gas units (limit) * Gas price per unit` i.e `21,000 * 200 = 4,200,000 gwei` or 0.0042 ETH

When Alice sent the money, 1.0042 ETH would be deducted from Alice's account. Bob would be credited 1.0000 ETH. Miner would receive 0.0042 ETH.

This video offers a concise overview of gas and why it exists:

AFTER THE LONDON UPGRADE

The London Upgrade was implemented on August 5th, 2021, to make transacting on Ethereum more predictable for users by overhauling Ethereum's transaction-fee-mechanism. The high-level benefits introduced by this change include better transaction fee estimation, generally quicker transaction inclusion, and offsetting the ETH issuance by burning a percentage of transaction fees.

Starting with the London network upgrade, every block has a base fee, the minimum price per unit of gas for inclusion in this block, calculated by the network based on demand for block space. As the base fee of the transaction fee is burnt, users are also expected to set a tip (priority fee) in their transactions. The tip compensates miners for executing and propagating user transactions in blocks and is expected to be set automatically by most wallets.

Calculating the total transaction fee works as follows: `Gas units (limit) * (Base fee + Tip)`

Let's say Jordan has to pay Taylor 1 ETH. In the transaction, the gas limit is 21,000 units and the base fee is 100 gwei. Jordan includes a tip of 10 gwei.

Using the formula above we can calculate this as `21,000 * (100 + 10) = 2,310,000 gwei` or 0.00231 ETH.

When Jordan sends the money, 1.00231 ETH will be deducted from Jordan's account. Taylor will be credited 1.0000 ETH. Miner receives the tip of 0.00021 ETH. Base fee of 0.0021 ETH is burned.

Additionally, Jordan can also set a max fee (`maxFeePerGas`) for the transaction. The difference between the max fee and the actual fee is refunded to Jordan, i.e. `refund = max fee - (base fee + priority fee)`. Jordan can set a maximum amount to pay for the transaction to execute and not worry about overpaying "beyond" the base fee when the transaction is executed.

Block size

Before the London Upgrade, Ethereum had fixed-sized blocks. In times of high network demand, these blocks operated at total capacity. As a result, users often had to wait for high demand to reduce to get included in a block, which led to a poor user experience.

The London Upgrade introduced variable-size blocks to Ethereum. Each block has a target size of 15 million gas, but the size of blocks will increase or decrease in accordance with network demand, up until the block limit of 30 million gas (2x the target block size). The protocol achieves an equilibrium block size of 15 million on average through the process of tâtonnement. This means if the block size is greater than the target block size, the protocol will increase the base fee for the following block. Similarly, the protocol will decrease the base fee if the block size is less than the target block size. The amount by which the base fee is adjusted is proportional to how far the current block size is from the target. More on blocks.

Base fee

Every block has a base fee which acts as a reserve price. To be eligible for inclusion in a block the offered price per gas must at least equal the base fee. The base fee is calculated independently of the current block

and is instead determined by the blocks before it - making transaction fees more predictable for users. When the block is mined this base fee is "burned", removing it from circulation.

The base fee is calculated by a formula that compares the size of the previous block (the amount of gas used for all the transactions) with the target size. The base fee will increase by a maximum of 12.5% per block if the target block size is exceeded. This exponential growth makes it economically non-viable for block size to remain high indefinitely.

Block Number	Included Gas	Fee Increase	Current Base Fee
1	15M	0%	100 gwei
2	30M	0%	100 gwei
3	30M	12.5%	112.5 gwei
4	30M	12.5%	126.6 gwei
5	30M	12.5%	142.4 gwei
6	30M	12.5%	160.2 gwei
7	30M	12.5%	180.2 gwei
8	30M	12.5%	202.7 gwei

Relative to the pre-London gas auction market, this transaction-fee-mechanism change causes fee prediction to be more reliable. Following the table above - to create a transaction on block number 9, a wallet will let the user know with certainty that the **maximum base fee** to be added to the next block is `current base fee * 112.5%` or `202.8 gwei * 112.5% = 228.1 gwei`.

It's also important to note it is unlikely we will see extended spikes of full blocks because of the speed at which the base fee increases proceeding a full block.

Block Number	Included Gas	Fee Increase	Current Base Fee
30	30M	12.5%	2705.6 gwei
...	...	12.5%	...
50	30M	12.5%	28531.3 gwei
...	...	12.5%	...
100	30M	12.5%	10302608.6 gwei

Priority fee (tips)

Before the London Upgrade, miners would receive the total gas fee from any transaction included in a block.

With the new base fee getting burned, the London Upgrade introduced a priority fee (tip) to incentivize miners to include a transaction in the block. Without tips, miners would find it economically viable to mine empty blocks, as they would receive the same block reward. Under normal conditions, a small tip provides miners a minimal incentive to include a transaction. For transactions that need to get preferentially executed ahead of other transactions in the same block, a higher tip will be necessary to attempt to outbid competing transactions.

Max fee

To execute a transaction on the network, users can specify a maximum limit they are willing to pay for their transaction to be executed. This optional parameter is known as the `maxFeePerGas`. For a transaction to be executed, the max fee must exceed the sum of the base fee and the tip. The transaction sender is refunded the difference between the max fee and the sum of the base fee and tip.

Calculating fees

One of the main benefits of the London upgrade is improving the user's experience when setting transaction fees. For wallets that support the upgrade, instead of explicitly stating how much you are willing to pay to get your transaction through, wallet providers will automatically set a recommended transaction fee (base fee + recommended priority fee) to reduce the amount of complexity burdened onto their users.

EIP-1559

The implementation of EIP-1559 in the London Upgrade made the transaction fee mechanism more complex than the previous gas price auction, but it has the advantage of making gas fees more predictable, resulting in a more efficient transaction fee market. Users can submit transactions with a `maxFeePerGas` corresponding to how much they are willing to pay for the transaction to be executing, knowing that they will not pay more than the market price for gas (`baseFeePerGas`), and get any extra, minus their tip, refunded.

This video explains EIP-1559 and the benefits it brings:

If you are interested, you can read the exact EIP-1559 specifications.

Continue down the rabbit hole with these EIP-1559 Resources.

WHY DO GAS FEES EXIST?

In short, gas fees help keep the Ethereum network secure. By requiring a fee for every computation executed on the network, we prevent bad actors from spamming the network. In order to avoid accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use. The fundamental unit of computation is "gas".

Although a transaction includes a limit, any gas not used in a transaction is returned to the user (i.e. `max fee - (base fee + tip)` is returned).

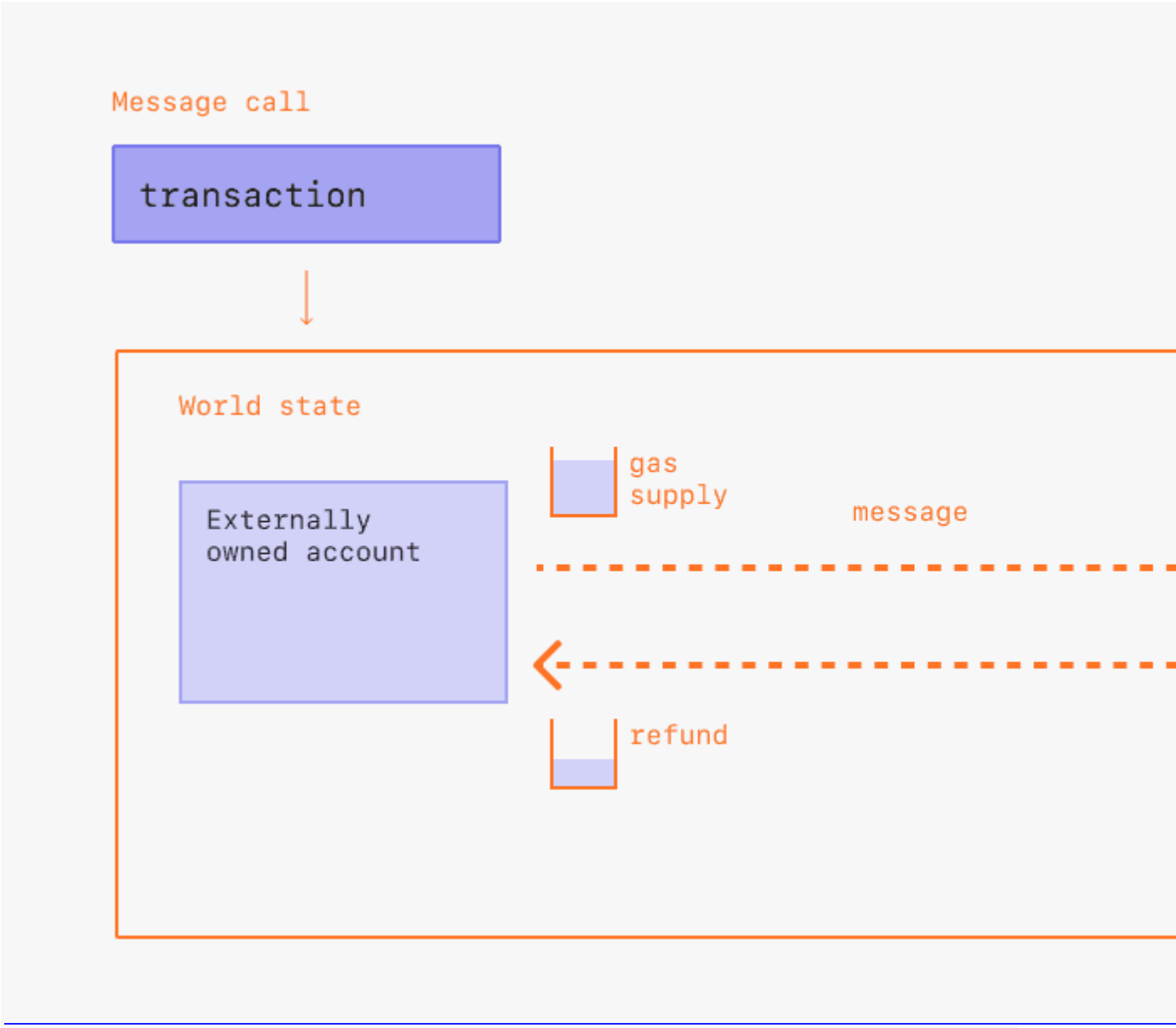


Diagram adapted from [Ethereum EVM illustrated]

WHAT IS GAS LIMIT?

Gas limit refers to the maximum amount of gas you are willing to consume on a transaction. More complicated transactions involving smart contracts require more computational work, so they require a higher gas limit than a simple payment. A standard ETH transfer requires a gas limit of 21,000 units of gas.

For example, if you put a gas limit of 50,000 for a simple ETH transfer, the EVM would consume 21,000, and you would get back the remaining 29,000. However, if you specify too little gas, for example, a gas limit of 20,000 for a simple ETH transfer, the EVM will consume your 20,000 gas units attempting to fulfill the transaction, but it will not complete. The EVM then reverts any changes, but since the miner has already done 20k gas units worth of work, that gas is consumed.

WHY CAN GAS FEES GET SO HIGH?

High gas fees are due to the popularity of Ethereum. Performing any operation on Ethereum requires consuming gas, and gas space is limited per block. Fees include calculations, storing or manipulating data, or transferring tokens, consuming different amounts of "gas" units. As dapp functionality grows more complex, the number of operations a smart contract performs also grows, meaning each transaction takes up more space of a limited size block. If there's too much demand, users must offer a higher tip amount to try and outbid other users' transactions. A higher tip can make it more likely that your transaction will get into the next block.

Gas price alone does not actually determine how much we have to pay for a particular transaction. To calculate the transaction fee, we have to multiply the gas used by the transaction fee, which is measured in gwei.

INITIATIVES TO REDUCE GAS COSTS

The Ethereum scalability upgrades should ultimately address some of the gas fee issues, which will, in turn, enable the platform to process thousands of transactions per second and scale globally.

Layer 2 scaling is a primary initiative to greatly improve gas costs, user experience and scalability. More on layer 2 scaling.

The new proof-of-stake model, introduced on the Beacon Chain, should reduce high power consumption and reliance on specialized hardware. This chain will allow the decentralized Ethereum network to agree and keep the network secure, while limiting energy consumption by instead requiring a financial commitment.

