

Ethereum Development

Introduction to Ethereum

Development Tools

Writing Contracts, Tests and Deployment

Background



Matt Lockyer

Soloblock Solutions Inc.

ERC-998 Composable NFTs

Blockchain and Cryptography

<https://medium.com/@mattdlockyer>

<https://twitter.com/mattdlockyer>

<https://linkedin.com/in/mattlockyer>

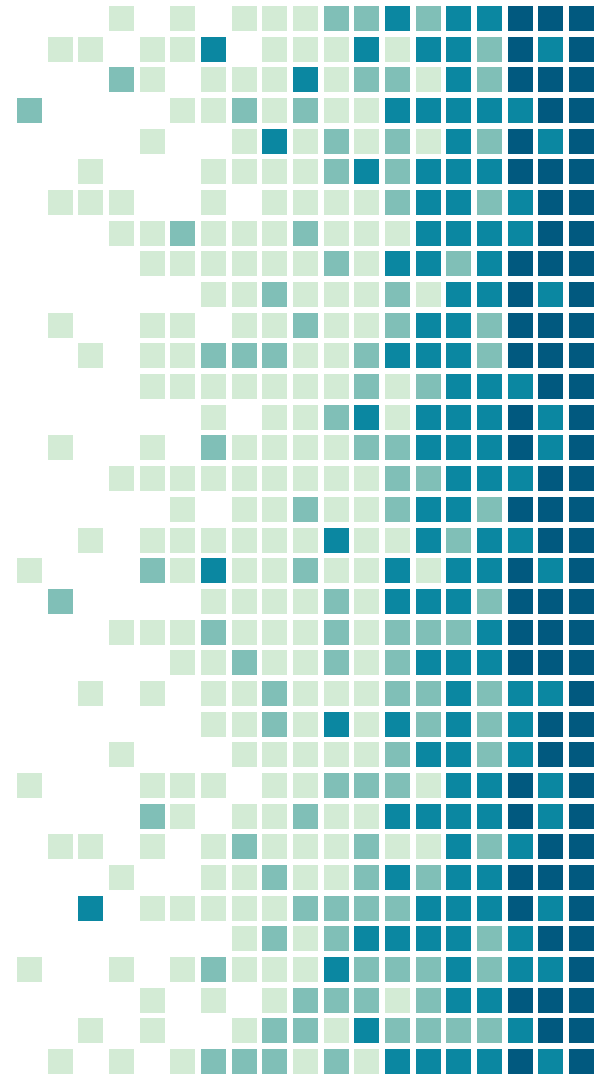
Part 1 Overview

1. How Ethereum Works
2. Smart Contracts
3. Development Environment

1.

Nuts and Bolts

Ethereum Blockchain in Depth



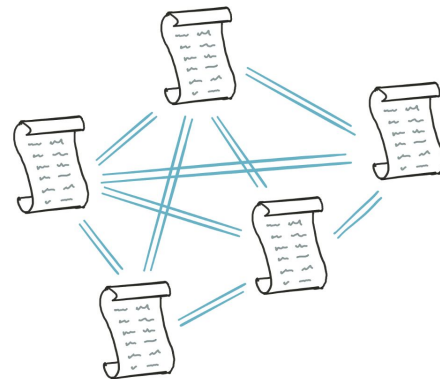
The Ethereum Blockchain

Other Blockchains / Technologies

- Bitcoin - limited script and accounts logic
- Distributed Ledger Technology - private databases
- Polkadot, Libra, Rootstock, etc...

Ethereum is public, immutable, decentralized application state

- Anyone can join, publish smart contracts, transact, read state
- Updates are transactional, verified by merkle proofs
- Quasi-turing complete “global computer”



Technical Details

Ether (ETH) is the settlement currency

10^{18} Wei = 1 ETH ~ \$??? USD <https://coinmarketcap.com/>

Computation uses Gas <https://ethgasstation.info/> <https://converter.murkin.me/>

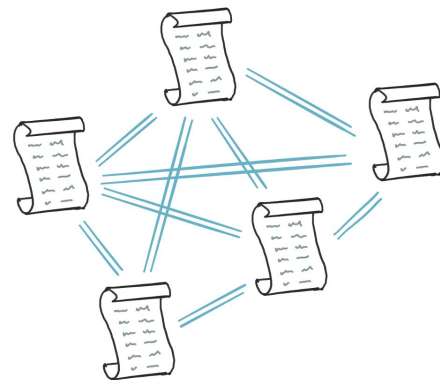
Charging for computation: a solution to the halting problem (quasi-turing complete)

