Learning Blocks

Rashi Jaiswal

Smart Contracts

Features of Smart Contracts

- A piece of computer code between two or more parties
- Runs on top of a blockchain
- When the set of predefined rules is met, the smart contract executes itself to produce the output
- Allows you to exchange anything of value including money, shares, property etc, in a transparent manner, thus eliminating the need for a middleman

Unpacking the definition

- **Computer programs**- Smart contracts are simply computer programs. The word “contract” has no legal meaning in this context.

- **Immutable**- Once deployed, the code of a smart contract cannot change. To modify a smart contract one needs to deploy a new instance.

- **Deterministic**- The outcome of the execution of a smart contract is the same for everyone who runs it, given the context of the transaction that initiated its execution and the state of the Ethereum blockchain at the moment of execution.

- **EVM context**- Smart contracts operate with a very limited execution context. They can access their own state, the context of the transaction that called them, and some information about the most recent blocks.

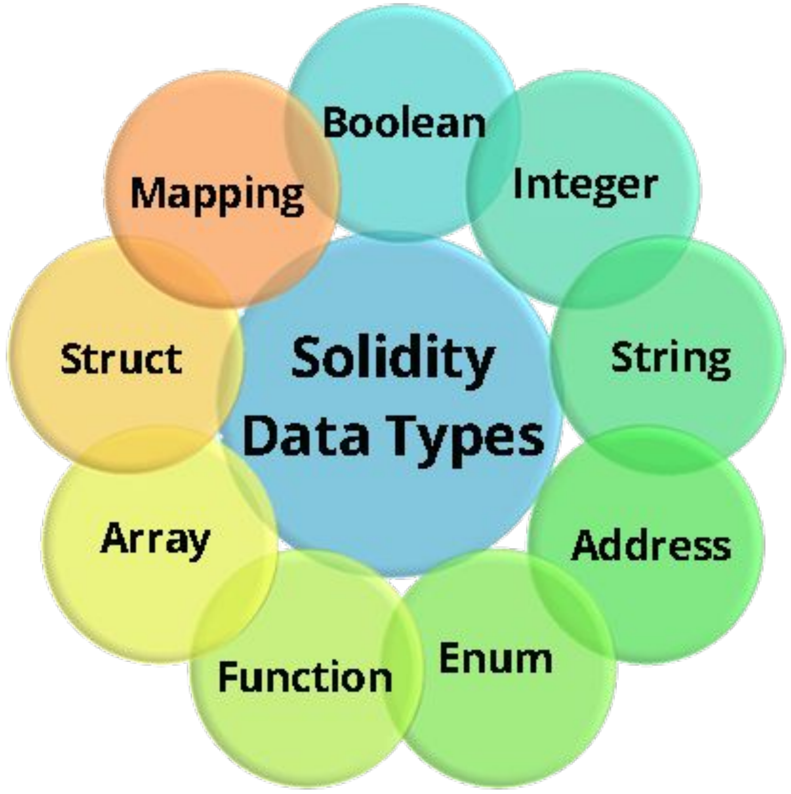
- **Decentralized world computer**- The EVM runs as a local instance on every Ethereum node, but because all instances of the EVM operate on the same initial state and produce the same final state, the system as a whole operates as a single "world computer."

The ABC of Solidity

https://learnxinyminutes.com/docs/solidity/

- Data Types
- Predefined global variables and functions
- Functions and Modifiers
- Events

Data Types



- **Fixed point (fixed, ufixed)-** Fixed-point numbers, declared with `(u)fixedMxN` where M is the size in bits (increments of 8 up to 256) and N is the number of decimals after the point (up to 18); e.g., ufixed32x2.

- **Address-** A 20-byte Ethereum address. The address object has many helpful member functions, the main ones being balance (returns the account balance) and `transfer` (transfers ether to the account).

- **Byte array (fixed)-** Fixed-size arrays of bytes, declared with bytes1 up to bytes32. **Byte array (dynamic)-** Variable-sized arrays of bytes, declared with bytes or string.