Anyone with at least 32 ETH can stake them and become a validator responsible for processing transactions, validating blocks, and proposing new blocks to add to the chain. Users who have less than 32 ETH can join staking pools.

OPCODES FOR THE EVM

Stack	Name	Gas	Initial Stack	Resulting Stack
00	STOP	0		
01	ADD	3	a, b	a + b
02	MUL	5	a, b	a * b
03	SUB	3	a, b	a - b
04	DIV	5	a, b	a // b
05	SDIV	5	a, b	a // b
06	MOD	5	a, b	a % b
07	SMOD	5	a, b	a % b
08	ADDMOD	8	a, b, N	(a + b) % N
09	MULMOD	8	a, b, N	(a * b) % N
0A	EXP	A1	a, b	a ** b

Stack	Name	Gas	Initial Stack	Resulting Stack
0B	SIGNEXTEND	5	b, x	SIGNEXTEND(x, b)
0C-0F	invalid			
10	LT	3	a, b	a < b
11	GT	3	a, b	a > b
12	SLT	3	a, b	a < b
13	SGT	3	a, b	a > b
14	EQ	3	a, b	a == b
15	ISZERO	3	a	a == 0
16	AND	3	a, b	a && b
17	OR	3	a, b	a || b
18	XOR	3	a, b	a ^ b
19	NOT	3	a	~a
1A	BYTE	3	i, x	(x >> (248 - i * 8)) && 0xFF
1B	SHL	3	shift, val	val << shift
1C	SHR	3	shift, val	val >> shift
1D	SAR	3	shift, val	val >> shift
1E-1F	invalid			
20	SHA3	A2	ost, len	keccak256(mem[ost:ost+le
21-2F	invalid			
30	ADDRESS	2	.	address(this)
31	BALANCE	A5	addr	addr.balance
32	ORIGIN	2	.	tx.origin
33	CALLER	2	.	msg.sender
34	CALLVALUE	2	.	msg.value
35	CALLDATALOAD	3	idx	msg.data[idx:idx+32]
36	CALLDATASIZE	2	.	len(msg.data)
37	CALLDATACOPY	A3	dstOst, ost, len	.
38	CODESIZE	2	.	len(this.code)
39	CODECOPY	A3	dstOst, ost, len	.

Stack	Name	Gas	Initial Stack	Resulting Stack
3A	GASPRICE	2	.	tx.gasprice
3B	EXTCODESIZE	A5	addr	len(addr.code)
3C	EXTCODECOPY	A4	addr, dstOst, ost, len	.
3D	RETURNDATASIZE	2	.	size
3E	RETURNDATACOPY	A3	dstOst, ost, len	.
3F	EXTCODEHASH	A5	addr	hash
40	BLOCKHASH	20	blockNum	blockHash(blockNum)
41	COINBASE	2	.	block.coinbase
42	TIMESTAMP	2	.	block.timestamp
43	NUMBER	2	.	block.number
44	DIFFICULTY	2	.	block.difficulty
45	GASLIMIT	2	.	block.gaslimit
46	CHAINID	2	.	chain_id
47	SELFBALANCE	5	.	address(this).balance
48	BASEFEE	2	.	block.basefee
49-4F	invalid			
50	POP	2	_anon	.
51	MLOAD	3*	ost	mem[ost:ost+32]
52	MSTORE	3*	ost, val	.
53	MSTORE8	3*	ost, val	.
54	SLOAD	A6	key	storage[key]
55	SSTORE	A7	key, val	.
56	JUMP	8	dst	.
57	JUMPI	10	dst, condition	.
58	PC	2	.	$pc
59	MSIZE	2	.	len(mem)
5A	GAS	2	.	gasRemaining
5B	JUMPDEST	1		

Stack	Name	Gas	Initial Stack	Resulting Stack
5C-5F	invalid			
60	PUSH1	3	.	uint8
61	PUSH2	3	.	uint16
62	PUSH3	3	.	uint24
63	PUSH4	3	.	uint32
64	PUSH5	3	.	uint40
65	PUSH6	3	.	uint48
66	PUSH7	3	.	uint56
67	PUSH8	3	.	uint64
68	PUSH9	3	.	uint72
69	PUSH10	3	.	uint80
6A	PUSH11	3	.	uint88
6B	PUSH12	3	.	uint96
6C	PUSH13	3	.	uint104
6D	PUSH14	3	.	uint112
6E	PUSH15	3	.	uint120
6F	PUSH16	3	.	uint128
70	PUSH17	3	.	uint136
71	PUSH18	3	.	uint144
72	PUSH19	3	.	uint152
73	PUSH20	3	.	uint160
74	PUSH21	3	.	uint168
75	PUSH22	3	.	uint176
76	PUSH23	3	.	uint184
77	PUSH24	3	.	uint192
78	PUSH25	3	.	uint200
79	PUSH26	3	.	uint208
7A	PUSH27	3	.	uint216
7B	PUSH28	3	.	uint224
7C	PUSH29	3	.	uint232

Stack	Name	Gas	Initial Stack	Resulting Stack
7D	PUSH30	3	.	uint240
7E	PUSH31	3	.	uint248
7F	PUSH32	3	.	uint256
80	DUP1	3	a	a, a
81	DUP2	3	_, a	a, _, a
82	DUP3	3	_, _, a	a, _, _, a
83	DUP4	3	_, _, _, a	a, _, _, _, a
84	DUP5	3	..., a	a, ..., a
85	DUP6	3	..., a	a, ..., a
86	DUP7	3	..., a	a, ..., a
87	DUP8	3	..., a	a, ..., a
88	DUP9	3	..., a	a, ..., a
89	DUP10	3	..., a	a, ..., a
8A	DUP11	3	..., a	a, ..., a
8B	DUP12	3	..., a	a, ..., a
8C	DUP13	3	..., a	a, ..., a
8D	DUP14	3	..., a	a, ..., a
8E	DUP15	3	..., a	a, ..., a
8F	DUP16	3	..., a	a, ..., a
90	SWAP1	3	a, b	b, a
91	SWAP2	3	a, _, b	b, _, a
92	SWAP3	3	a, _, _, b	b, _, _, a
93	SWAP4	3	a, _, _, _, b	b, _, _, _, a
94	SWAP5	3	a, ..., b	b, ..., a
95	SWAP6	3	a, ..., b	b, ..., a
96	SWAP7	3	a, ..., b	b, ..., a
97	SWAP8	3	a, ..., b	b, ..., a
98	SWAP9	3	a, ..., b	b, ..., a
99	SWAP10	3	a, ..., b	b, ..., a
9A	SWAP11	3	a, ..., b	b, ..., a
9B	SWAP12	3	a, ..., b	b, ..., a

Stack	Name	Gas	Initial Stack	Resulting Stack
9C	SWAP13	3	a, ..., b	b, ..., a
9D	SWAP14	3	a, ..., b	b, ..., a
9E	SWAP15	3	a, ..., b	b, ..., a
9F	SWAP16	3	a, ..., b	b, ..., a
A0	LOG0	A8	ost, len	.
A1	LOG1	A8	ost, len, topic0	.
A2	LOG2	A8	ost, len, topic0, topic1	.
A3	LOG3	A8	ost, len, topic0, topic1, topic2	.
A4	LOG4	A8	ost, len, topic0, topic1, topic2, topic3	.
A5-EF	invalid			
F0	CREATE	A9	val, ost, len	addr
F1	CALL	AA	gas, addr, val, argOst, argLen, retOst, retLen	success
F2	CALLCODE	AA	gas, addr, val, argOst, argLen, retOst, retLen	success
F3	RETURN	0*	ost, len	.
F4	DELEGATECALL	AA	gas, addr, argOst, argLen, retOst, retLen	success
F5	CREATE2	A9	val, ost, len, salt	addr
F6-F9	invalid			
FA	STATICCALL	AA	gas, addr, argOst, argLen, retOst, retLen	success
FB-FC	invalid			
FD	REVERT	0*	ost, len	.
FE	INVALID	AF		
FF	SELFDESTRUCT	AB	addr	

Explain about EVM applications

Explain about the Ethereum blockchain data structure.

Explain working of EVM.

Ans: The EVM Constantly Checks for Transactions

State machines (machines with memory) can be thought of as beings who never sleep. As a state machine, the EVM has a constant history of all transactions within their memory banks, leading all the way back to the very first transaction. Unlike people, who have to deal with imperfect memory, a computer's state (as it exists today) is the specific outcome of every single state-change that has taken place inside that machine since it was first switched on.

Creating a Common Machine

Transactions, therefore, represent a kind of machine narrative—a computationally valid arc between one state and another. As Gavin Wood's Ethereum Yellow Paper says: There exist far more invalid state changes than valid state changes. Invalid state changes might, e.g., be things such as reducing an account balance without an equal and opposite increase elsewhere. A valid state transition is one which comes about through a transaction.3 As time advances, the system (as in Bitcoin) seeks to create a trustworthy history for ensuring that each subsequent state change is legitimate, and not an instruction inserted by a bad actor

Cryptographic Hashing

Hash Algorithms

g, the purpose of hash functions, in the context of a blockchain, is to compare large datasets quickly and evaluate whether their contents are similar. A oneway algorithm processes the entire block's transactions into 32 bytes of data—a hash, or string, of letters and numbers that contains no discernible information about the transactions within. The hash creates an unmistakable signature for a block, allowing the next block to build on top of it. Unlike the ciphertext that results from encryption, which can be decrypted, the result of a hash cannot be "un-hashed."

Explain about Global Special Variables, Units, and Functions in solidity programming.

ANS: Global special variables can be called by any Solidity smart contract on the EVM; they're built in to the language. Most of them return information about the Ethereum chain. Units of time and ether are also globally available. Literal numbers can take a suffix of wei, finney, szabo or ether and will auto-convert between subdenominations of Ether. Ether currency numbers without a suffix are assumed to be Wei. Time-related suffixes can be used after literal numbers to convert between units of time. Here, seconds are the base unit, and units are treated as general units. Owing to the existence of leap years, be careful when using these suffixes to calculate time, as not all years have 365 days, and not days have 24 hours

1 == 1 seconds

1 minutes == 60 seconds

1 hours == 60 minutes

1 days == 24 hours

1 weeks = 7 days

1 years = 365 days

Block and Transaction Properties

• block.blockhash(uintblockNumber) returns (bytes32): Hash of the given block, works for only the 256 most recent blocks

• block.coinbase (address): Current block miner’s address

• block.difficulty (uint): Current block difficulty

• block.gaslimit (uint): Current block gas limit

• block.number (uint): Current block number

• block.timestamp (uint): Current block timestamp

• msg.data (bytes): Complete call data

• msg.gas (uint): Remaining gas
• msg.sender (address): Sender of the message (current call)
• msg.sig (bytes4): First 4 bytes of the call data (function identifier)
• msg.value (uint): Number of wei sent with the message
• now (uint): Current block timestamp (alias for block. timestamp)
• tx.gasprice (uint): Gas price of the transaction
• tx.origin (address): Sender of the transaction (full call chain)

Operators Cheat Sheet

Precedence	Description	Operator
1	Postfix increment and decrement	++, --
	Function-like call	<func>(<args...>)
	Array subscripting	<array>[<index>]
	Member access	<object>.<member>
	Parentheses	(<statement>)
2	Prefix increment and decrement	++, --
	Unary plus and minus	+, -
	Unary operations	delete
	Logical NOT	!
	Bitwise NOT	~
3	Exponentiation	**
4	Multiplication, division, and modulo	*, /, %
5	Addition and subtraction	+, -
6	Bitwise shift operators	<<, >>
7	Bitwise AND	&

Precedence	Description	Operator
8	Bitwise XOR	^
9	Bitwise OR	|
10	Inequality operators	<, >, <=, >=
11	Equality operator, does-not-equal operator	==, !=
12	Logical AND	&&
13	Logical OR	||
14	Ternary operator	<conditional> ? <if-true> : <if-false>
15	Assignment operators	=, |=, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=
16	Comma operator	,

