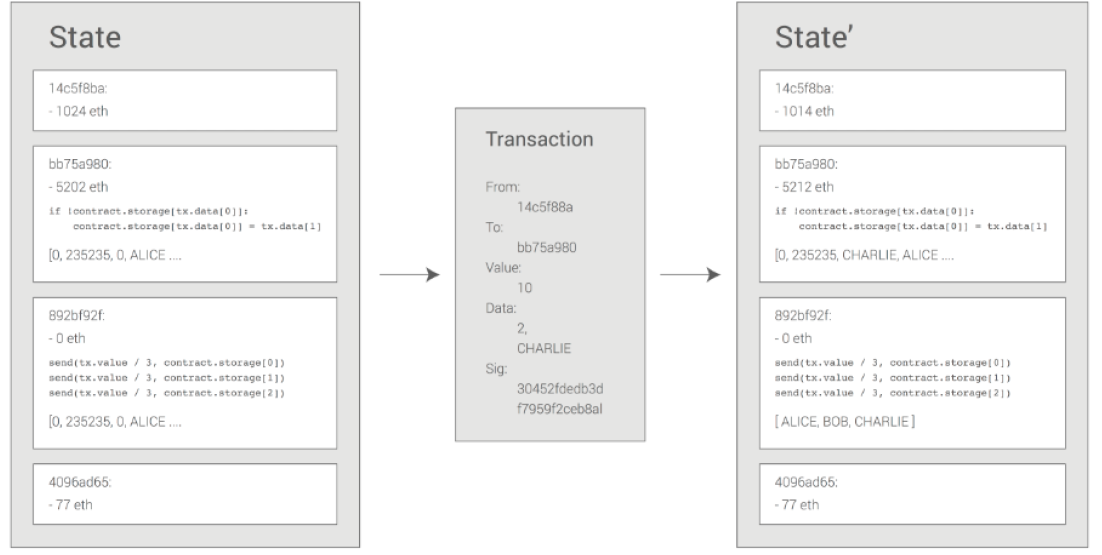
4 March 2022

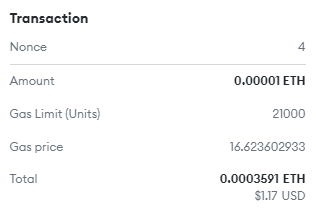
Ethereum vs BTC

* Hard money vs smart contract functionality
  + BTC – fixing the double spend problem via digital currency
* Ethereum
  + Smart contracts in Bitcoin lack Turing completeness – does not support all computation e.g programmability restrictions?
* Ethereum – a blockchain with a built-in Turing complete programming language – allowing anyone to write smart contracts and decentralised apps, where they can create their own arbitrary rules for ownership, transaction formats and state transition functions
* Smart contracts – cryptographic ‘boxes’ that contain value and can only unlock if certain conditions are met – cannot be done on BTC bc of added powers of Turing-completeness, value-awareness, blockchain-awareness and state
* The state is made up of *objects* called “accounts” – each account has a 20-byte address
  + State transitions – direct transfers of value and information between accounts
  + Each account has four fields
    - The nonce, a counter to make sure each transaction can only be processed once
    - Account’s current ether balance
    - Account’s contract code, if present
    - Account’s storage (empty by default)
  + Ether is the main internal crypto fuel of the Ethereum network
  + Externally owned accounts (private key ownership) vs contract accounts (controlled by contract code)
    - External accounts have no code, one can send messages by creating and signing a transaction
    - For contract account, they activate the code for each message received, allowing it to read/write to internal storage, send other messages or create contracts in turn
* Messages and transactions
  + A “message” – an Ethereum message can be created by an external entity or contract, whereas a BTC transaction can only be created externally (from account to account transaction)
    - Ethereum messages can contain data
    - The receipient, if it is a contract account, has the option to return a response
    - Message mechanism has the ‘first class citizen’ property – contracts have equivalent powers to external accounts, including ability to send message and create other contracts
  + A “transaction” – the signed data package that stores a message to be sent from an externally owned account, containing:
    - The recipient of the message
    - A signature identifying the sender
    - The amount of other and data to send
    - ‘STARTGAS’ and ‘GAS PRICE’ – to prevent exponential blowup/infinite loops in code, each transaction needs to set a limit to the amount of computational steps of code execution. STARTGAS is this limit, GASPRICE is the fee to pay to the miner per computational step. If transaction runs out of gas, **all state changes revert except for the payment of fees.** If transaction execution halts with some gas left, remaining is refunded to the sender.