EVM: 256 bit virtual machine - stack based to a depth of 1024

Storage: every contract has its own key:value database, unlimited if you pay for it

Languages: Solidity (JavaScript), Serpent (python), LLL (lua)

Networks: mainnet (real) - testnets - private networks

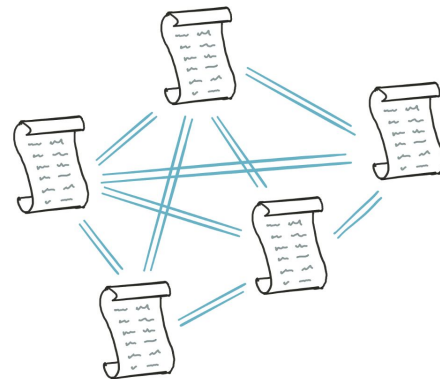


Gas Costs

APPENDIX G. FEE SCHEDULE

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero} .
G_{base}	2	Amount of gas to pay for operations of the set W_{base} .
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low} .
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid} .
G_{high}	10	Amount of gas to pay for operations of the set W_{high} .
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{suicide}$	24000	Refund given (added into refund counter) for suiciding an account.
$G_{suicide}$	5000	Amount of gas to pay for a SUICIDE operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SUICIDE operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	10	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the <i>Homestead transition</i> .
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatenonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.



Yellow Paper
Highly Technical Overview of the EVM
<http://yellowpaper.io/>

Accounts and Transactions

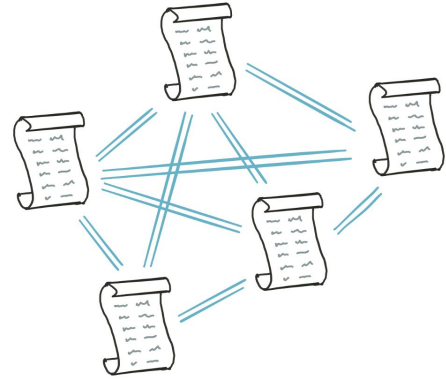
Two different account types: EOA and Contract

Externally Owned Accounts (EOA)

- This is your wallet (like in Bitcoin) controlled by private key
- Contains nonce, balance, storageRoot
- Transaction cost = 21,000 gas

Contract Accounts (EOA + code)

- Can contain, hold, send, restrict access to ETH
- Transactions execute code, execution costs gas
- Sending ETH to contracts could be more costly than EOA to EOA



Transactions

Transactions contain

- recipient, signature, value, gas limit, gas price, [data]

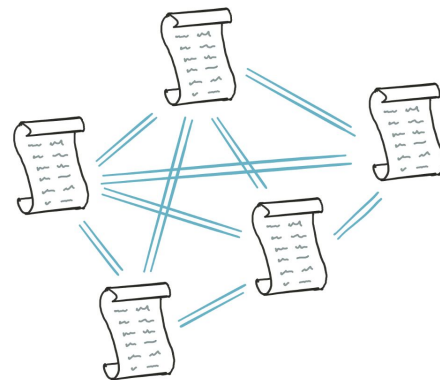
Transaction lifecycle

- Local / Node: created, signed, sent to node, hashed, broadcast
- All Nodes: include in block, new state / events created, mining
- Winner: broadcast, verified, update state + events, confirmation(s)

Transaction IDs are a 256bit hash of the transaction for tracking

Example

- <https://etherscan.io/tx/0x93951ca835179683d5c10945e76ac585b4c5e34ed56b4e44b158f8bbea664a01>