Global Functions

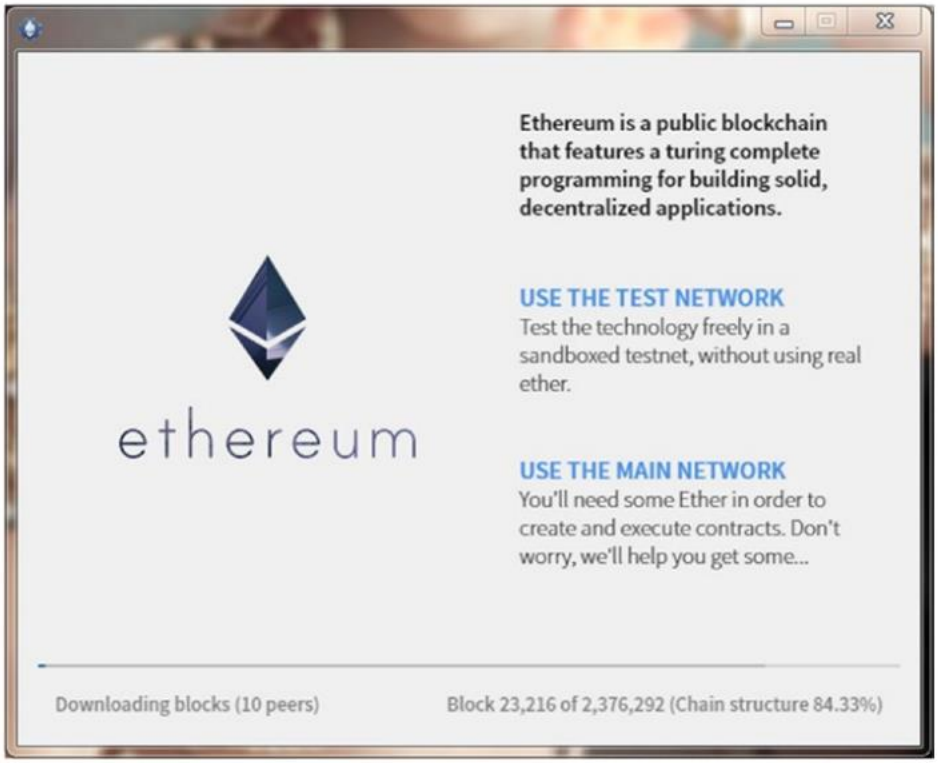
- `keccak256(...) returns (bytes32)`: Computes the Ethereum-SHA-3 (Keccak-256) hash of the (tightly packed) arguments
- `sha3(...) returns (bytes32)`: An alias to `keccak256()`
- `sha256(...) returns (bytes32)`: Computes the SHA-256 hash of the (tightly packed) arguments. "Tightly packed" means that the arguments are concatenated without padding. To see how
- `ripemd160(...) returns (bytes20)`: Computes the RIPEMD-160 hash of the (tightly packed) arguments
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: Recovers address associated with the public key from elliptic curve signature, returns 0 on error
- `addmod(uint x, uint y, uint k) returns (uint)`: Computes `(x + y) % k`, where the addition is performed with arbitrary precision and does not wrap around at `2**256`
- `mulmod(uint x, uint y, uint k) returns (uint)`: Computes `(x * y) % k`, where the multiplication is performed with arbitrary precision and does not wrap around at `2**256`
- `this (current contract's type)`: The current contract, explicitly convertible to its address

Explain the Mist browser configuration steps.

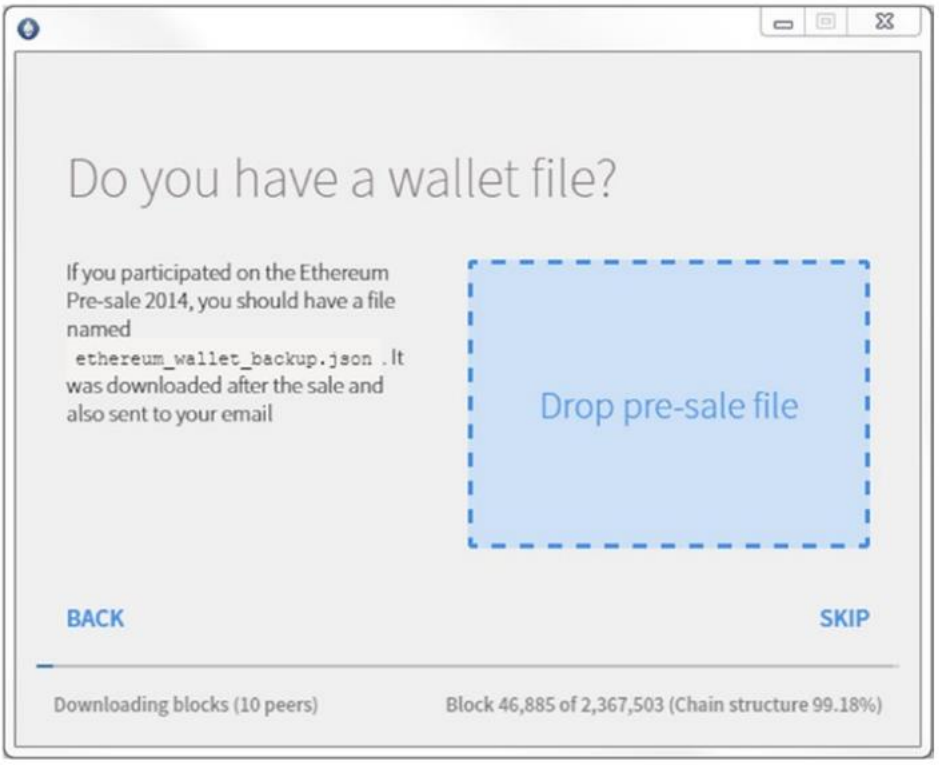
First, download Mist from https://github.com/ethereum/mist/releases,

Configuring Mist

After you download and open the installer, you'll see a welcome screen like the one in Figure 2-3. (There are some of those big promises from Chapter 1!)



Next you'll see the screen shown in Figure 2-4, which you can skip—unless you participated in the Ethereum crowdsale back in 2014. In that case, follow those instructions to redeem your ether.



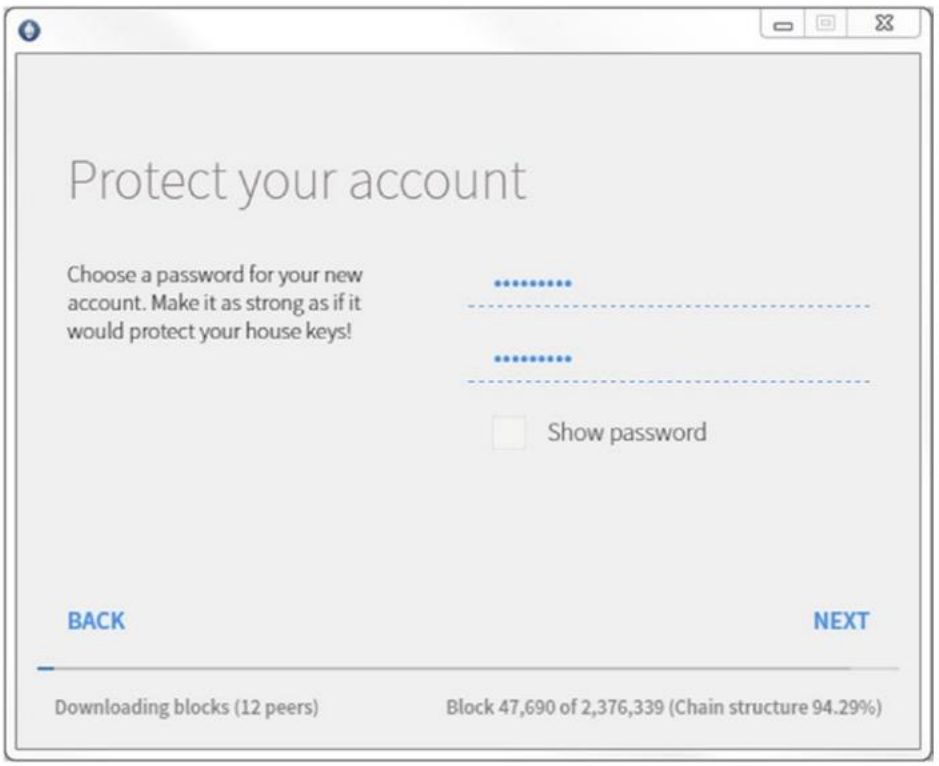


Figure 2-5. *Next, choose a password*

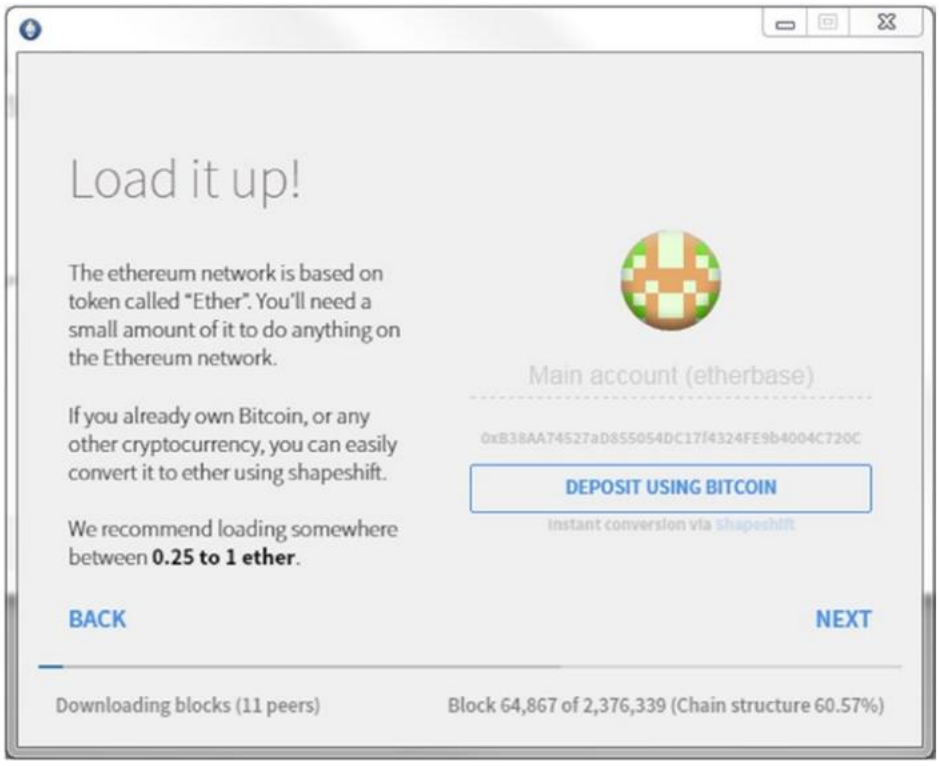


Figure 2-6. *Here you can see the new address. You can also deposit bitcoins to be converted into ether by the Shapeshift.io API.*

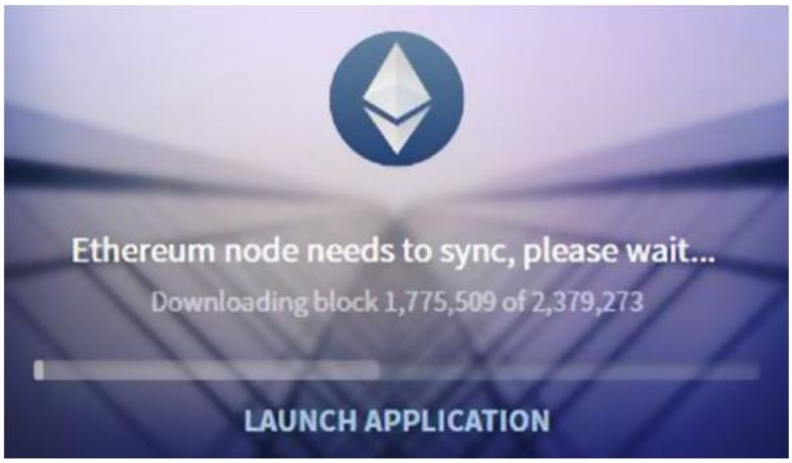


Figure 2-7. *This will take a while. Your new account will show up when it's done.*

1) What is Gas. Explain the working of gas.
   Gas is a unit of work used to measure how computationally expensive an Ethereum operation will be. Gas costs are paid with small amounts of ether.