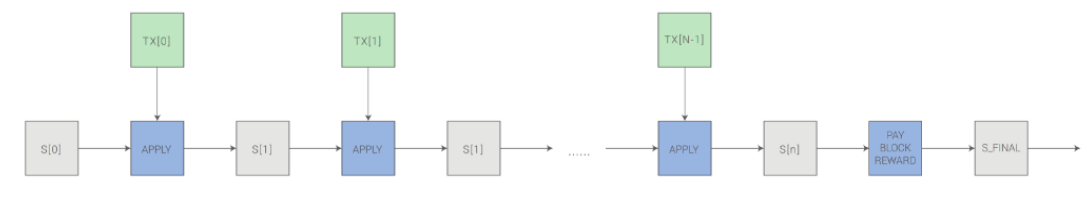


* The Ethereum state transition function, APPLY(S, TX) -> S’ can be defined as follows:
  + Check if transaction is well formed (has right number of values), valid signature and nonce matches the nonce in sender’s account. If not, return an error
  + Calculate transaction fee as: STARTGAS\*GASPRICE and determine sending address from the signature. Subtract fee from sender’s account balance, increment the sender’s nonce (recall that nonce is # transaction counter). If not enough balance to spend, return error.
  + Initialise GAS = STARTGAS, take off a certain quantity of gas per byte to pay for the bytes in the transaction
    - Larger transaction = more computation = more gas?
  + Transfer transaction value from sender’s account to receiving account. **If the receiving account does not exist yet, create it.** If the receiving account is a contract, run the contract’s code either to completion or until the execution runs out of gas
  + If the value transfer failed bc the sender did not have enough money, or code execution ran out of gas, revert all state changes except payment of fees, and add the fees to the miner’s account
  + Otherwise, refund the fees for all remaining gas to sender and send fees paid for gas consumed to miner
* Case study - 
  + Other details – 10 ETH value, 2000 gas, 0.001 gas price, two data fields [2, ‘CHARLIE’]
  + Check that the transaction is *valid* and *well formed*
  + Check that the sender has at least 2000 \* 0.001 = 2 ether (currency is in ETH no gwei). Then subtract from sender’s balance.
  + Initalise GAS = 2000. Assuming transaction is 170 bytes long and byte-fee is 5, subtract 170 \* 5 so that there is 1150 gas left
  + Subtract 10 more ether for the transaction value from sender’s account, add it to contract’s account
  + Run code – checks if contract’s storage at index 2 is used, notices that it is not, sets the storage at index 2 to the value of CHARLIE. Suppose this takes 187 gas, remaining gas: 1150 – 187 = 963
  + Add 963 \* 0.001 = 0.963 back to sender’s account and return the resulting state
  + Other notes:
    - If there was no contract at the receiving end, total tx fee would be equal to provided GASPRICE multiplied by length of transaction in bytes. Data sent alongside the tx would be irrelevant.
    - Contract-initiated messages can assign gas limits to the computation they spawn – if the sub-computation runs out of gas, it gets reverted onto to point of the message call. This means contracts can secure their limited computation resources by setting strict limits on their sub-computations
    - Contract demands 10 ETH upfront + 2000\*0.001 = 12 ETH maximum
    - After the contract-size and code logic execution, any remaining gas gets sent back to the sender

\*\* Metamask – Nonce counter. Does not seem to display on Etherscan.io



* Code execution – code in contracts is written in ‘low-level, stack-based bytecode language’ referred to Ethereum Virtual Machine code or EVM code. This code consists of a series of bytes, where *each byte* represents an operation
  + Code execution is an infinite loop that repeatedly carries out the operation based on a counter that begins at 0 then increments the counter by one all the way until the end of the code is reached, or an error or STOP or RETURN instruction is detected
  + Operations have access to 3 spaces for which to store data:
    - The stack – last-in-first-out container to which 32-byte values can be pushed and popped
    - Memory, an infinitely expandable byte array
    - Contract’s long term storage – a key/value store where keys and values are both 32 bytes. Unlike stack and memory which reset after computation ends, storage persists for long term
      * 1. Insert data here
      * 2. Insert data here
  + Code can also access the value, sender and data of the incoming message as well as block header data. Code can also return a byte array of data as an output
  + Full computational state can be defined by the *tuple* (block\_state, transaction, message, code, memory, stack, pc, gas)
    - Block\_state – global state containing all accounts, including balances and storage
* Blockchain and mining – the Ethereum blockchain is similar to the BTC blockchain but with some differences
  + Ethereum blocks contain a copy of both the TX list and the most recent state. The block number and difficulty are also stored in the block
  + Block validation algorithm:
    - Check if previous block referenced exists and is valid
    - Check that timestamp of block is greater than that of referenced previous block and **less than 15 mins into the future**
    - Check that block number, difficulty, transaction root, uncle root and gas limit are valid
    - Check that proof of work on the block is valid (how?)
      * PoW – run iterations to find a special number, the miner that successfully does this secures the block reward and pushes the latest block onto the blockchain
        + Miners compute SHA256 on slightly modified versions of the block header millions of times over, until one node eventually comes up with a version whose hash is less than the target (2^190)
      * But how to easily check if a PoW is valid? \*\*
    - Let S[0] be the STATE\_ROOT of the *previous* block
    - Let TX be the block’s transaction list, with n transactions. For all in 0…n-1, setS[i+1] = APPLY(S[i],TX[i]). If any applications returns an error or if total gas consumed in block up to this point exceeds GASLIMIT, return an error
    - Let S\_FINAL be S[n] but adding block reward paid to miner
    - Check if S\_FINAL is same as the STATE\_ROOT – if it is, block is valid; otherwise it is not
      * Not sure ‘same’ in what regard? \*\*
  + “Patricia tree” – modification on the Merkle tree concept
    - Allows for nodes to be inserted and deleted, and not just changed, efficiently
  + Because all the state information is part of the last block, there is no need to store the entire blockchain history