- **Enum-** User-defined type for enumerating discrete values: enum NAME {LABEL1, LABEL 2, ...}.

- **Mapping-** Hash lookup tables for key => value pairs: mapping(KEY_TYPE => VALUE_TYPE) NAME.

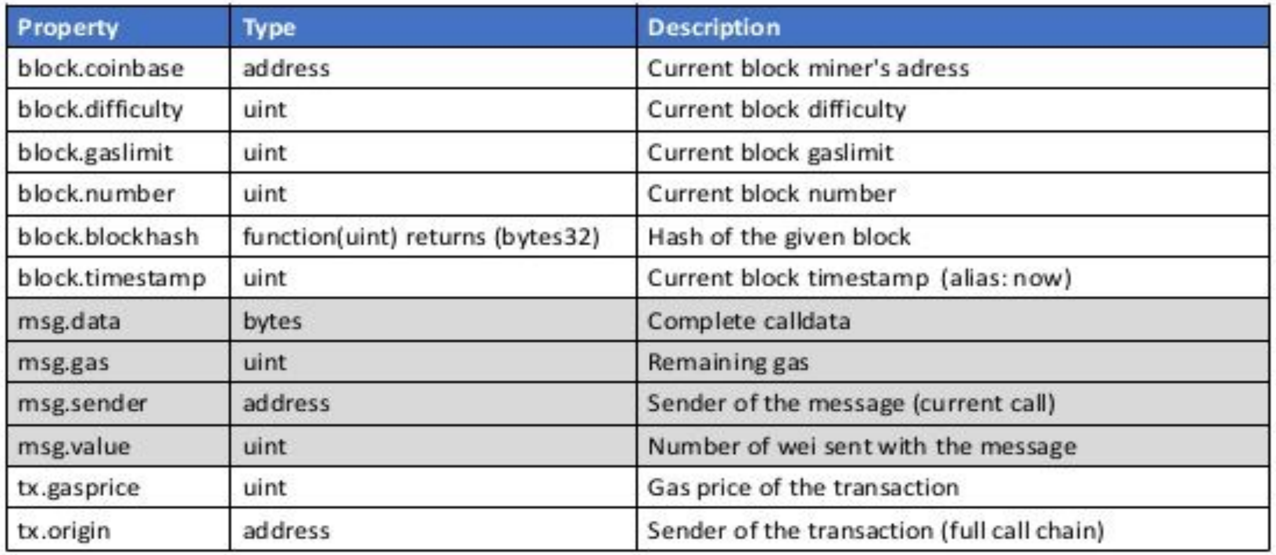
Data Types (Contd.)

- Solidity also offers a variety of value literals that can be used to calculate different units:
  - ***Time units***- The units seconds, minutes, hours, and days can be used as suffixes, converting to multiples of the base unit seconds.
  - ***Ether units***- The units wei, finney, szabo, and ether can be used as suffixes, converting to multiples of the base unit wei.

Eg.

```
require(withdraw_amount <= 100000000000000000)  // 10^17
require(withdraw_amount <= 0.1 ether)
```

Predefined global variables



Property	Type	Description
block.coinbase	address	Current block miner's adress
block.difficulty	uint	Current block difficulty
block.gaslimit	uint	Current block gaslimit
block.number	uint	Current block number
block.blockhash	function(uint) returns (bytes32)	Hash of the given block
block.timestamp	uint	Current block timestamp (alias: now)
msg.data	bytes	Complete calldata
msg.gas	uint	Remaining gas
msg.sender	address	Sender of the message (current call)
msg.value	uint	Number of wei sent with the message
tx.gasprice	uint	Gas price of the transaction
tx.origin	address	Sender of the transaction (full call chain)

Functions and Modifiers

The syntax we use to declare a function in Solidity is as follows:

```
function FunctionName([parameters]) {public|private|internal|external}
[pure|constant|view|payable] [modifiers] [returns (return types)]
```

The following set of keywords specify the function's *visibility*:

public- Public is the default; such functions can be called by other contracts or EOA transactions, or from within the contract.

external- External functions are like public functions, except they cannot be called from within the contract unless explicitly prefixed with the keyword this.

internal- Internal functions are only accessible from within the contract—they cannot be called by another contract or EOA transaction. They can be called by derived contracts (those that inherit this one).

private- Private functions are like internal functions but cannot be called by derived contracts.