Gas Specifics

Let's review some details about working with gas:

- Unfortunately, the term *gas* creates some confusion. Every transaction requires a `STARTGAS` value. This value is referred to as `gasLimit` in the Yellow Paper and often just as gas in Geth and Web3.js.
- Every transaction also requires the user to specify a gas price.
- The amount stipulated in `STARTGAS`, multiplied by the gas price, is held in escrow while your transaction executes.
- If the gas price you offer for a transaction is too low, nodes won't process your transaction, and it will sit unprocessed on the network.
- If your gas price is acceptable to the network, but the gas cost runs over what's available in your wallet balance, the transaction fails and is rolled back; this failed transaction is recorded to the blockchain, and you get a refund of any `STARTGAS` not used in the transaction.
- Using excessive `STARTGAS` does not cause your transactions to be processed more quickly, and in some cases may make your transaction less appealing to miners.[8]

How Gas Relates to Scaling the System

If you send a computationally difficult set of instructions to the EVM, the only person this hurts is you. The work will spend your ether, and stop when the ether you allocated to the transaction runs out. It has no effect on anyone else's transactions. There is no way to jam up the EVM without paying a lot, in the form of transaction fees, to do it.

Scaling is handled in a de facto way through the gas fee system. Miners are free to choose the transactions that pay the highest fee rates, and can also choose the block gas limit collectively. The gas limit determines how much computation can happen (and how much storage can be allocated) per block.

2) Write a short note on Ethereum account.
Externally owned accounts
Contracts accounts

Externally Owned Accounts

An *externally owned account* (EOA) is also known as an account controlled by a pair of private keys, which may be held by a person or an external server. These accounts cannot hold EVM code. Characteristics of an EOA include the following:

- Contains a balance of ether
- Capable of sending transactions
- Controlled by the account's private keys
- Has no code associated with it
- A key/value database contained in each account, where keys and values are both 32-byte strings

UIT III
1)
2

Contract Accounts

Contract accounts are not controlled by humans. They store instructions and are activated by external accounts or other contract accounts. Contract accounts have the following characteristics:

- Have an ether balance
- Hold some contract code in memory
- Can be triggered by humans (sending a transaction) or other contracts sending a message
- When executed, can perform complex operations
- Have their own persistent state and can call other contracts
- Have no owner after being released to the EVM
- A key/value database contained in each account, where keys and values are both 32-byte strings

UNIT III

hyperledger fabric.

Hyperledger Fabric is an open-source platform for building distributed ledger solutions, with a modular architecture that delivers high degrees of confidentiality, flexibility, resiliency, and scalability. This enables solutions developed with fabric to be adapted for any industry. This is a private and confidential blockchain framework managed by the Linux Foundation.

Components:

- Hyperledger fabric is an enterprise-level permission blockchain network. It is made up of various unique organizations or members that interact with each other to serve a specific purpose. For example, these organizations can be a bank, financial institution, or a supply chain network. Each organization is identified and they have a fabric certificate authority. These organizations are called members.
- Each member of the fabric can set up one or more authorized peers to participate in the network using the fabric certificate authority. All of these peers must be authorized properly.
- There is a client-side application connected to the network written with the software development kit (SDK) of any particular programming language.

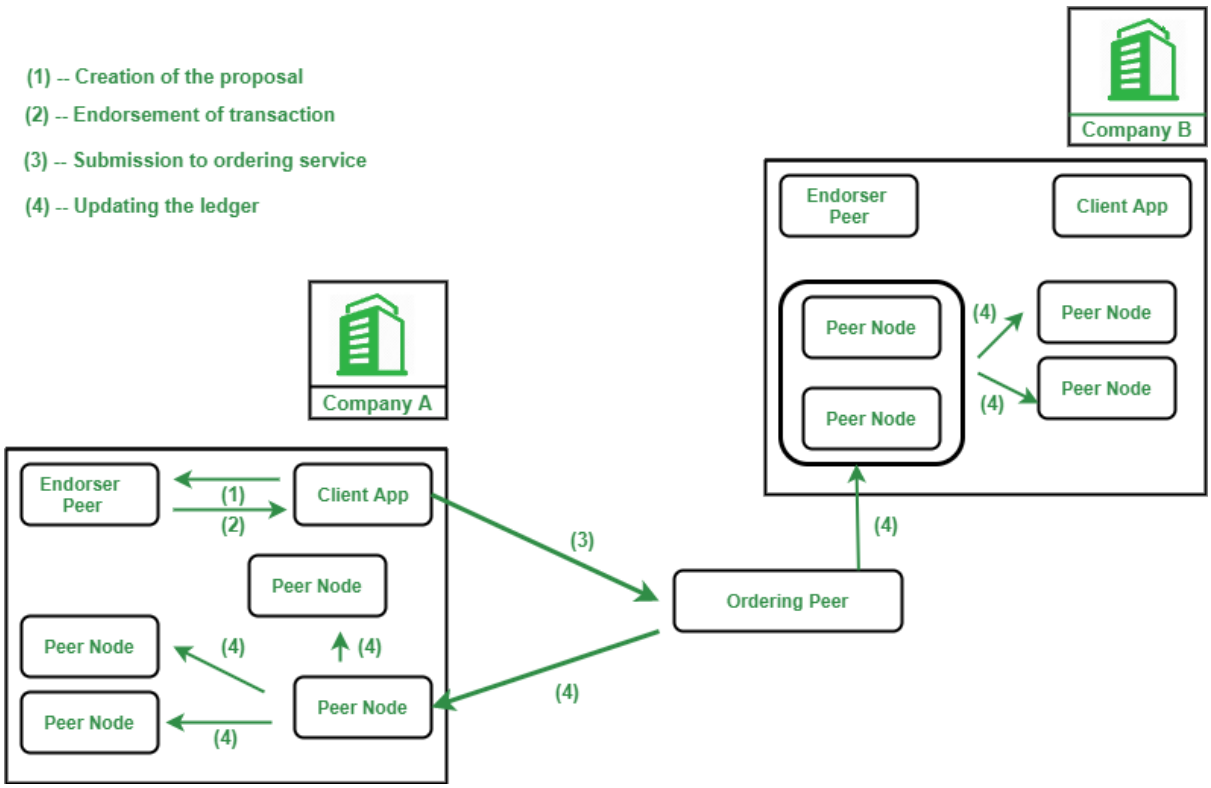
Workflow:

For each and every transaction in the fabric, the following steps are followed-

1. **Creation of the proposal:** Imagine a deal between a smartphone manufacturer company and a smartphone dealership. The transaction begins when a member organization proposes or invokes a transaction request with the help of the client application or portal. Then the client application sends the proposal to peers in each organization for endorsement.
2. **Endorsement of the transaction:** After the proposal reaches the endorser peers (peers in each organization for endorsement of a proposal) the peer checks the fabric

certificate authority of the requesting member and other details that are needed no authenticate the transaction. Then it executes the chain code (a piece of code that is written in one of the supported languages such as Go or Java) and returns a response. This response indicates the approval or rejection of the following transaction. The response is carried out to the client.

3. **Submission to ordering service:** After receiving the endorsement output, the approved transactions are sent to the ordering service by the client-side application. The peer responsible for the ordering service includes the transaction into a specific block and sends it to the peer nodes of different members of the network.
4. **Updating the ledger:** After receiving this block the peer nodes of such organizations update their local ledger with this block. Hence the new transactions are now committed.



1)) Explain the steps to create the token on testnet.

Launch the Mist wallet on your desktop computer. Navigate to the Develop menu of the Mist wallet, and you should find a Network menu that allows you to select the testnet, as shown in Figure 5-2.

Figure 5-2. *Connecting to the testnet*

Once you're using the testnet, you should see an alert in the Mist browser highlighted in red, as shown in Figure 5-3.

Getting Test Ether from the Faucet

In Ethereum, it is trivial to set up a *faucet* that spouts faux ether you can use on the Ropsten testnet. In this section, you won't set up your own faucet, but will use a third-party faucet pictured in Figure 5-4 and available at `http://faucet.ropsten.be:3001/`.

You'll also find an up-to-date shortlink to this faucet at `http://faucet.eth.guide`. Follow these steps to receive testnet ether from the faucet:

1. After making sure your Mist wallet is on the testnet with the steps above, create an address if you haven't already. Copy this long hexidecial address (beignning with 0x...) to your system clipboard and then paste it into the address field:
2. To get ether, click the button entitled "send me 1 test ether."

Figure 5-4. *The Ethereum testnet comes with the facility for dispensing test ether that can be used while writing or debugging contracts*

If you'd like to test out the transfer of ether, you can do that by transferring test ether from one address in your Mist wallet to another address in your Mist wallet. To do this: Go back to Mist and create a new wallet address in the Home view. You can use the Send dialog box to send ether from one of your wallet addresses to another. It will be approximately the same speed whether you are sending ether to yourself or to someone on the other side of the world; that's the beauty of distributed systems.

The testnet also has a blockchain explorer, where you can see all your testnet transactions. Simply enter one your testnet Mist addresses into the search box at the following testnet blockchain explorer, and you'll see all its transactions listed: https://testnet.etherscan.io/

Now that we've messed around with test ether on the Ropsten chain, let's take the next step toward making your own ether subcurrency, also known as a token, with zero coding.

Explain the Steps to develop an Ethereum Smart Contract.

Step 1: Create a wallet at meta-mask

Install MetaMask in your Chrome browser and enable it. Once it is installed, click on its icon on the top right of the browser page. Clicking on it will open it in a new tab of the browser.

Click on “Create Wallet” and agree to the terms and conditions by clicking “I agree” to proceed further. It will ask you to create a password.

After you create a password, it will send you a secret backup phrase used for backing up and restoring the account. Do not disclose it or share it with someone, as this phrase can take away your Ethers.

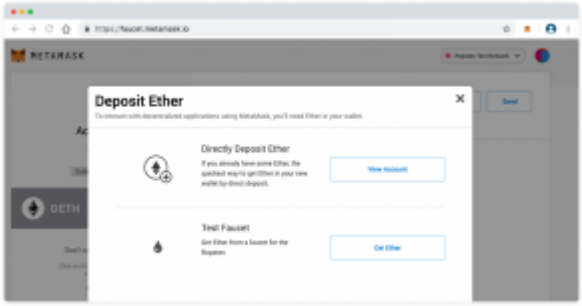
Step 2: Select any one test network

You might also find the following test networks in your MetaMask wallet:

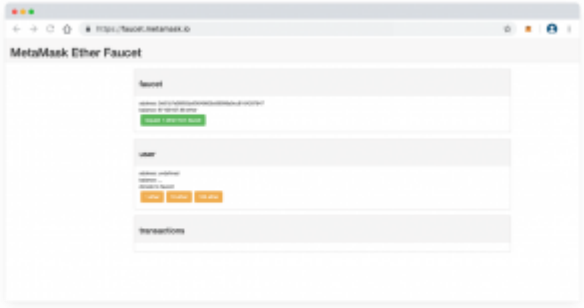
- Robsten Test Network
- Kovan Test Network
- Rinkeby Test Network
- Goerli Test Network

- **Step 3: Add some dummy Ethers to your wallet**
- In case you want to test the smart contract, you must have some dummy ethers in your MetaMask wallet.
- For example, if you want to test a contract using the Robsten test network, select it and you will find 0 ETH as the initial balance in your account.

- To add dummy ethers, click on the “Deposit” and “Get Ether” buttons under Test Faucet.



To proceed, you need to click “request one ether from the faucet,” and 1 ETH will be added to your wallet. You can add as many Ethers you want to the test network.



Step 4: Use editor remix to write the smart contract in Solidity

We will use Remix Browser IDE to write our Solidity code. The remix is the best option for writing smart contracts as it comes with a handful of features and offers a comprehensive development experience.

It is usually used for writing smaller-sized contracts. Remix’s features include:

- Warnings like gas cost, unsafe code, checks for overlapping variable names, whether functions can be constant or not
- Syntax and error highlighting
- Functions with injected Web3 objects
- Static analysis
- Integrated debugger
- Integrated testing and deployment environment
- Deploy directly to Mist or MetaMask

Let’s start writing smart contract code by visiting https://remix.ethereum.org.

Step 5: Create a .sol extension file

Open Remix Browser and click on the plus icon on the top left side, next to the browser to create a .sol extension file.