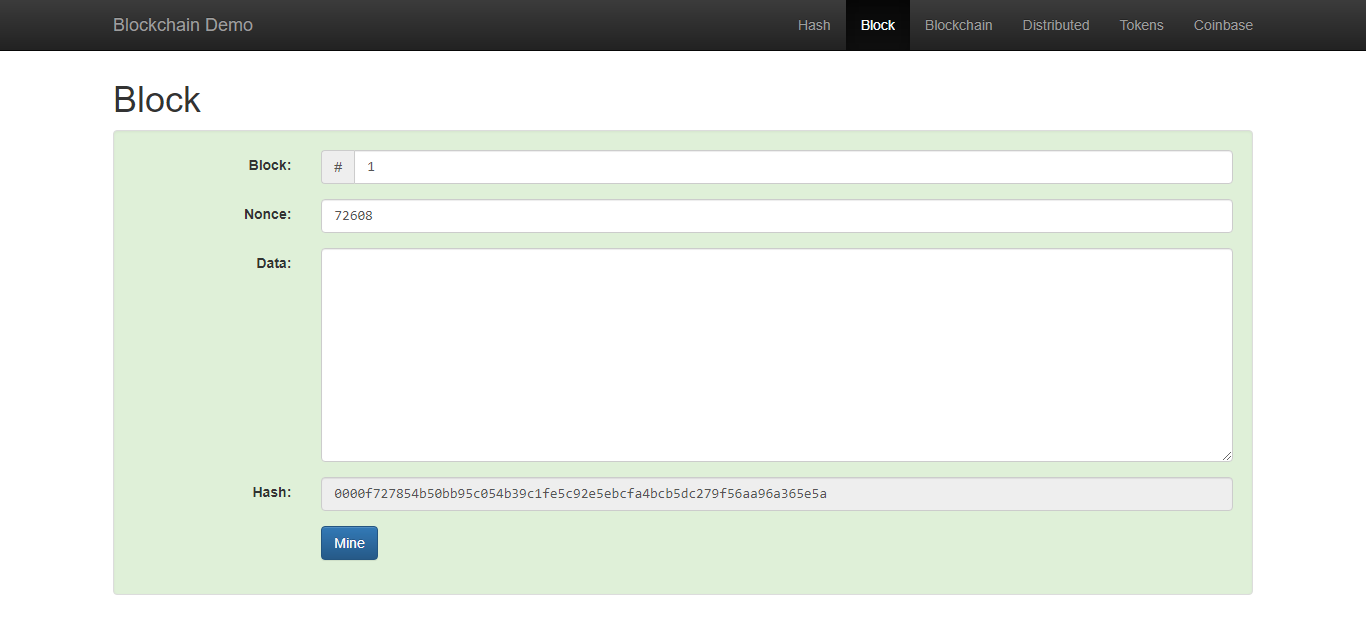
* Mining centralisation
  + Bitcoin mining activities dominated by ASICs -> favoured to capital heavy players, mining has shifted to be less decentralised / egalitarian. Most miners also do not perform block validation locally, instead relying on centralised mining pools to provide block headers
    - The top two mining pools indirectly control ~ 50% of processing power, but miners can switch if a pool attempts a 51% attack
  + Ethereum intends to use a mining algorithm based on randomly generating a unique hash function for every 1000 nonces (# of transactions), using a sufficiently broad range to remove the benefit of specialized hardware
    - Individual players with low tech laptop/desktop can perform a certain quantity of mining activity, paying only for electricity costs but after a point of 100% CPU utilisation any additional mining would demand more electricity/hardware investment.
    - If centralisation gain can be kept below this ratio (E + H) / E, then there will still be room for ordinary miners
  + The algorithm would demand that mining require access to the entire blockchain, forcing miners to store the entire blockchain or at least be capable of verifying every transaction
* Scaling
  + Like BTC, every transaction must be processed by every node in the network (Gossip protocol)
  + Key parameters – blockchain size AND rate of growth over time
  + The problem with large blockchains is centralisation risk – only a very small number of players would run full nodes
    - Full nodes can then band together and agree to cheat
    - Light nodes would have no way of detecting this immediately
    - Eventually, information of the fraud would trickle out but it would be too late
* Decentralised applications
  + Contract mechanism allows anyone to build a command line application run on a virtual machine that is executed by consensus across the entire network, allowing it to modify a *globally accessible state* as its “hard drive”
  + User friendliness challenge – a complete Dapp – consist of both low-level business logic components and high-level graphical user interface components
  + Ethereum client – serve as web browser and include support for a ETH Javascript API object, which can be used to interact with the Ethereum blockchain
  + These web pages are entirely static content, with the blockchain handling any user-initiated requests
* Conclusion – the existence of a Turing-complete programming language means that **arbitrary contracts can theoretically be created for any transaction type or application**
  + Moving beyond currency - financial contracts, gambling markets, decentralised applications, decentralised file storage etc
  + Adding value to peer-to-peer protocols by adding for the first time an economic layer
  + Ethereum is open-ended by design -> foundational layer for financial and non-financial protocols

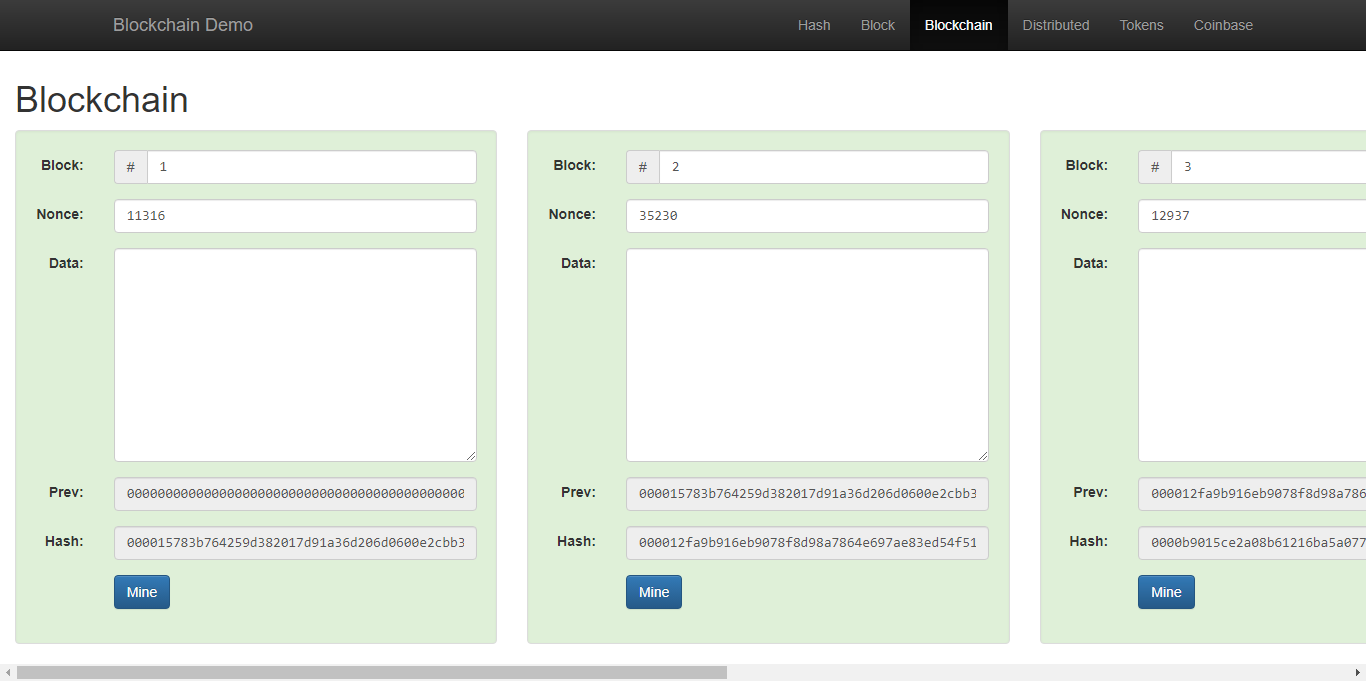
Beacon chain – introducing proof-of-stake