Messages

Contract to contract, virtual objects in EVM, never serialized

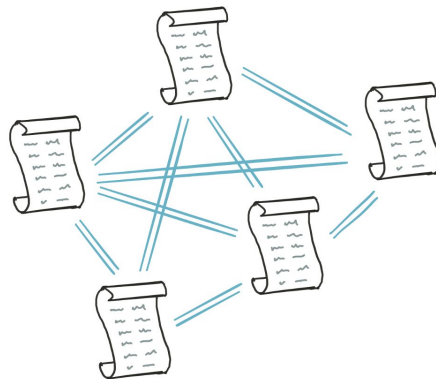
Messages contain

- sender, recipient, gas limit, data

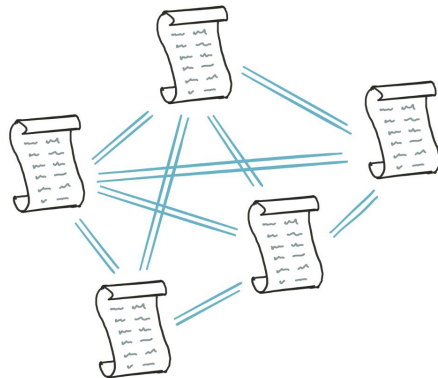
Example EtherDelta Trade function

- <https://etherscan.io/tx/0xbb2d0d8909138a4fe9fc7f5cde25c7107aa9c95a46d0eaf09e2545e3482bfa09>

What goes in the data?



Dissecting a Function Call



```
transfer(address _to, uint256 _value)
    example smart contract function
```

example function call

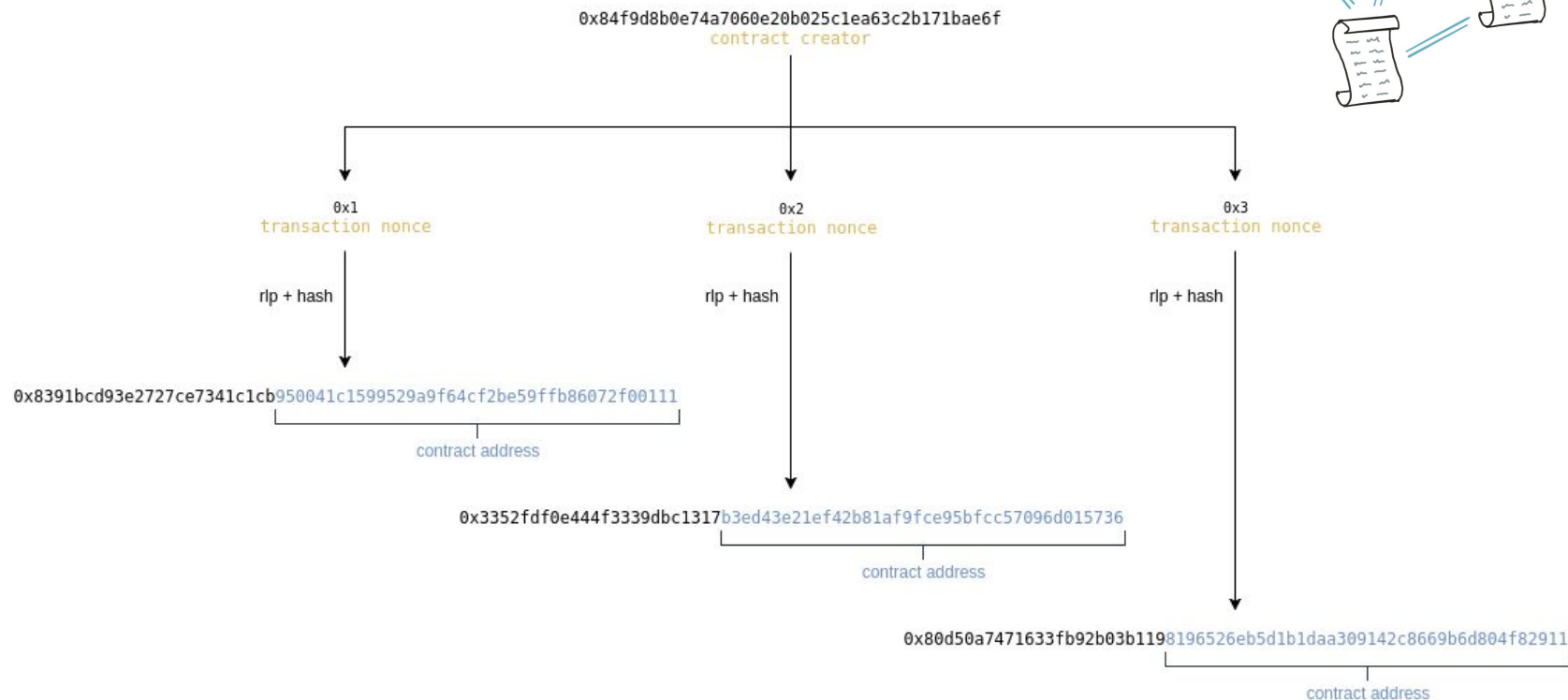
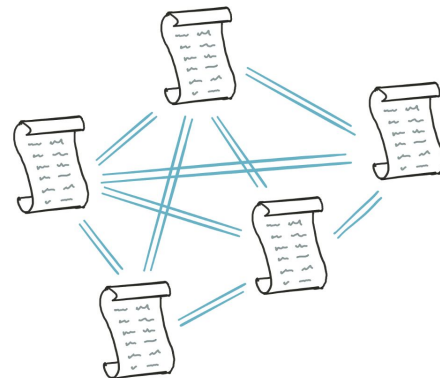
[illegible]