Step 6: A sample smart contract code to create ERC20 tokens

ERC20.sol is a standard template for ERC20 tokens.

```
pragma solidity ^0.4.0;
```

```
import "./ERC20.sol";
contract myToken is ERC20{
mapping(address =&gt;uint256) public amount;
uint256 totalAmount;
string tokenName;
string tokenSymbol;
uint256 decimal;
constructor() public{
totalAmount = 10000 * 10**18;
amount[msg.sender]=totalAmount;
tokenName="Mytoken";
tokenSymbol="Mytoken";
decimal=18;
}
function totalSupply() public view returns(uint256){
return totalAmount;
}
function balanceOf(address to_who) public view
returns(uint256){
return amount[to_who];
}
function transfer(address to_a,uint256 _value) public
returns(bool){
require(_value&lt;=amount[msg.sender]);
amount[msg.sender]=amount[msg.sender]-_value;
amount[to_a]=amount[to_a]+_value;
return true;
}
}
```

Select a version of the compiler from Remix to compile the solidity Ethereum smart contract code.

Step 7: Deploy your contract

Deploy the smart contract at the Ethereum test network by pressing the deploy button at the Remix window's right-hand side.

Wait until the transaction is complete.

After the transaction commits successfully, the address of the smart contract would be visible at the right-hand side of the remix
window.

At first, all the ERC20 tokens will be stored in the wallet of a user who is deploying the smart contract.
To check the tokens in your wallet, go to the metamask window, click add tokens, enter the smart contract address and click ok. You would be able to see the number of tokens there.

Hyperledger

Hyperledger is an open source project created to support the development of blockchain-based distributed ledgers. Hyperledger consists of a collaborative effort to create the needed frameworks, standards, tools and libraries to build blockchains and related applications.

Since Hyperledger's creation by the Linux Foundation in 2016, the project has had contributions from organizations such as IBM and Intel, Samsung, Microsoft, Visa, American Express and blockchain startups such as Blockforce. In all, the collaboration includes banking, supply chain management, internet of things (IoT), manufacturing and production-based fields.

Hyperledger acts as a hub for different distributed ledger frameworks and libraries. With this, a business could use one of Hyperledger's frameworks, for example, to improve the efficiency, performance and transactions in their business processes.

Hyperledger works by providing the needed infrastructure and standards for developing blockchain systems and applications. Developers use Hyperledger Greenhouse (the frameworks and tools that make up Hyperledger) to develop business blockchain projects. Network participants are all known to each other and can participate in consensus-making processes.

Hyperledger-based technology works using these layers:

- A consensus layer, which makes an agreement on order and confirms if the transactions in a block are correct.
- A smart contract layer, which processes and authorizes transaction requests
- A communication layer, which manages peer-to-peer (P2P) message transport.
- An API, which allows other applications to communicate with the blockchain.
- Identity management services, which validates the identities of users and systems.

Notable frameworks: Hyperledger Fabric and Sawtooth

Two of the most notable Hyperledger frameworks include Hyperledger Fabric and Sawtooth.

Hyperledger Fabric

This is one of the most popular projects in Hyperledger. It is a permissioned blockchain infrastructure used to build blockchain-based products, software and applications. Hyperledger Fabric was made in cooperation with IBM and Digital Asset. It provides a modular architecture that defines roles between nodes, execution of smart contracts and configurable consensus services. Features of Fabric include the use of smart contracts, as well as pluggable Hyperledger Fabric consensus protocols. Fabric also supports different programming languages through the installation of modules. Hyperledger Fabric is used with integration projects that need a distributed ledger.

Hyperledger Sawtooth

This is a permissioned modular blockchain platform contributed by Intel. Organizations use Sawtooth to deploy, run and build distributed ledgers. It can help businesses that have a difficult time working with blockchain technology. Sawtooth features include: Dynamic Consensus, Transaction Families, Proof of Elapsed Time (a type of consensus algorithm), Parallel Transaction Execution (which allows the creation of individual chains) and Private Transactions. It also supports Ethereum smart contracts. Software development kits (SDKs) for Python, Go, JavaScript, Rust, Java, and C++ are also available. Sawtooth is meant for businesses that need a permissioned and modular blockchain platform.

Other Hyperledger tools and projects

Hyperledger Fabric and Sawtooth are not the only two projects Hyperledger has. Hyperledger offers multiple projects and tools currently active or under incubation, meaning they require certain exit criteria before being declared active and production-ready. Some of these projects include:

- **Hyperledger Iroha.** A blockchain framework used to integrate with existing networks. Iroha has a modular design, control-based access, access to many libraries, as well as asset and identity management. It is used in fields such as financial services, healthcare and education.
- **Hyperledger Indy.** A framework made for decentralized identities. It comes with components, tool sets and libraries. It also includes self-sovereignty, which securely stores all identity-based documentation.
- **Hyperledger Besu.** An open source Ethereum codebase that can run on private permissioned platforms or the Ethereum public network. It features the Ethereum Virtual Machine (EVM), consensus algorithms, user-facing APIs and monitoring.

- **Hyperledger Caliper.** A blockchain benchmark tool. Caliper is used to evaluate the performance of blockchain implementations. However, it doesn't come with predefined standards because blockchain implementations may all require different sets of standards.
- **Hyperledger Explorer.** A dashboard utility tool that allows a user to monitor, search and maintain blockchain and related data. With it, an organization can check nodes, blocks, transactions and smart contracts. It also allows users to make code changes.
- **Hyperledger Cello.** A blockchain-as-a-service toolkit used to create, terminate and manage blockchain services.
- **Hyperledger Burrow.** A permissioned Ethereum smart contract blockchain node. This handles transactions and smart contract code execution on the EVM.

History and mission of Hyperledger

The Linux Foundation announced the creation of the Hyperledger Project in 2015, one year prior to its release. Brian Behlendorf was appointed the position of executive director. Behlendorf stated that the Hyperledger project would never build its own cryptocurrency.

In 2016, the project also started to accept proposals for incubation of codebases and other core element technologies. Two of the initial blockchain framework codebases accepted were Hyperledger Fabric and libconsensus. Later, Intel's distributed ledger, Sawtooth, was incubated.

In 2018, the production-ready Sawtooth 1.0 was added. In 2019, a long-term-support version of Hyperledger Fabric was announced.

Smart Contracts and Tokens:

Smart contracts are potentially one of the most useful tools associated with blockchain, and they can enable the transfer of everything from bitcoin and fiat currency to goods transported around the world. Here's what they do and why they're likely to gain traction. Smart contracts are self-executing, business automation applications that run on a decentralized network such as blockchain.

And because they're able to remove administrative overhead, smart contracts are one of most attractive features associated with blockchain technology. While blockchain acts as a kind of database, confirming that transactions have taken place, smart contracts execute pre-determined conditions; think about a smart contract as a computer executing on "if/then," or conditional, programming.

Understanding tokens and smart contracts

For example, an insurance company could use smart contracts to automate the release of claim money based on events such as large-scale floods, hurricanes or droughts. Or, once a

cargo shipment reaches a port of entry and IoT sensors inside the container confirm the contents have been unopened and remained stored properly throughout the journey, a bill of lading can automatically be issued.

Smart contracts are also the basis for the transference of cryptocurrency and digital tokens (in essence, a digital representation of a physical asset or utility). For example, Ethereum blockchain's ERC-20 and ERC-721 tokens are themselves smart contracts.

But not all smart contracts are tokens, according to Martha Bennett, a principal analyst at Forrester Research. "You can have smart contracts running on Ethereum that trigger an action based on a condition without an ERC-20 or ERC-721 token involved," she said.

Smart contracts can govern the transference of other cryptocurrencies, such as bitcoin. Once payment is verified, bitcoin can change hands from seller to buyer.

Most enterprise blockchain networks don't use tokens, Bennett pointed out. In those that do, the rules in smart contracts govern how tokens get allocated and define the conditions of transfer.

"That still doesn't mean the token is the smart contract - it all depends on how the token has been constructed," Bennett said. "And tokens don't have to be about economic value; a token can simply be something you hold that gives you the right to vote on a decision; casting your token means you've voted, and can't vote on this decision again – no economic value associated."

How smart contracts work

Smart contracts work by following simple “if/when…then…” statements that are written into code on a blockchain. A network of computers executes the actions when predetermined conditions have been met and verified. These actions could include releasing funds to the appropriate parties, registering a vehicle, sending notifications, or issuing a ticket. The blockchain is then updated when the transaction is completed. That means the transaction cannot be changed, and only parties who have been granted permission can see the results.

Within a smart contract, there can be as many stipulations as needed to satisfy the participants that the task will be completed satisfactorily. To establish the terms, participants must determine how transactions and their data are represented on the blockchain, agree on the “if/when...then…” rules that govern those transactions, explore all possible exceptions, and define a framework for resolving disputes.

Then the smart contract can be programmed by a developer – although increasingly, organizations that use blockchain for business provide templates, web interfaces, and other online tools to simplify structuring smart contracts.

Benefits of smart contracts

Speed, efficiency and accuracy

Once a condition is met, the contract is executed immediately. Because smart contracts are digital and automated, there’s no paperwork to process and no time spent reconciling errors that often result from manually filling in documents.

Trust and transparency

Because there's no third party involved, and because encrypted records of transactions are shared across participants, there's no need to question whether information has been altered for personal benefit.

Security

Blockchain transaction records are encrypted, which makes them very hard to hack. Moreover, because each record is connected to the previous and subsequent records on a distributed ledger, hackers would have to alter the entire chain to change a single record.

Savings

Smart contracts remove the need for intermediaries to handle transactions and, by extension, their associated time delays and fees.

Applications of smart contracts

Safeguarding the efficacy of medications

Sonoco and IBM are working to reduce issues in the transport of lifesaving medications by increasing supply chain transparency. Powered by IBM Blockchain Transparent Supply, Pharma Portal is a blockchain-based platform that tracks temperature-controlled pharmaceuticals through the supply chain to provide trusted, reliable and accurate data across multiple parties.

Increasing trust in retailer-supplier relationships

The Home Depot uses smart contracts on blockchain to quickly resolve disputes with vendors. Through real-time communication and increased visibility into the supply chain, they are building stronger relationships with suppliers, resulting in more time for critical work and innovation.

Making international trade faster and more efficient

By joining we.trade, the trade finance network convened by IBM Blockchain, businesses are creating an ecosystem of trust for global trade. As a blockchain-based platform, we.trade uses standardized rules and simplified trading options to reduce friction and risk while easing the trading process and expanding trade opportunities for participating companies and banks.

Tokens

A token is a *representation of something in the blockchain*. This something can be money, time, services, shares in a company, a virtual pet, anything. By representing things as tokens, we can allow smart contracts to interact with them, exchange them, create or destroy them.A token contract is simply an Ethereum smart contract. "Sending tokens" actually means "calling a method on a smart contract that someone wrote and deployed". At the end of the day, a token contract is not much more a mapping of addresses to balances, plus some methods to add and subtract from those balances.

It is these balances that represent the tokens themselves. Someone "has tokens" when their balance in the token contract is non-zero. That's it! These balances could be considered

money, experience points in a game, deeds of ownership, or voting rights, and each of these tokens would be stored in different token contracts.

Different Kinds of Tokens

Note that there's a big difference between having two voting rights and two deeds of ownership: each vote is equal to all others, but houses usually are not! This is called fungibility. Fungible goods are equivalent and interchangeable, like Ether, fiat currencies, and voting rights. Non-fungible goods are unique and distinct, like deeds of ownership, or collectibles.

In a nutshell, when dealing with non-fungibles (like your house) you care about which ones you have, while in fungible assets (like your bank account statement) what matters is how much you have.

Standards

Even though the concept of a token is simple, they have a variety of complexities in the implementation. Because everything in Ethereum is just a smart contract, and there are no rules about what smart contracts have to do, the community has developed a variety of **standards** (called EIPs or ERCs) for documenting how a contract can interoperate with other contracts.

You've probably heard of the ERC20 or ERC721 token standards, and that's why you're here. Head to our specialized guides to learn more about these:

- ERC20: the most widespread token standard for fungible assets, albeit somewhat limited by its simplicity.
-
-

ERC721: the de-facto solution for non-fungible tokens, often used for collectibles and games.

-
-

ERC777: a richer standard for fungible tokens, enabling new use cases and building on past learnings. Backwards compatible with ERC20.