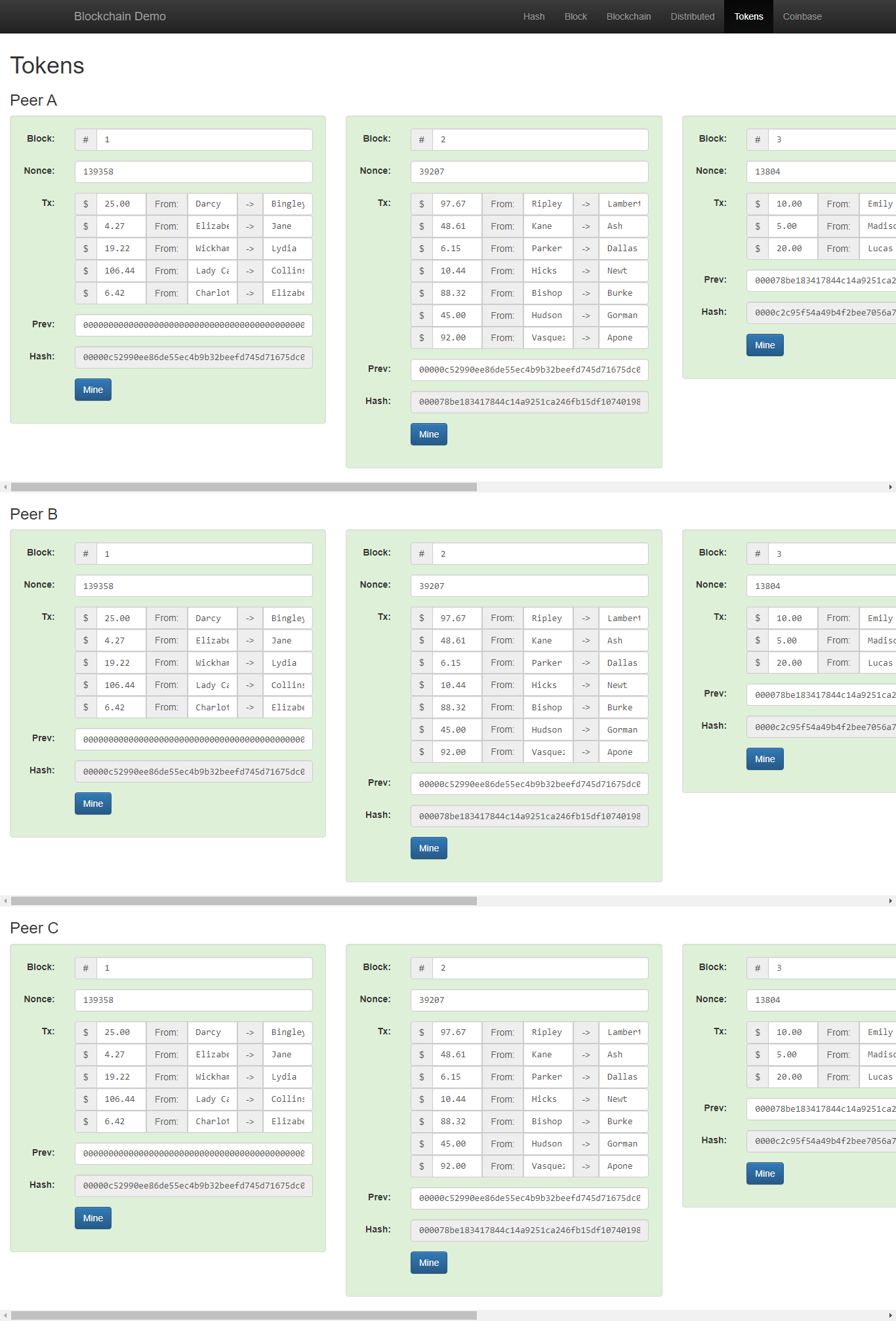
* Staking ETH activates validator software – a validator processes transactions and creates new blocks in the chain. This is easier than mining.
  + Validation is important for introducing shard chains – extra, cheaper storage layers for applications and rollups to store data. Enable layer 2 solutions to offer low transaction fees while having the mainnet security
  + Sharding – splitting a database horizontally to spread the load
    - Reduce network congestion and increase TX/second – producing new chains known as ‘shards’
    - Everyone can run a node – scale comes by increasing the size of the database (centralised actors) or reducing the size of the database (decentralised actors)
    - Validators only need to store/run data for the shard they’re validating, not the entire network – speeding things up and reducing hardware requirements
    - Rollups “layer 2” – allow dapps to bundle or roll up transactions to a single transaction off-chain, generate a cryptographic proof and then submit to chain (reducing the data needed for the TX)
    - The Beacon Chain contains all the logic for keeping shards secure and synced – coordinating stakers in the network, assigning them to shards they need to work on. Also facilitates comms between shards by receiving/storing shard TX data that is accessible by other shards. This gives shards a snapshot of state to keep everything up-to-date
    - Shard chains enter the ecosystem via proof-of-state consensus
  + Proof-of-stake
    - Participants stake their ETH to become validators
    - Validators perform validation similar to miners in PoW – ordering TXs, creating new blocks so that all nodes can agree on the state of the network
    - Benefits: lower barrier to entry through reduced hardware requirements, better energy efficiency, stronger immunity to centralisation, support for shard chains (distributed mining)
    - PoS – activates validators upon receipt of enough stake – currently 32 ETH
      * Validators are chosen at random to create blocks and are responsible for checking/confirming blocks they don’t create
      * Incentivises good behaviour by creating the penalty of losing stake for going offline (failing to validate) or deliberate collusion
    - PoS validators are selected randomly and are not competing, they just need to create blocks when chosen and validate proposed blocks when not – this validation is known as attesting e.g “this block looks good to me.”
      * If you attest malicious blocks, you lose your stake
      * How to differentiate legitimate blocks from malicious blocks?
    - The Beacon Chain (via PoS) unlocks the roadmap for 64 shard chains – each having a shared understanding of the state of the network
    - How validation works
      * TX submitted -> a validator is responsible for adding the TX to a shard block, where the validator is chosen randomly
      * Attestation – if a validator is not chosen, they’ll have to attest to another validator’s proposal and confirm that everything is legit. It is the attestation that is recorded in the beacon chain rather than the TX itself. **At least 128 validators are required to attest to each shard block – known as a ‘committee’, which is selected at random.** The committee has a time-frame to propose/validate a shard block, known as a ‘slot’. Only one valid block is created per slot, and there are 32 slots in an ‘epoch’ – after each epoch, 32 blocks are processed, the committee is disbanded and reformed with different, random participants.
      * Crosslinks – once a new shard block proposal has enough attestations, a ‘crosslink’ is created which confirms the inclusion of the block and your transaction in the beacon chain. Once this is formed, the validator who proposed the block gets their reward.
      * Finality – a TX has finality when it’s part of a block that cannot change
        + Casper, a finality protocol, gets validators to agree on the state of a block at certain checkpoints. **So long as 2/3 of validators agree, the block is finalised.** Validators will lose their entire stake if they try and revert this later via a 51% attack – would be like if a miner participates in 51% attack, all their mining hardware being burnt down
    - 51% attack risk still exists but is more risky for attackers
      * 51% attack means owning 51% of staked ETH – not only is this a lot of money, but it would probably cause ETH’s value to drop
        + There is very little incentive to destroy the value of a currency you have majority stake in
        + The incentive to play well is even greater
      * There will also be stake slashings, ejections and other penalties coordinated by the Beacon chain to prevent other acts of bad behaviour – validators will be responsible for flagging these incidents
  + The Merge
    - The point when Ethereum mainnet will merge with Beacon chain – marking the end of PoW for Ethereum