Functions (Contd.)

The next set of keywords (pure, constant, view, payable) affect the behavior of the function:

constant or view- A function marked as a *view* promises not to modify any state.

pure- A pure function is one that neither reads nor writes any variables in storage. It can only operate on arguments and return data, without reference to any stored data. Pure functions are intended to encourage declarative-style programming without side effects or state.

payable- A payable function is one that can accept incoming payments. Functions not declared as payable will reject incoming payments. There are two exceptions, due to design decisions in the EVM: coinbase payments and SELFDESTRUCT inheritance will be paid even if the fallback function is not declared as payable, but this makes sense because code execution is not part of those payments anyway.

Events

- Events are inheritable members of contracts.
- When you call them, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible.
- The Log and its event data is not accessible from within contracts (not even from the contract that created them).

```
pragma solidity >=0.4.21 <0.6.0;

contract ClientReceipt {

        event Deposit(address indexed _from, bytes32
indexed _id, uint _value);

        function deposit(bytes32 _id) public payable {
/* Events are emitted using `emit`, followed by the name
of the event and the arguments (if any) in parentheses. Any
such invocation (even deeply nested) can be detected from
the JavaScript API by filtering for `Deposit`. */

        emit Deposit(msg.sender, _id, msg.value);
         }
}
```

Contract Checkpoints

Gas Considerations (The fuel of Ethereum)

Gas is a resource limiting the maximum amount of computation that can happen in a transaction. If the gas limit is exceeded during computation, the following series of events occurs:

- An "out of gas" exception is thrown.
- The state of the contract prior to execution is restored (reverted).
- All ether used to pay for the gas is taken as a transaction fee; it is *not* refunded.

There are certain practices that are recommended when constructing smart contracts, so as to minimize the gas cost of a function call.

Avoid Dynamically Sized Arrays- Any loop through a dynamically sized array where a function performs operations on each element or searches for a particular element introduces the risk of using too much gas.

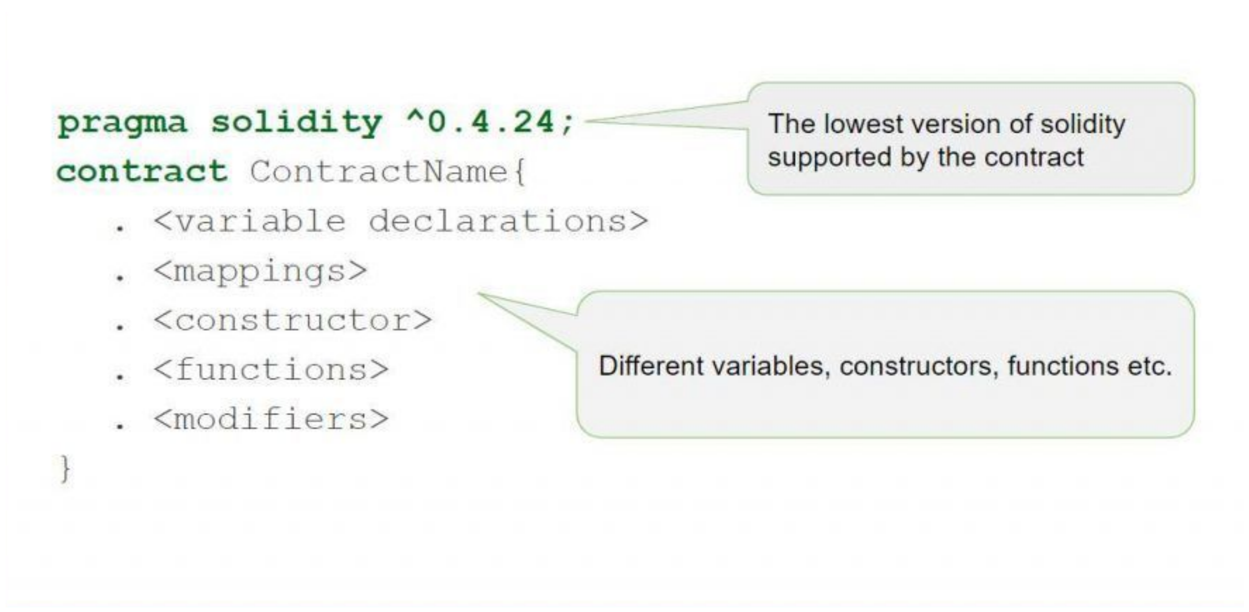
Avoid Calls to Other Contracts- Calling other contracts, especially when the gas cost of their functions is not known, introduces the risk of running out of gas. Avoid using libraries that are not well tested and broadly used.