-

UNIT IV

Define mining. Explain the factors required for block validation

In Ethereum, miners refers to a vast global network of computers, operated mostly by enthusiasts in their homes and offices, running Ethereum nodes that are paid in ether tokens for the work of executing smart contracts and validating the canonical order of transactions around the world.

The process of mining is undertaken by each individual node, but the term also refers to the collective effort of the network: individual nodes mine, and the network itself can be said to be secured by mining. Miners process transactions in groups known as blocks. We previously defined a block, in the abstract, as a collated set of transactions that take place over a given period of time. However, a block can also refer to the data object containing those transactions, stored on Ethereum nodes. Each time a node starts, it must download the blocks it missed while offline. Each block contains some metadata from the previous block, to prove it is authentic and build on the existing blockchain.

Factors Required for Block Validation

Every candidate block that each individual miner constructs and seeks to validate contains four pieces of data:

Hash of the transaction ledger for this block (as this machine heard about it)

Root hash of the entire blockchain

Block number since the chain started

Difficulty of this block

If all these things check out, this block is a candidate for winning block. However, even with this information correct, the miner must still solve the proof-of-work algorithm. As you'll see, the algorithm is essentially a guessing game designed to take a certain amount of time, in service of the ideal 15-second block time

When the guess is correct, this correct value, or nonce, is the final condition to render a block true, canonical, and valid. The nonce is known as evidence of solving the proof-of-work algorithm.

How Proof of Work Helps Regulate Block Time

Anyone who can optimize for the proof-of-work algorithm can find valid blocks faster, causing uncles to lag further and further behind. In the Bitcoin network, a small group of hardware companies has acquired a disproportionately huge amount of power over the network by creating hardware specifically built to run the Bitcoin PoW algorithm. The centralisation of mining efforts is highly profitable in Bitcoin, because it allows these big miners to find blocks faster, reaping all the block rewards.

Slower machines never get a chance to solve a block, and eventually, even their uncle blocks come in further and further behind the winning block. In Ethereum, uncle blocks are required to bolster the winning block. As uncles lag more, it becomes harder for the network to find a true block, being that valid uncles are a requirement. Enter the Ethash algorithm: The Ethereum protocol's defense against mining hardware optimization. Ethash is a derivative of Dagger-Hashimoto, which is a memoryhard algorithm that can't be brute-forced with a custom application-specific integrated circuit (ASIC), like the kind that are popular with Bitcoin mining enterprises. Key to this algorithm memory-hardness is its reliance on a directed acyclic graph (DAG) file, which is essentially a 1 GB dataset created anew every 125 hours, or 30,000 blocks. This period of 30,000 blocks is also known as an epoch. Directed acyclic graph is a technical term for a tree in which each node is allowed to have multiple parents, with ten levels including the root, and a total of up to 225 values.

What is cryptoeconomics. Explain it purpose.

The combination of incentives and cryptography to build systems, applications, and networks is known as Crypto-Economics. To put it another way, crypto economics is applied cryptography that considers economic incentives and economic theory.

Cryptography is used in both digital signatures and hash functions on blockchains.

he security mechanism of Bitcoin is based on the notion of majority rule. This means that a 51 percent attack, in which bad actors seize control of the majority of the network's computer capacity, may theoretically gain control of the blockchain.

The attackers would be able to block fresh transactions from receiving confirmations or even reverse transactions entirely in such a scenario. Getting control of this much hashing power, on the other hand, would be prohibitively expensive, needing major hardware and large quantities of electricity.

One of the reasons Bitcoin has been so successful is because of cryptoeconomics. Satoshi Nakamoto used assumptions to support certain incentives for the network's various

participant groups. The validity of these assumptions about how network participants react to various economic incentives is critical to the system's security assurances.

There would be no secure unit of account to compensate miners without the hardness of its cryptographic technology. Without the miners, there would be no way to verify the veracity of the distributed ledger's transaction history unless it was confirmed by a trusted third party, which would nullify one of Bitcoin's key benefits.

The symbiotic relationship between miners and the Bitcoin network promotes confidence, according to cryptoeconomic theories. This does not, however, guarantee that the system will continue to exist in the future.

UNIT V

How to Set Up a Private Ethereum Network

What is Blockchain?

- Blockchain is a chain of blocks that contains all information about all the transactions that take place in a network in an encrypted form by using public and private keys.
- Blockchain technology can strengthen the basic services that are essential in trade finance. The blockchain model works on a decentralized, digitalized, and distributed ledger model. Because of these properties, this is more robust and secure than the proprietary, centralized which are currently used in the trading system.
- In the simplest of terms, a blockchain is just a new form of a decentralized database.

What is Ethereum?

Ethereum is a decentralized open-source blockchain system that features its own cryptocurrency which is called ether(ETH). It is a platform that can be used for various Dapps which can be deployed by using Smart Contracts.

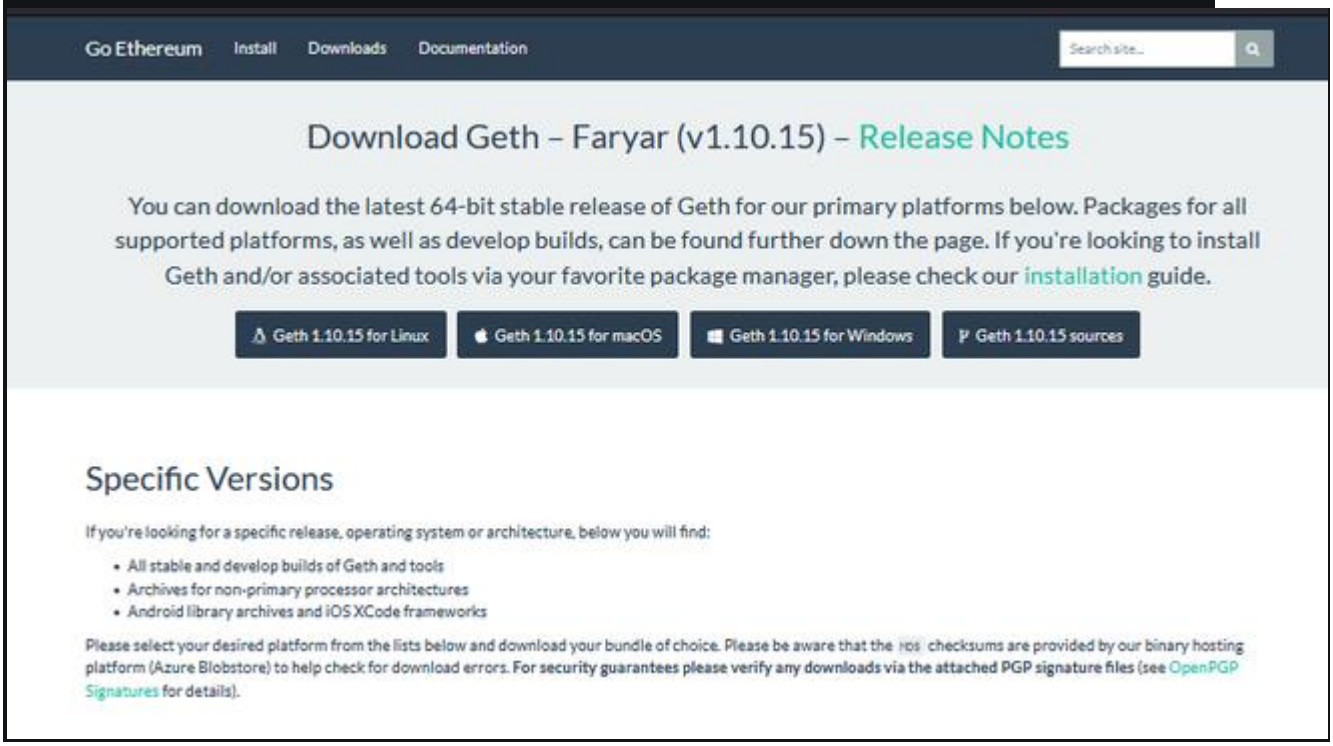
Why Build a Private Blockchain Network?

- In an Ethereum network if the nodes are not connected to the main network then it is known as a Private Ethereum Network.
- Only the nodes with the right permissions will be able to access this Blockchain.
- Privacy – Don’t want to key data on a public network that’s why enterprises build a private network.
- To test the smart contracts and to develop smart contracts.

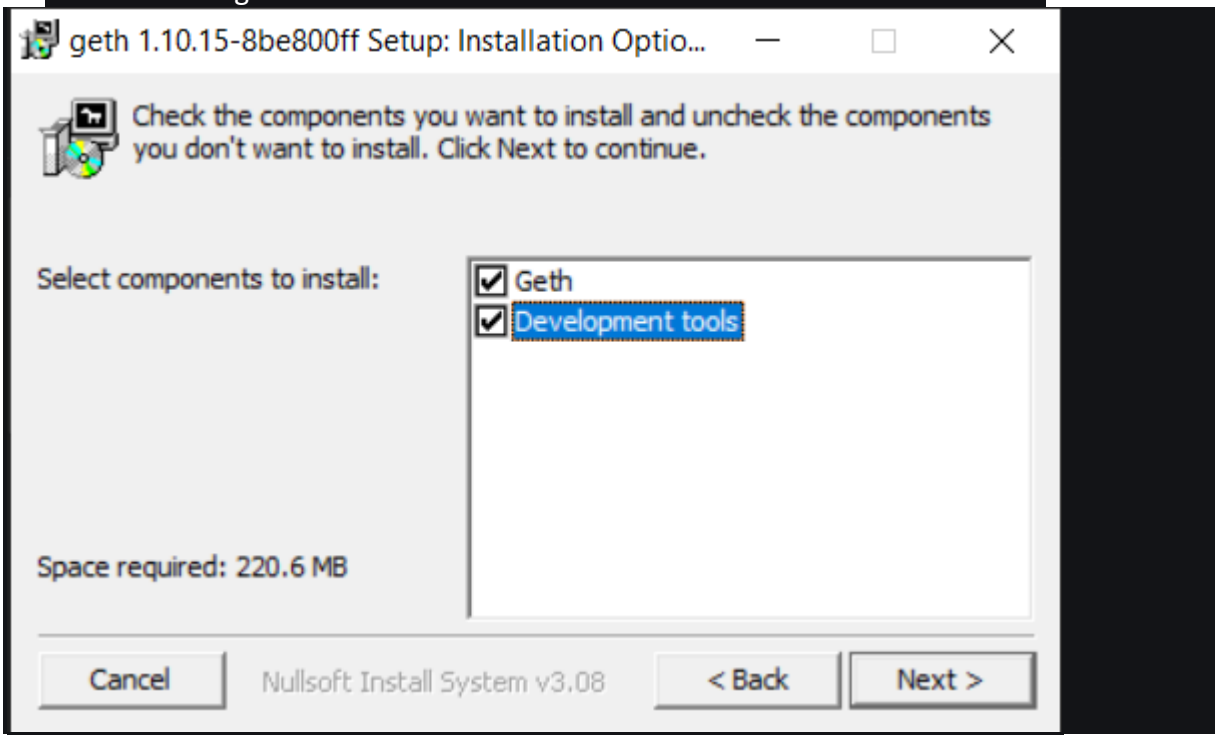
Steps to Set Up Private Ethereum Network

Below is the step-by-step guide to setting up a private Ethereum network.