10/4/22 - Back to the video - <https://www.youtube.com/watch?v=M576WGiDBdQ&t=2015s>

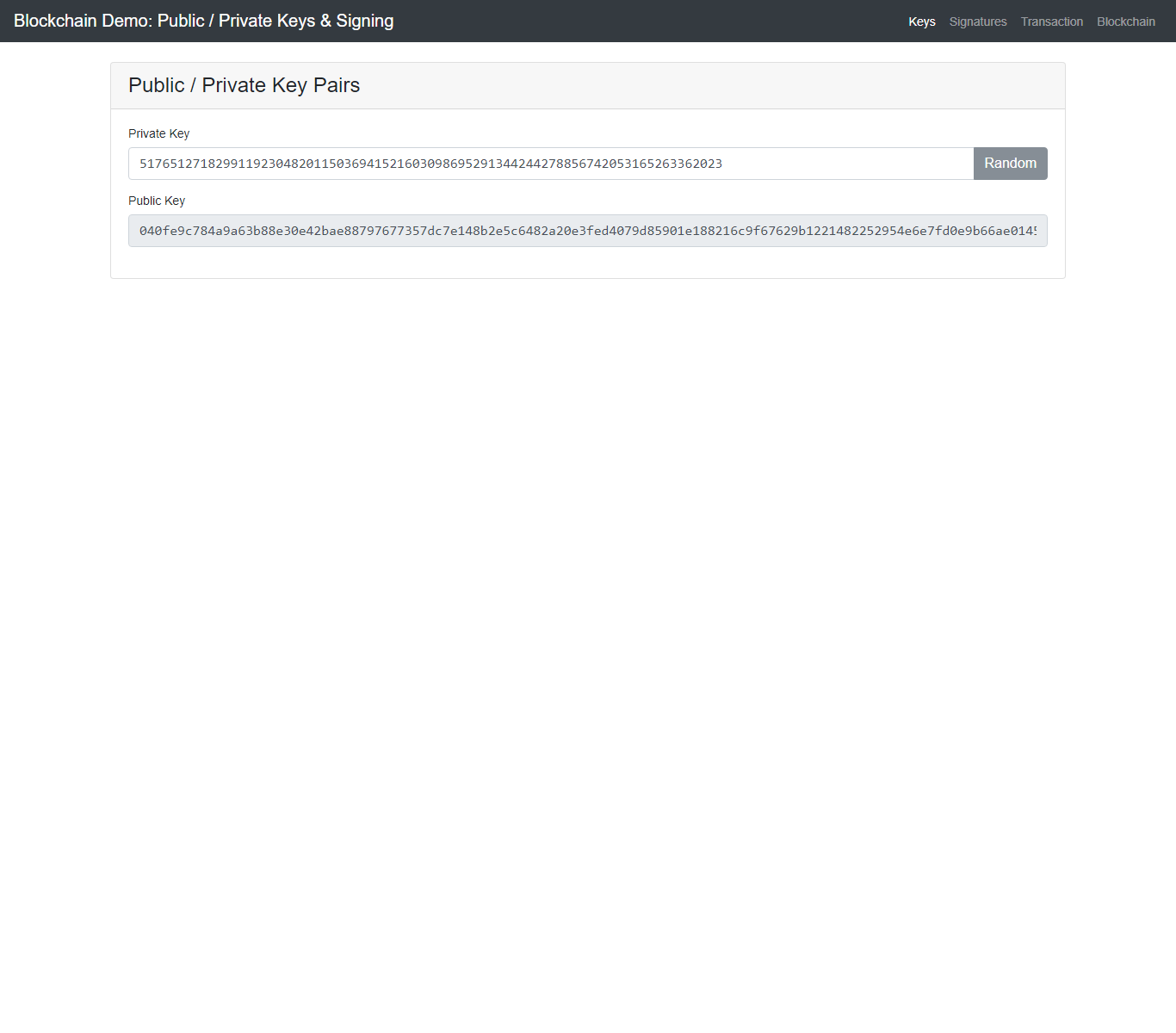
* Why blockchain?
  + Decentralised
  + Transparency & flexibility
  + Speed & efficiency
  + Security & immutability
  + Removal of counterparty risk
  + Trust minimised / trustless agreements
    - Insurance company – misaligned incentives between company and opt-in participants
  + Hybrid smart contracts – on and off-chain systems
* Ethereum blockchain interaction basics
  + Sending test ETH - <https://rinkebyfaucet.com/>
    - Check via rinkeby.etherscan.io - <https://rinkeby.etherscan.io/tx/0xdef78f24e31359a2460daffc36e9054eef3a80d2160135cec8b25a593413763f>
  + Elements on etherscan
    - TX hash
    - Block #
    - From / To addresses
    - Transaction value / transaction fee
    - Gas price / gas limit / gas fee
      * Gas limit – total gas that can be consumed in TX
      * Gas fee = gas required \* gas price
      * As determined by computational steps + byte size
    - There is a difference between computational fee / total transaction fee (miner reward)
      * Set the gas price to set the extent to which you incentivise miners
      * Do miner’s get any surplus remaining after computation up to the gas limit?? \*\*
        + Miners reach out for transactions from ‘mempools’ – mechanism that allows blockchains to store unconfirmed transactions. All valid TXs enter a mempool before being broadcast to miners (to be included into new blocks)
        + ‘Mempool sniping’ – tactics to get TXs broadcast into blocks as soon as possible
    - Checking gas prices - <https://ethgasstation.info/?currency=USD>
* Understanding the freakin’ blockchain – LFG
  + Hash functions - <https://andersbrownworth.com/blockchain/hash>
    - A hash is a unique, fixed length string used to identify a piece of data – created by placing the data into a ‘hash function’ e.g SHA256
    - Ethereum – Keccak256
    - From Haseeb Qureshi – a hash to have 3 characteristics
      * Deterministic – a hash function always produces the same output given a consistent input
      * Fixed output size
      * Uniform – hashes are distributed uniformly over the output space (incrementing the input by 1 will not linearly increase the output by 1)
      * One way – computationally intractable to invert the hash function and compute the preimage
      * Exhibit Avalanche effect – if any single bit changes, the rest gets jumbled. Two very similar inputs should have no discernible relationships between their outputs
      * Collision resistant – SHA256 – 2^256 possible outputs
  + SHA256 test