Estimating Gas Cost- If you need to estimate the gas necessary to execute a certain method of a contract considering its arguments, you could use the following procedure:

```
var contract = web3.eth.contract(abi).at(address);

var gasEstimate =
contract.myAweSomeMethod.estimateGas(arg1, arg2,
{from: account});
```

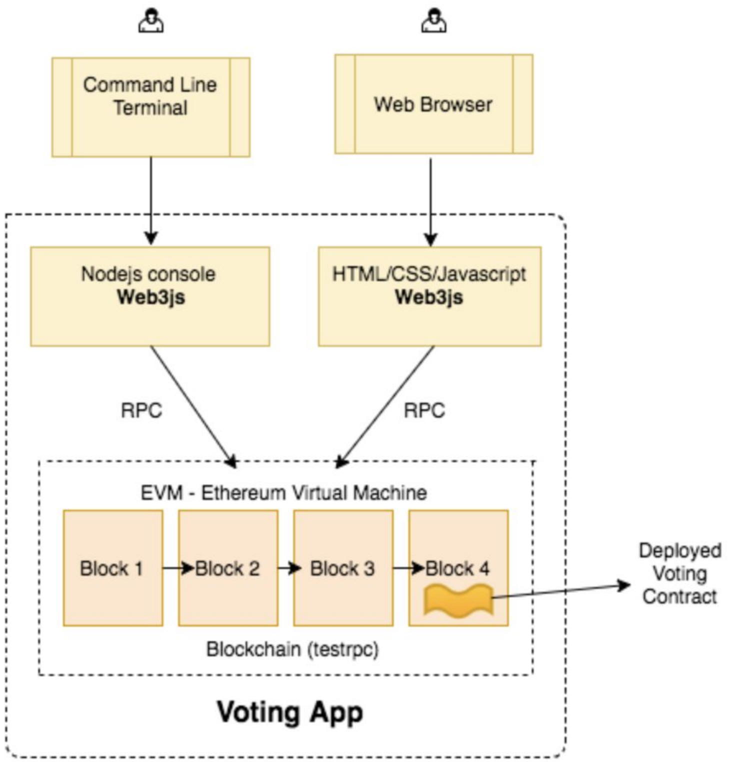