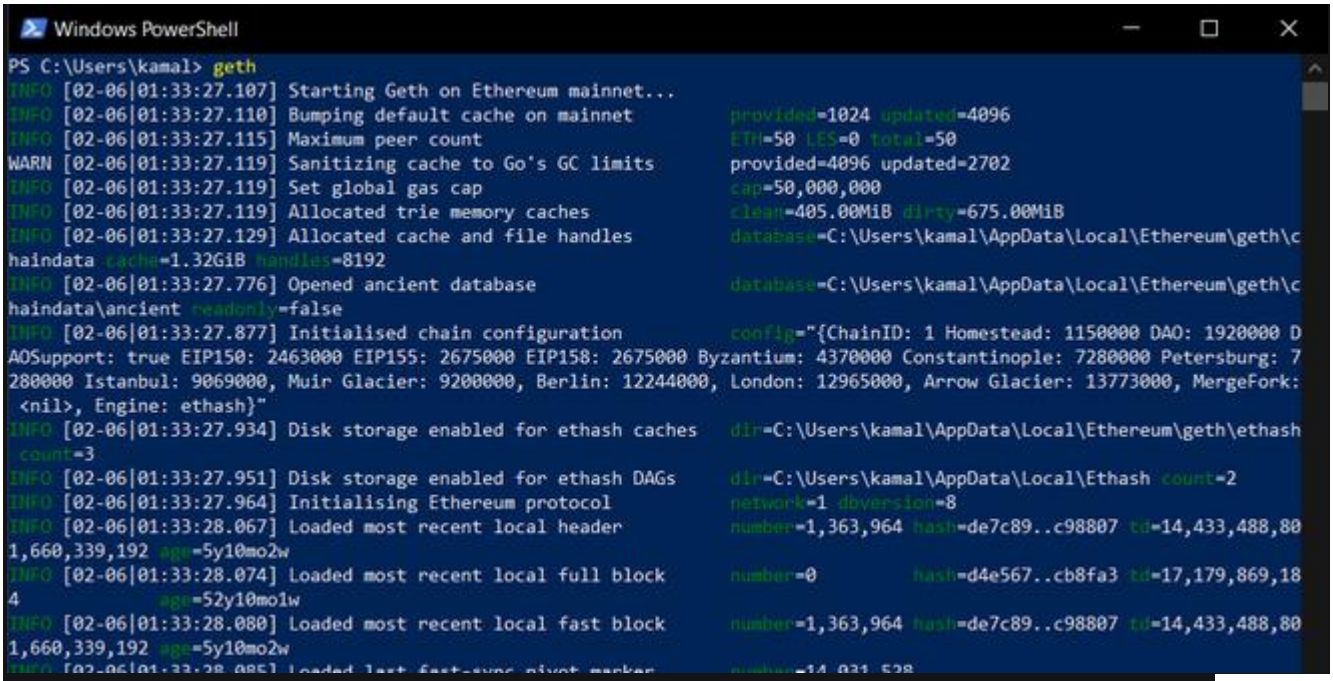
Step 1: Install Geth on Your System



- While installing Geth make sure to select both checkboxes as shown below.



- After installing Geth on your system open PowerShell or command prompt and type geth and press enter, the following output will be displayed.

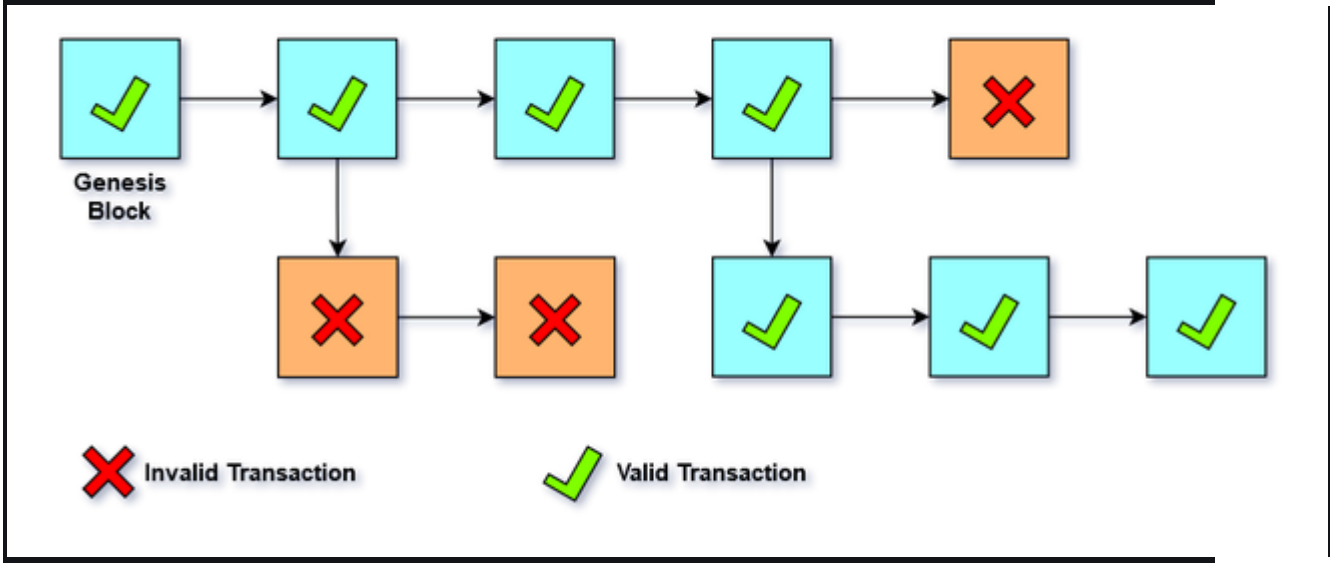


Step 2: Create a Folder For Private Ethereum

- Create a separate folder for this project. In this case, the folder is **MyNetwork.**
- Create a new folder inside the folder MyNetwork for the private Ethereum network as it keeps your Ethereum private network files separate from the public files. In this example folder is **MyPrivateChain.**

Step 3: Create a Genesis Block

The blockchain is a distributed digital register in which all transactions are recorded in sequential order in the form of blocks. There are a limitless number of blocks, but there is always one separate block that gave rise to the whole chain i.e. the genesis block.



As seen in the above diagram we can see that blockchain is initialized with the genesis block.

To create a private blockchain, a genesis blockis needed. To do this, create a genesis file, which is a JSON file with the following commands-

Explanation:

- **config:** It defines the blockchain configuration and determines how the network will work.
- **chainId:** This is the chain number used by several blockchains. The Ethereum main chain number is “1”. Any random number can be used, provided that it does not match with another blockchain number.
- **homesteadBlock:** It is the first official stable version of the Ethereum protocol and its attribute value is “0”.
- One can connect other protocols such as Byzantium, eip155B, and eip158. To do this, under the homesteadBlock add the protocol name with the Block prefix (for example, eip158Block) and set the parameter “0” to them.
- **difficulty:** It determines the difficulty of generating blocks. Set it low to keep the complexity low and to avoid waiting during tests.
- **gasLimit:** Gas is the “fuel” that is used to pay transaction fees on the Ethereum network. The more gas a user is willing to spend, the higher will be the priority of his transaction in the queue. It is recommended to set this value to a high enough level to avoid limitations during tests.
- **alloc:** It is used to create a cryptocurrency wallet for our private blockchain and fill it with fake ether. In this case, this option will not be used to show how to initiate mining on a private blockchain.

This file can be created by using any text editor and save the file with **JSON extension** in the folder MyNetwork.

Step 4: Execute genesis file

Open cmd or PowerShell in admin mode enter the following command-

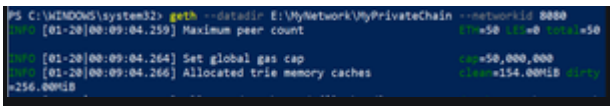
The above command instructs Geth to use the CustomGenesis.json file.

After executing the above command Geth is connected to the Genesis file and it seems like this:

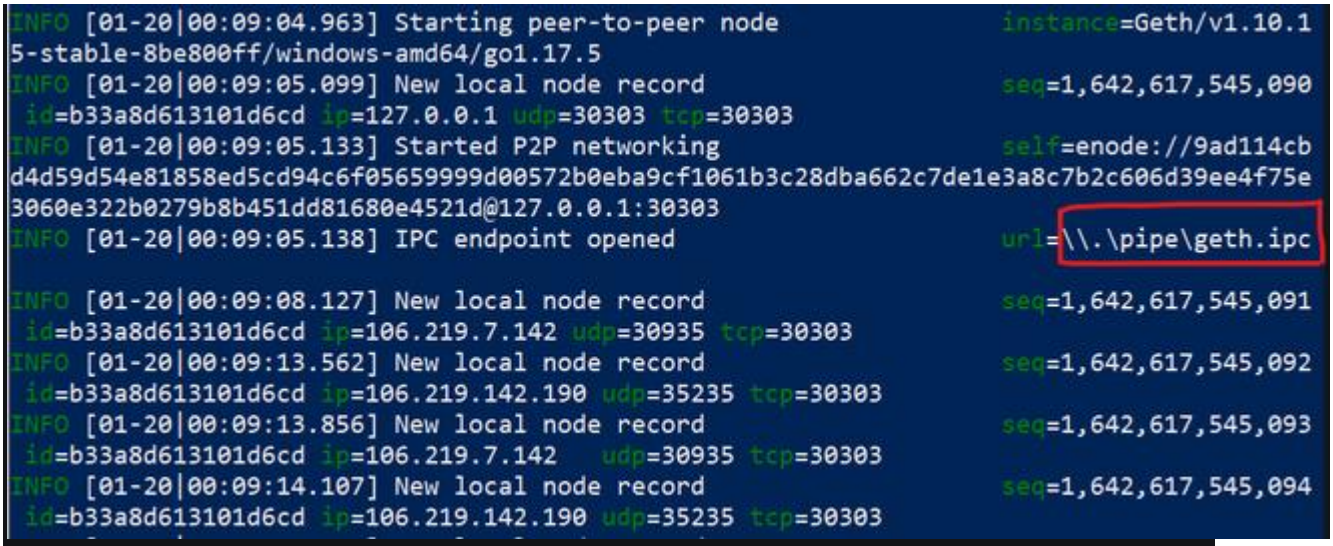
```
PS C:\WINDOWS\system32> geth --identity "MyBlockChain" init E:\MyNetwork\CustomGenesis.j
son --datadir E:\MyNetwork\MyPrivateChain
INFO [01-19|23:43:15.534] Maximum peer count                       ETH=50 LES=0 total=50

INFO [01-19|23:43:15.564] Set global gas cap                       cap=50,000,000
INFO [01-19|23:43:15.567] Allocated cache and file handles         database=E:\MyNetwork
\MyPrivateChain\geth\chaindata cache=16.00MiB handles=16
INFO [01-19|23:43:15.803] Writing custom genesis block
INFO [01-19|23:43:15.815] Persisted trie from memory database      nodes=0 size=0.00B ti
me="516µs" gcnodes=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [01-19|23:43:15.846] Successfully wrote genesis state         database=chaindata ha
sh=d1a12d..4c8725
INFO [01-19|23:43:15.862] Allocated cache and file handles         database=E:\MyNetwork
\MyPrivateChain\geth\lightchaindata cache=16.00MiB handles=16
INFO [01-19|23:43:16.092] Writing custom genesis block
INFO [01-19|23:43:16.103] Persisted trie from memory database      nodes=0 size=0.00B ti
me=0s      gcnodes=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [01-19|23:43:16.120] Successfully wrote genesis state         database=lightchainda
ta hash=d1a12d..4c8725
PS C:\WINDOWS\system32>
```

Step 5: Initialize the private network

Launch the private network in which various nodes can add new blocks for this we have to run the command-



The command also has the identifier **8080**. It should be replaced with an arbitrary number that is not equal to the identifier of the networks already created, for example, the identifier of the main network Ethereum (“networkid = 1”). After successfully executing the command we can see like this-



Note:

Every time there is a need to access the private network chain, one will need to run commands in the console that initiate a connection to the Genesis file and the private network.

Now a personal blockchain and a private Ethereum network is ready.

Step 6: Create an Externally owned account(EOA)

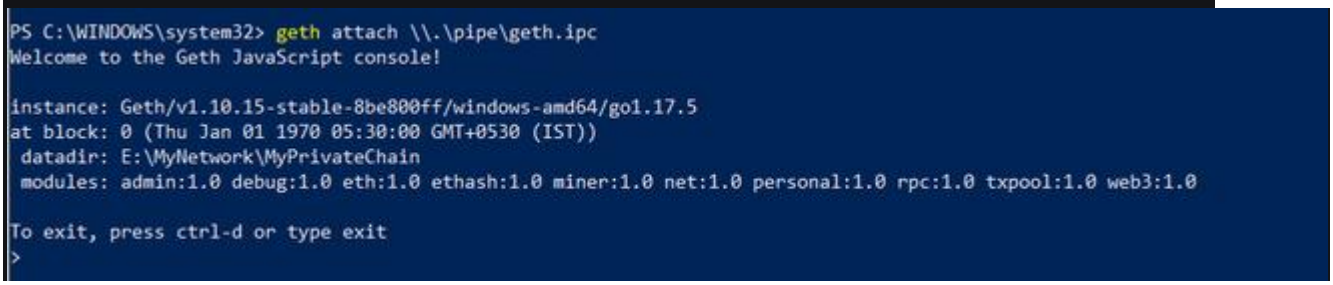
Externally Owned Account(EOA) has the following features-

- Controlled by an External party or person.
- Accessed through private Keys.
- Contains Ether Balance.
- Can send transactions as well as 'trigger' contract accounts.