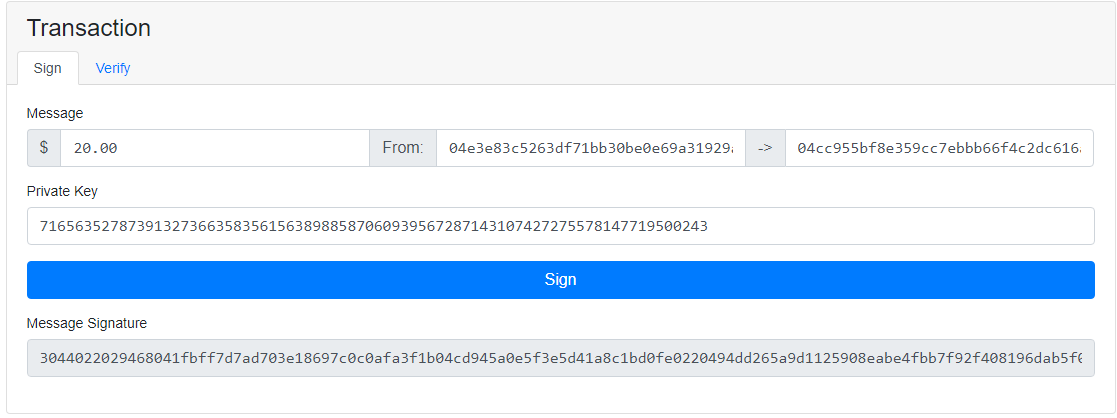
* Block / nonce / data -> hash function -> hash output
  + The problem that miners solve – given specific data and block information, what is the nonce that will give a hash output starting with 4 zeros?
  + There is also difficulty adjustment on block mining – ensuring 10 min / block output regardless of network computational power – dynamic hash rate determines quantity of frontrunning zeros
* 
* A blockchain – includes in addition to block #, nonce and data, the previous block’s hash
* 
* Then distributed:

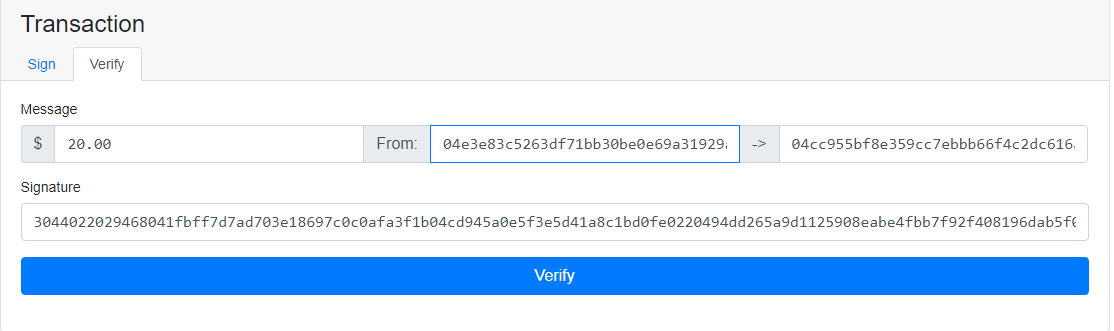


* Sender/recipient mechanics – public/private keys - https://andersbrownworth.com/blockchain/public-private-keys/keys



* + A private key is needed to sign and approve TX messages but it is impossible to decrypt a message signature back to the private key
  + Validation – made possible by using the public key + message signature
    - The unique public key has to match up with the unique message signature – so indirectly verifying ownership of the private key
    - For the message signature to be output, it is unique to both the message contained AND the private key information – so by crosschecking the TX is linked to the message signature, this is an indirect signal that the TX is valid?
    - Transactions steps:
      * Private key + message -> Signed transaction hash
      * Knowing the public key + message -> Crosscheck to the same signed transaction hash