Summarised Contract



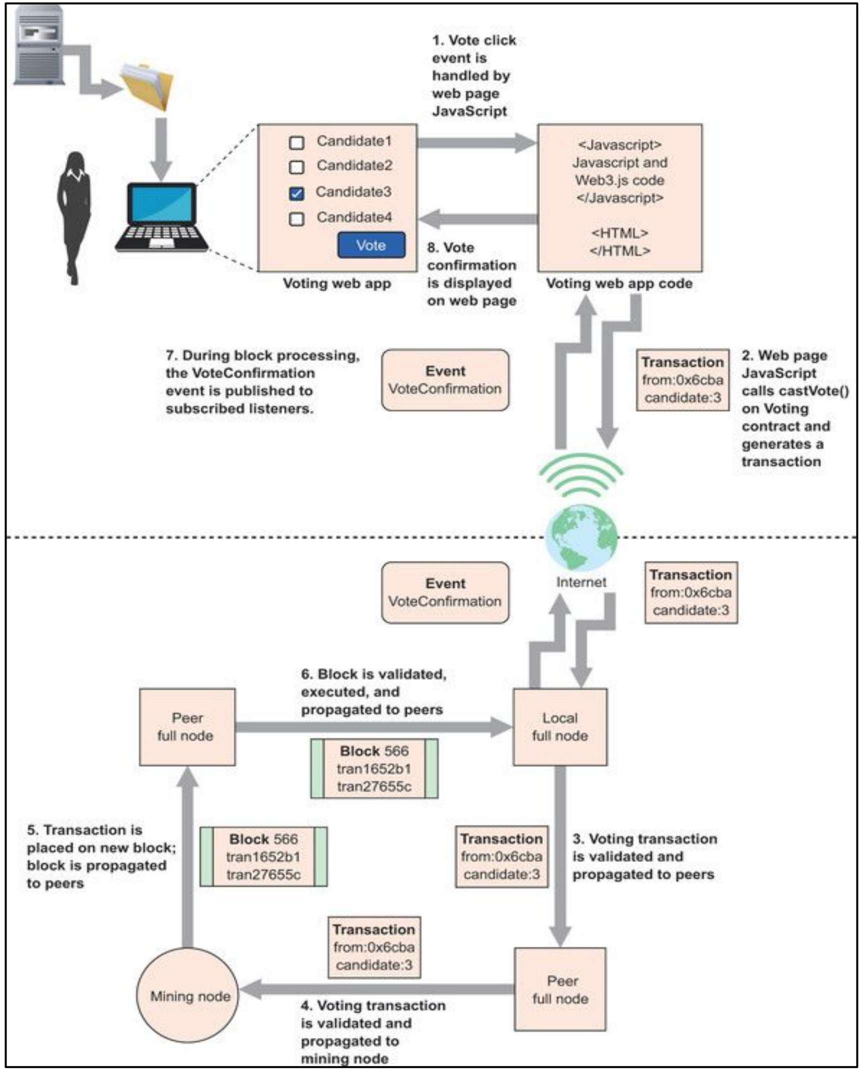
```
pragma solidity ^0.4.24;
contract ContractName{
    . <variable declarations>
    . <mappings>
    . <constructor>
    . <functions>
    . <modifiers>
}
```

The lowest version of solidity supported by the contract
Different variables, constructors, functions etc.