Steps to create EOA are:

To manage the blockchain network, one need EOA. To create it, run Geth in two windows. In the second window console enter the following command-

This will connect the second window to the terminal of the first window. The terminal will display the following-



Create an account by using the command-

After executing this command enter Passphrase and you will get your account number and save this number for future use.

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0x125c7bce5af112d0e271092be64c87ce5c31696c"
>
```

To check the balance status of the account execute the following command-

```
> eth.getBalance("0x125c7bce5af112d0e271092be64c87ce5c31696c")
0
```

It can be seen from the above screenshot that it shows zero balance. This is because when starting a private network in the genesis file, we did not specify anything in the alloc attribute.

Step 7: Mining our private chain of Ethereum

If we mine in the main chain of Ethereum it will require expensive equipment with powerful graphics processors. Usually, ASICs are used for this but in our chain high performance is not required and we can start mining by using the following command-

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

If the balance status is checked after a couple of seconds the account is replenished with fake ether. After that, one can stop mining by using the following command-

```
> eth.getBalance("0x125c7bce5af112d0e271092be64c87ce5c31696c")
2000000000000000000000
> miner.stop()
null
>
```

1) How to deploy and interact with smart contract.

Unlike most software, smart contracts don't run on your computer or somebody's server: they live on the Ethereum network itself. This means that interacting with them is a bit different from more traditional applications.

This guide will cover all you need to know to get you started using your contracts, including:

- Setting up a Local Blockchain
- Deploying a Smart Contract
- Interacting from the Console
- Interacting Programmatically

Setting up a Local Blockchain

Before we begin, we first need an environment where we can deploy our contracts. The Ethereum blockchain (often called "mainnet", for "main network") requires spending real money to use it, in the form of Ether (its native currency). This makes it a poor choice when trying out new ideas or tools.

To solve this, a number of "testnets" (for "test networks") exist: these include the Ropsten, Rinkeby, Kovan and Goerli blockchains. They work very similarly to mainnet, with one difference: you can get Ether for these networks for free, so that using them doesn't cost you a single cent. However, you will still need to deal with private key management, blocktimes in the range of 5 to 20 seconds, and actually getting this free Ether.

During development, it is a better idea to instead use a *local* blockchain. It runs on your machine, requires no Internet access, provides you with all the Ether that you need, and mines blocks instantly. These reasons also make local blockchains a great fit for automated tests.

If you want to learn how to deploy and use contracts on a *public* blockchain, like the Ethereum testnets, head to our Connecting to Public Test Networks guide.

Hardhat comes with a local blockchain built-in, the Hardhat Network.

Upon startup, Hardhat Network will create a set of unlocked accounts and give them Ether.

```
$ npx hardhat node
```

Hardhat Network will print out its address, `http://127.0.0.1:8545`, along with a list of available accounts and their private keys.

Keep in mind that every time you run Hardhat Network, it will create a brand new local blockchain - the state of previous runs is **not** preserved. This is fine for short-lived experiments, but it means that you will need to have a window open running Hardhat Network for the duration of these guides.

Hardhat will always spin up an instance of **Hardhat Network** when no network is specified and there is no default network configured or the default network is set to `hardhat`.

You can also run an actual Ethereum node in *development mode*. These are a bit more complex to set up, and not as flexible for testing and development, but are more representative of the real network.

Deploying a Smart Contract

In the Developing Smart Contracts guide we set up our development environment.

If you don't already have this setup, please create and setup the project and then create and compile our Box smart contract.

With our project setup complete we're now ready to deploy a contract. We'll be deploying Box, from the Developing Smart Contracts guide. Make sure you have a copy of Box in contracts/Box.sol.

Hardhat doesn't currently have a native deployment system, instead we use scripts to deploy contracts.

We will create a script to deploy our Box contract. We will save this file as scripts/deploy.js.

```
// scripts/deploy.js
async function main () {
  // We get the contract to deploy
  const Box = await ethers.getContractFactory('Box');
  console.log('Deploying Box...');
  const box = await Box.deploy();
  await box.deployed();
  console.log('Box deployed to:', box.address);
}

main()
  .then(() => process.exit(0))
  .catch(error => {
    console.error(error);
    process.exit(1);
  });
```

We use ethers in our script, so we need to install it and the @nomiclabs/hardhat-ethers plugin.

```
$ npm install --save-dev @nomiclabs/hardhat-ethers ethers
```

We need to add in our configuration that we are using the @nomiclabs/hardhat-ethers plugin.

```
// hardhat.config.js
require('@nomiclabs/hardhat-ethers');

...
module.exports = {
...
};
```

Using the run command, we can deploy the Box contract to the local network (Hardhat Network):

```
$ npx hardhat run --network localhost scripts/deploy.js
Deploying Box...
Box deployed to: 0x5FbDB2315678afecb367f032d93F642f64180aa3
```

Hardhat doesn't keep track of your deployed contracts. We displayed the deployed address in our script (in our example, `0x5FbDB2315678afecb367f032d93F642f64180aa3`). This will be useful when interacting with them programmatically.

All done! On a real network this process would've taken a couple of seconds, but it is near instant on local blockchains.

If you got a connection error, make sure you are running a local blockchain in another terminal.

Remember that local blockchains **do not** persist their state throughout multiple runs! If you close your local blockchain process, you'll have to re-deploy your contracts.

Interacting from the Console

With our `Box` contract deployed, we can start using it right away.

We will use the Hardhat console to interact with our deployed `Box` contract on our localhost network.

We need to specify the address of our `Box` contract we displayed in our deploy script.

It's important that we explicitly set the network for Hardhat to connect our console session to. If we don't, Hardhat will default to using a new ephemeral network, which our Box contract wouldn't be deployed to.

```
$ npx hardhat console --network localhost
Welcome to Node.js v12.22.1.
Type ".help" for more information.
> const Box = await ethers.getContractFactory('Box');
undefined
> const box = await Box.attach('0x5FbDB2315678afecb367f032d93F642f64180aa3')
undefined
```

Sending transactions

`Box`'s first function, `store`, receives an integer value and stores it in the contract storage. Because this function *modifies* the blockchain state, we need to *send a transaction* to the contract to execute it.

We will send a transaction to call the `store` function with a numeric value:

```
> await box.store(42)
{
  hash: '0x3d86c5c2c8a9f31bedb5859efa22d2d39a5ea049255628727207bc2856cce0d3',
```

...

Querying state

Box's other function is called retrieve, and it returns the integer value stored in the contract. This is a *query* of blockchain state, so we don't need to send a transaction:

```
> await box.retrieve()
BigNumber { _hex: '0x2a', _isBigNumber: true }
```

Because queries only read state and don't send a transaction, there is no transaction hash to report. This also means that using queries doesn't cost any Ether, and can be used for free on any network.

Our Box contract returns uint256 which is too large a number for JavaScript so instead we get returned a big number object. We can display the big number as a string using (await box.retrieve()).toString().

```
> (await box.retrieve()).toString()
'42'
```

To learn more about using the console, check out the Hardhat documentation.

Interacting programmatically

The console is useful for prototyping and running one-off queries or transactions. However, eventually you will want to interact with your contracts from your own code.

In this section, we'll see how to interact with our contracts from JavaScript, and use Hardhat to run our script with our Hardhat configuration.

Keep in mind that there are many other JavaScript libraries available, and you can use whichever you like the most. Once a contract is deployed, you can interact with it through any library!

Setup

Let's start coding in a new scripts/index.js file, where we'll be writing our JavaScript code, beginning with some boilerplate, including for writing async code.

```
// scripts/index.js
async function main () {
  // Our code will go here
}

main()
  .then(() => process.exit(0))
  .catch(error => {
```

```
  console.error(error);
  process.exit(1);
});
```

We can test our setup by asking the local node something, such as the list of enabled accounts:

```
// Retrieve accounts from the local node
const accounts = await ethers.provider.listAccounts();
console.log(accounts);
```

We won't be repeating the boilerplate code on every snippet, but make sure to always code *inside* the `main` function we defined above!

Run the code above using `hardhat run`, and check that you are getting a list of available accounts in response.

```
$ npx hardhat run --network localhost ./scripts/index.js
[
  '0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266',
  '0x70997970C51812dc3A010C7d01b50e0d17dc79C8',
...
]
```

These accounts should match the ones displayed when you started the local blockchain earlier. Now that we have our first code snippet for getting data out of a blockchain, let's start working with our contract. Remember we are adding our code *inside* the `main` function we defined above.

Getting a contract instance

In order to interact with the `Box` contract we deployed, we'll use an ethers contract instance.

An ethers contract instance is a JavaScript object that represents our contract on the blockchain, which we can use to interact with our contract. To attach it to our deployed contract we need to provide the contract address.

```
// Set up an ethers contract, representing our deployed Box instance
const address = '0x5FbDB2315678afecb367f032d93F642f64180aa3';
const Box = await ethers.getContractFactory('Box');
const box = await Box.attach(address);
```

Make sure to replace the `address` with the one you got when deploying the contract, which may be different to the one shown here.

We can now use this JavaScript object to interact with our contract.

Calling the contract

Let's start by displaying the current value of the Box contract.

We'll need to call the read only retrieve() public method of the contract, and await the response:

```
// Call the retrieve() function of the deployed Box contract
const value = await box.retrieve();
console.log('Box value is', value.toString());
```

This snippet is equivalent to the query we ran earlier from the console. Now, make sure everything is running smoothly by running the script again and checking the printed value:

```
$ npx hardhat run --network localhost ./scripts/index.js
Box value is 42
```

If you restarted your local blockchain at any point, this script may fail. Restarting clears all local blockchain state, so the Box contract instance won't be at the expected address.

If this happens, simply start the local blockchain and redeploy the Box contract.

Sending a transaction

We'll now send a transaction to store a new value in our Box.

Let's store a value of 23 in our Box, and then use the code we had written before to display the updated value:

```
// Send a transaction to store() a new value in the Box
await box.store(23);

// Call the retrieve() function of the deployed Box contract
const value = await box.retrieve();
console.log('Box value is', value.toString());
```

In a real-world application, you may want to estimate the gas of your transactions, and check a gas price oracle to know the optimal values to use on every transaction.

How to execute smart contract functions.

There are a couple of ways you can call a Ethereum smart contract functions. In this article we will cover three ways: through **Remix** (a Ethereum Javascript Based Compiler), with **Javascript** and with **SolC** (a Solitidy Compiler).

Firstly, we will need to deploy a contract, later on we will call this contracts functions. In order to do that we will keep the same workflow:

1. Deploy a simple contract
2. Load it, creating an instance of it.
3. Call its functions

Here is the code of our contract

```
pragma solidity ^0.4.20;

contract SimpleExample {
   uint counter;
   address owner;

   function SimpleExample(uint _initialCount, address _owner) public {
      counter = _initialCount;
      owner = _owner;
   }

   function addOne() public {
      counter = counter + 1;
   }

   function SetCounterToZero() public {
      require(msg.sender == owner);
      counter = 0;
   }

   function Counter() constant public returns (uint) {
      return counter;
   }

   function Sender() constant public returns (address) {
      return msg.sender;
   }

   function OddOrEven() constant public returns (string) {
      if(counter != 0) {
         if (counter % 2 != 0  ) {
            return "Odd";
         }
```

```
        else {
           return "Even";
        }
     } else {
        return "Counter == 0";
     }

  }

}
```

Write a short note on Dapp Contract Data Models.

pplications that utilize smart contracts are known as *decentralized applications*, or **dapps** for short. Dapps are frontend apps that interface with *smart contracts* (instead of servers) to persist or retrieve data on the *blockchain* (instead of databases). Clients interact with dapps through *external accounts*. To help web developers looking to get into Ethereum dApp development, the author hopes to impart a simple **mental model** for these architectural layers that make up the blockchain application stack.

https://medium.com/heartbanklab/a-complete-mental-model-for-ethereum-dapp-development-5ce08598ed0a