* Popular blockchain consensus mechanisms (chain selection, Sybil resistance)
  + Proof of Work - Sybil resistance mechanism – preventing bad actors from creating fake nodes to declare “I did the work and deserve the reward!”
    - The necessity of computational power + difficulty adjustment
    - How to determine which is the truest chain?
      * BTC – Nakamoto consensus – longest chain wins, using # block confirmations to bet that the chain is true
    - Nakamoto consensus = PoW + longest chain rule
    - Block rewards – PoW rewards miners via rewards
      * Transaction fee (gas) vs block rewards
      * Block reward – pre-programmed quantity of rewards for securing blocks
    - Sybil attack – creating # pseudo-anonymous accounts
    - 51% attack – node overwhelm
  + Proof of Stake – Avalanche, Solana, Polkadot, Polygon
    - No miners, yes validators
    - PoS – different Sybil resistance mechanism
      * Staked ETH – skin in the game to become validator, rewards for good behaviour, penalties for bad behaviour
    - Validator nodes are randomly chosen to propose new blocks -> committees -> epochs -> reshuffle -> committees -> …
      * What is random? RanDAO
    - Remember: very easy to verify that a node / transaction is valid
    - Energy efficient – single node, mass validation
    - How decentralised is ‘decentralised enough’?
    - Scalability – more TX power, cheaper & more transactions – Sharding
      * Main chain + multiple side chains
    - Layers
      * L1 - Base layer blockchain implementations
      * L2 – Any application built on top of L1

Remix – Simple Storage

17 April 2022

[Remix IDE](https://remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.7+commit.e28d00a7.js)

Establishing a smart contract

* The doctype HTML equivalent – “pragma solidity 0.6.0”
* “Contract” – a keyword that defines a smart contract, similar to JS ‘class’
* [Types](https://docs.soliditylang.org/en/v0.8.6/types.html) - Uint256 (unsigned integer) / int256 / bool / string / address / bytes

A screenshot of a computer

Description automatically generated with medium confidence

Very similar structure to JavaScript (!!)

* Deploying contract

Graphical user interface, text, application

Description automatically generated

* “Public” keyword – visibility of variable or function to UI. Can be called either internally or via messages. External is the opposite. **The default setting is internal.**



Visibility - <https://docs.soliditylang.org/en/v0.7.3/contracts.html#visibility-and-getters>

* View and returns keyword in function declaration
  + View – reading some state off the blockchain
  + Pure – just does maths

Graphical user interface, application

Description automatically generated

* Creating ‘objects’ via struct keyword

Graphical user interface, text

Description automatically generated

struct People {

        uint256 favouriteNumber;

        string name;

    }

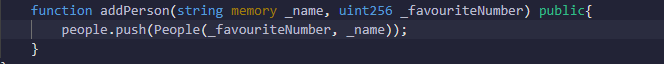
    People public person = People({favouriteNumber: 2, name: 'Patrick'});

The ‘People’ is the class, public the visibility and person the name of the new variable being declared

A screenshot of a computer

Description automatically generated

* Memory keyword – data will only be stored during execution of the function
  + Vs ‘storage’ – persistent storage



* Inputting values into the contract – note how “Patrick” and 2 have indexes 1 and 0 respectively, yet they are accepted as opposite in the input field

Graphical user interface

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

* “Mapping” keyword – data structure mapping to store keys & values
  + Mapping(keyType => valueType) public mappingVariable
* mapping(string => uint256) public nameToFavouriteNumber;
* Finished contract
* //SPDX-License-Identifier: MIT
* pragma solidity >=0.6.0 <0.9.0;
* contract SimpleStorage{
* // Not necessary to initialise explicitly, defaults to 0
* uint256 favouriteNumber;
* struct People {
* uint256 favouriteNumber;
* string name;
* }
* People[] public people;
* mapping(string => uint256) public nameToFavouriteNumber;
* function store(uint256 \_favouriteNumber) public {
* favouriteNumber = \_favouriteNumber;
* }
* function retrieve() public view returns(uint256){
* return favouriteNumber;
* }
* function addPerson(string memory \_name, uint256 \_favouriteNumber) public{
* people.push(People(\_favouriteNumber, \_name));
* nameToFavouriteNumber[\_name] = \_favouriteNumber;
* }
* }
* Deploying to Testnet
  + <https://rinkeby.etherscan.io/tx/0xc4ba5645f0ad48cb2ee1644d4cf5e4646404d725672eb8af1f16fccc6f3925c5>
* A screenshot of a computer

  Description automatically generated with medium confidence
  + Note how people / nameToFavourite / retrieve are view only – requires no gas to obtain the info from the contract
    - Only addPerson and store requires gas – due to state change via interaction with the smart contract
* “EVM” – Ethereum Virtual Machine
  + Before our contract got deployed live, it all got compiled down to EVM
  + “EVM compatible” – other chains can accept Solidity EVM compatible contracts

Storage Factory

* Importing other smart contracts
  + Text

    Description automatically generated
  + Notice SimpleStorage – class declaration, new keyword just like ES6 classes declaration

24 April 2022

Final code – storageFactory.sol

// SPDX-License-Identifier: MIT

pragma solidity >=0.6.0;

//Import SimpleStorage.sol contract

import "./SimpleStorage.sol";

contract StorageFactory {

    SimpleStorage[] public simpleStorageArray;

    function createSimpleStorageContract() public {

        SimpleStorage simpleStorage = new SimpleStorage();

        simpleStorageArray.push(simpleStorage);

    }

    function sfStore(uint256 \_simpleStorageIndex, uint256 \_simpleStorageNumber) public {

        // Address

        // ABI - Application Binary Interface

        //Retrieve the contract address at specific index

        // SimpleStorage simpleStorage = SimpleStorage(address(simpleStorageArray[\_simpleStorageIndex]));

        // simpleStorage.store(\_simpleStorageNumber);

        SimpleStorage(address(simpleStorageArray[\_simpleStorageIndex])).store(\_simpleStorageNumber);

    }

    function sfGet(uint256 \_simpleStorageIndex) public view returns (uint256) {

        // SimpleStorage simpleStorage = SimpleStorage(address(simpleStorageArray[\_simpleStorageIndex]));

        // return simpleStorage.retrieve();

        return SimpleStorage(address(simpleStorageArray[\_simpleStorageIndex])).retrieve();

    }

}

* Method inheritance across ‘classes’ via ‘is’ keyword – just like ‘extends’ keyword in JS