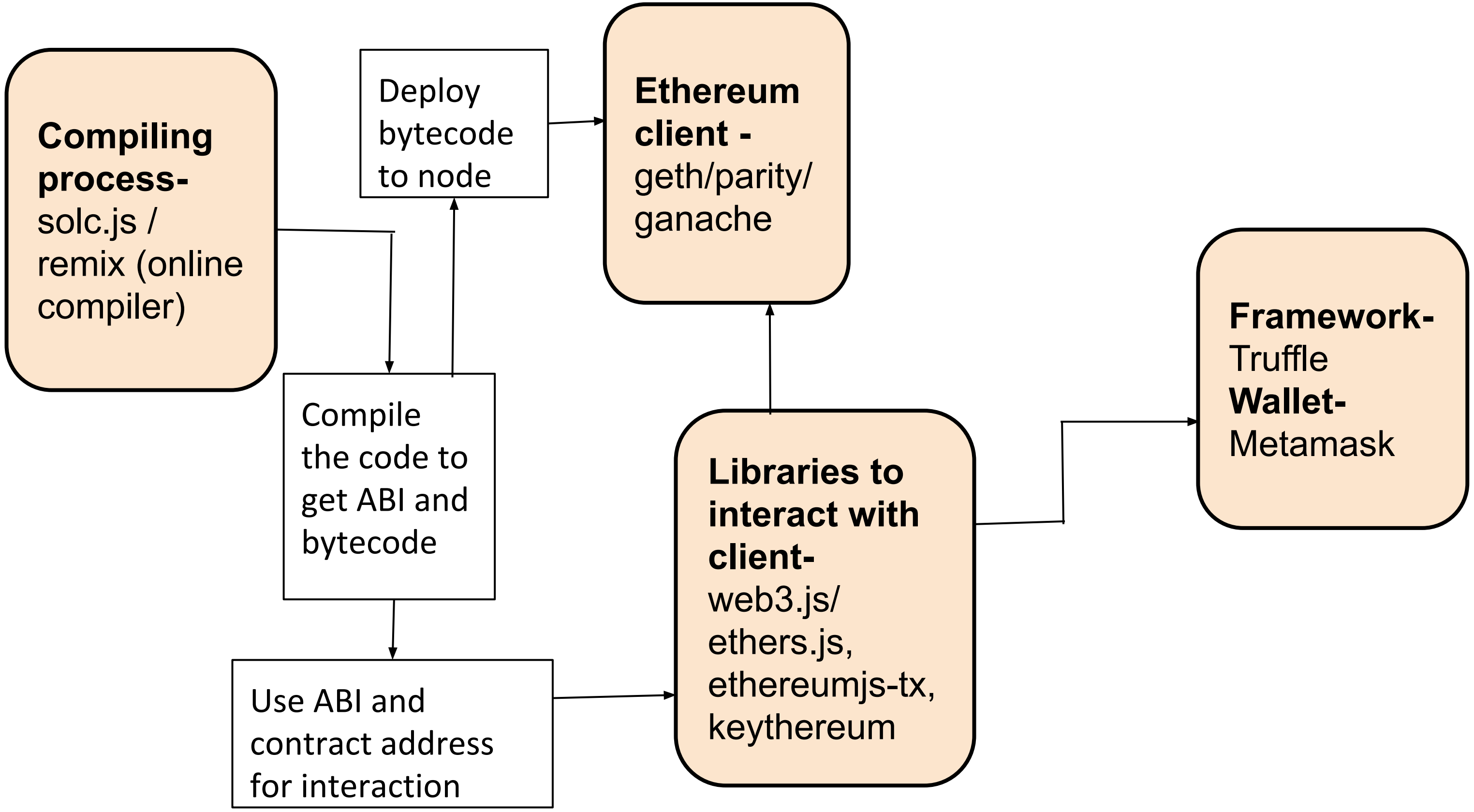
Where exactly is the contract? How do I talk to it?