A screenshot of a computer

Description automatically generated with medium confidence

Fund me

24 April 2022

* ‘payable’ keyword
  + Function to be used specifically for payments – every function has a ‘value’ parameter
* Mapping keyword, msg.sender, msg.value – funding smart contracts
* // SPDX-License-Identifier: MIT
* pragma solidity >=0.6.6;
* contract FundMe {
* mapping(address => uint256) public addressToAmountFunded;
* function fund() public payable {
* addressToAmountFunded[msg.sender] += msg.value;
* }
* }
* How to convert other currencies into Eth?
  + Smart contracts cannot interact with external systems (via APIs etc) due to being deterministic “walled gardens” – this is a necessary feature of the consensus mechanism between nodes
    - Calling a “random” number on one node would be misaligned to a “random” number on another
  + **Centralised oracles are prone to attack/failure**

Diagram

Description automatically generated

* Creating a live contract price oracle

A screenshot of a computer

Description automatically generated

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";

contract PriceConsumerV3 {

    AggregatorV3Interface internal priceFeed;

    /\*\*

     \* Network: Kovan

     \* Aggregator: ETH/USD

     \* Address: 0x9326BFA02ADD2366b30bacB125260Af641031331

     \*/

    constructor() {

        priceFeed = AggregatorV3Interface(0x9326BFA02ADD2366b30bacB125260Af641031331);

    }

    /\*\*

     \* Returns the latest price

     \*/

    function getLatestPrice() public view returns (int) {

        (

            /\*uint80 roundID\*/,

            int price,

            /\*uint startedAt\*/,

            /\*uint timeStamp\*/,

            /\*uint80 answeredInRound\*/

        ) = priceFeed.latestRoundData();

        return price;

    }

}

//Importing via chainlink/contracts npm repo

import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";

* Application Binary Interface
  + “Interfaces” compile down into ABIs – tells Solidity what functions can be called on another contract
  + **You will need an ABI anytime you interact with an already established smart contract**
* Aggregator V3 interface

<https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol>

Text

Description automatically generated

* Tuples – returning multiple data streams
  + A list of objects of potentially different types whose number is constant at compile time
* function getPrice() public view returns (uint256){
* AggregatorV3Interface priceFeed = AggregatorV3Interface(0x8A753747A1Fa494EC906cE90E9f37563A8AF630e);
* (uint80 roundId,
* int256 answer,
* uint256 startedAt,
* uint256 updatedAt,
* uint80 answeredInRound
* ) = priceFeed.latestRoundData();
* //Need to convert int256 to uint256
* return uint256(answer);
* }

AggregatorInterfaceV3

A picture containing graphical user interface

Description automatically generated

* Cleaning up the tuples
* (,int256 answer,,,) = priceFeed.latestRoundData();
* Math limitations in Solidity

A screenshot of a computer

Description automatically generated with medium confidence

* If the value exceeds the ‘class size’ declaration, >=v0.8.0 makes sure that the value is captured and avoids the final result from ‘wrapping’ (e.g starting from 0) – protecting from overflows/underflows
* Importing SafeMath - [https://docs.openzeppelin.com/contracts/4.x/utilities#api:math.adoc#SafeMath](https://docs.openzeppelin.com/contracts/4.x/utilities#api:math.adoc)
  + “using” and “for” keywords
  + A library is similar to contracts but they are isolated code that is deployed only once at a specific address and their code is reused

Graphical user interface, text, application

Description automatically generated

* Require keyword – check for lower threshold of value
* function fund() public payable {
* uint256 minimumUSD = 50 / getPrice();
* require(getConversionRate(msg.value) >= minimumUSD, "You need to spend more ETH");
* addressToAmountFunded[msg.sender] += msg.value;
* //Conversion rates of other currencies
* }

26 April 2022

* Withdraw function
  + Msg.sender and this keyword
* function withdraw() payable public {
* msg.sender.transfer(address(this).balance);
* }
* How to establish proper ‘ownership’ of the smart contract?
  + Require keyword
  + Constructor keywod
* address public owner;
* //Constructor keyword - runs on deployment
* constructor() {
* owner = msg.sender;
* }
  + Ran into bug – address that is being sent to needs to be of type ‘address payable’
* function withdraw() payable public {
* require(msg.sender == owner);
* address payable msgSender = payable(msg.sender);
* msgSender.transfer(address(this).balance);
* }
* Resetting after full withdrawal
* // Set funders balances to 0
* for (uint256 funderIndex = 0; funderIndex < funders.length; funderIndex++){
* address funder = funders[fundersIndex];
* addressToAmountFunded[funder] = 0;
* }
* //Empty out funders array
* funders = new address[](0);

Lesson 4 - Web3.py Simple Storage

* Remix – starting ground as a Web IDE – Integrated Development Environment
  + Web based
  + Modules for testing, debugging, deploying
  + Local JS VM to simulate Ethereum
  + Quick and easy to create & test smart contracts
* Connecting into Python – web3.py
  + Contract deployment in VS via Python
  + At least a couple hours troubleshooting the lack of autosave with black formatter

*//Python*

  "[python]": {

    "editor.defaultFormatter": null,

    "editor.formatOnSave": true

  }

* Installing compiller – pip install py-solc-x
  + More bugs from the solc install…. 🙄

Graphical user interface, text, application, email

Description automatically generated

* Retrieving byte code of contract
* *# get bytecode of the contract*
* bytecode = compiled\_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["evm"]["bytecode"]["object"]

Text

Description automatically generated

* Retrieving abi

Text

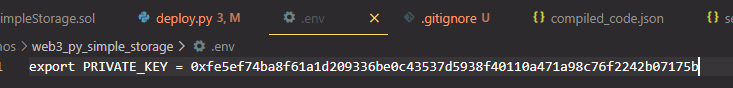
Description automatically generated

* Test deploying via ‘JavaScript EVM’ via Ganache

2 May 2022

Build contract deploy / sign / send transaction

* Obfuscating private key via os and dotenv
* from dotenv import load\_dotenv
* import os



* buildTransaction method
  + Found bugs when maxFeePerGas and maxPriorityFeePerGas were not in input variables

Text

Description automatically generated

* More bugs with send\_raw\_transaction() 🤢

Questions:

* Blockchain oracle -?
  + Reputation system to vet and bridge ‘real world’ connections into smart contracts
  + Chainlink – connecting smart contracts with off-chain data and services
  + Results in hybrid smart contracts
* Who holds the keys to smart contracts?