Lifecycle of Voting Transaction

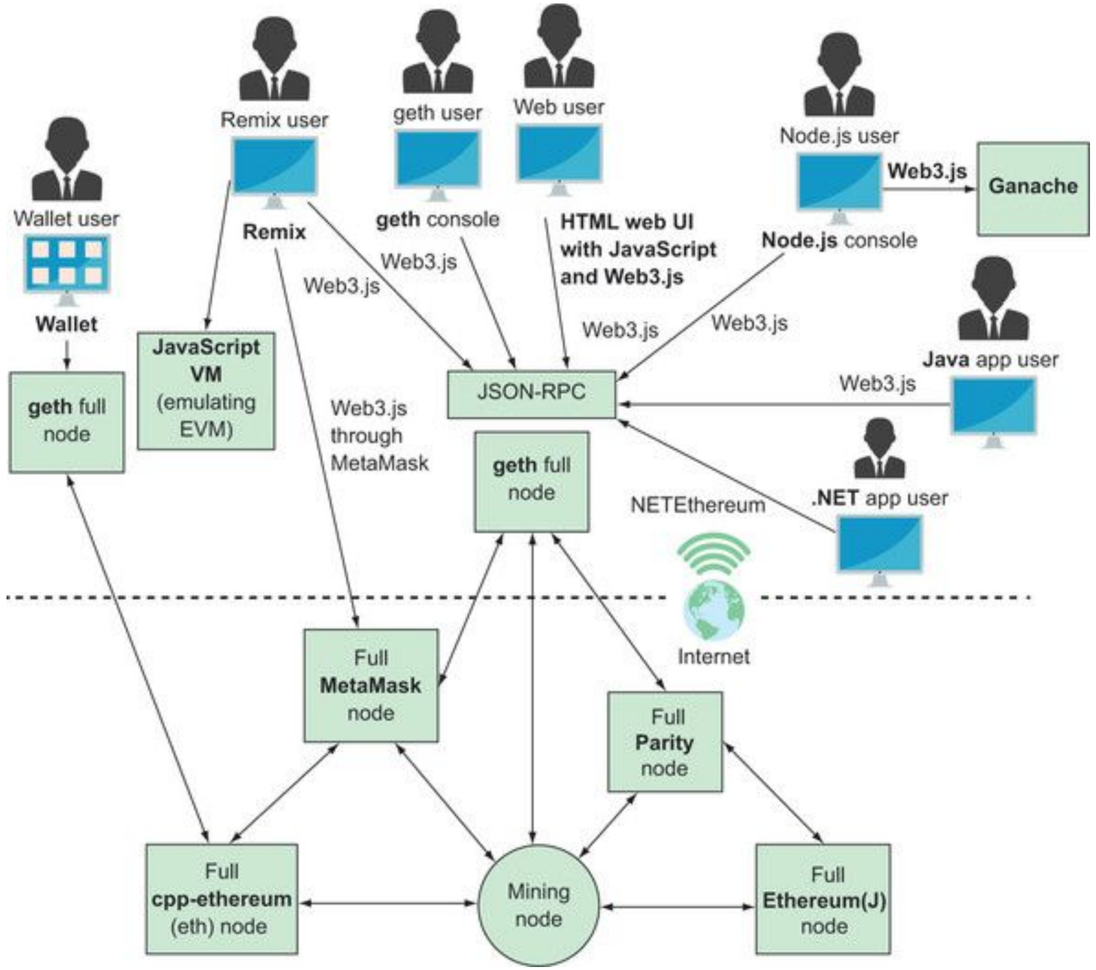

1. Vote click event is handled by web page JavaScript
Candidate1
Candidate2
Candidate3
Candidate4
Vote
Voting web app
<Javascript> Javascript and Web3.js code </Javascript>
<HTML> </HTML>
Voting web app code
8. Vote confirmation is displayed on web page
7. During block processing, the VoteConfirmation event is published to subscribed listeners.
Event VoteConfirmation
Transaction from:0x6cba candidate:3
2. Web page JavaScript calls castVote() on Voting contract and generates a transaction
Internet
Event VoteConfirmation
Transaction from:0x6cba candidate:3
6. Block is validated, executed, and propagated to peers
Peer full node
Local full node
Block 566 tran1652b1 tran27655c
5. Transaction is placed on new block; block is propagated to peers
Block 566 tran1652b1 tran27655c
Transaction from:0x6cba candidate:3
3. Voting transaction is validated and propagated to peers
Transaction from:0x6cba candidate:3
Mining node
Peer full node
4. Voting transaction is validated and propagated to mining node

Components of DApp Development



Compiling process- solc.js / remix (online compiler)
Deploy bytecode to node
Ethereum client - geth/parity/ ganache
Compile the code to get ABI and bytecode
Framework- Truffle **Wallet-** Metamask
Libraries to interact with client- web3.js/ ethers.js, ethereumjs-tx, keythereum
Use ABI and contract address for interaction

The Ecosystem



References

https://github.com/ethereumbook/ethereumbook/blob/develop/07smart-contracts-solidity.asciidoc#what-is-a-smart-contract

https://solidity.readthedocs.io/en/latest/contracts.html

https://medium.com/@mvmurthy/full-stack-hello-world-voting-ethereum-dapp-tutorial-part-1-40d2d0d807c2

https://livebook.manning.com/book/building-ethereum-dapps/