0x7adee867ea91533879d083dd47ea81f0eee3a37e
address to

```
0x2ab486cedbffff
uint256 value
```

```
0xa9059cbb2ab09eb219583f4a59a5d0623ade346d962bcd4e46b11da047c9049b
      hash of "transfer(address,uint256)"
```

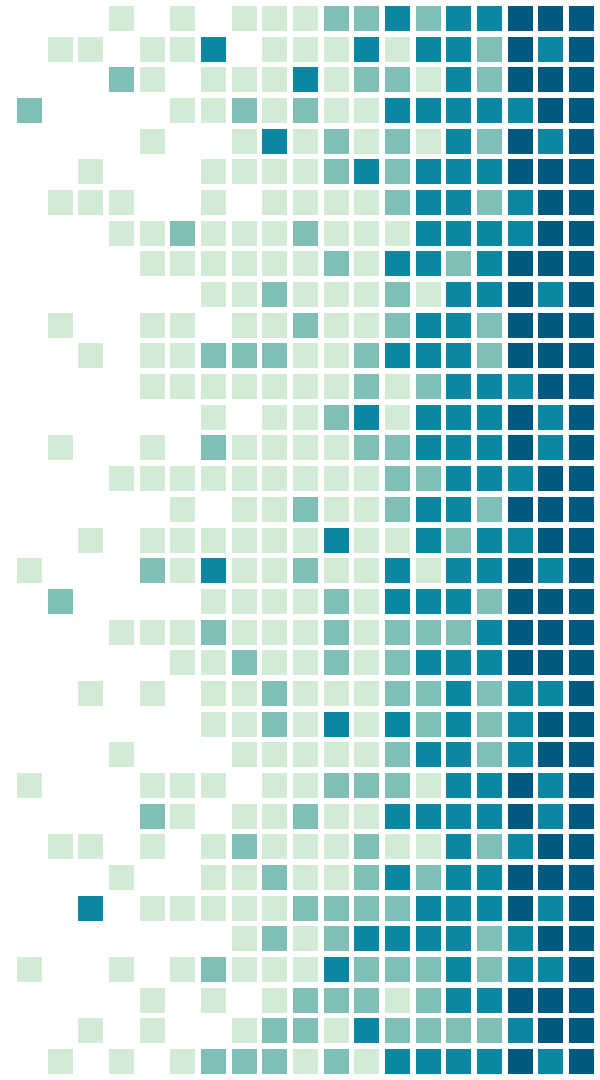
Dissecting Contract Addresses



2.

Smart Contracts

Programmable Digital Assets



A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs. -Szabo '94



Smart Contracts Overview

Code written in Solidity or other lang. and compiled

Stored on blockchain as Contract Account with EVM bytecode

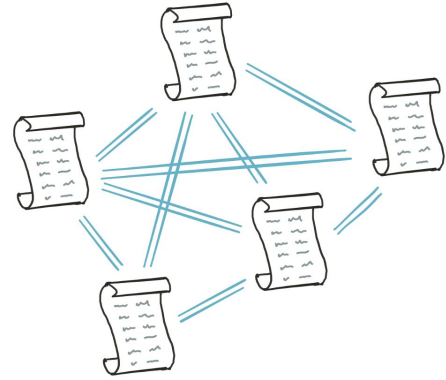
Functions called by sending transactions with or without ETH

- Gas costs must be paid to execute even if no ETH is being sent to contract

Contracts can call functions and create instances of other contracts

Functions are computed and verified by miners

An ERC20 token is a single instance of a smart contract



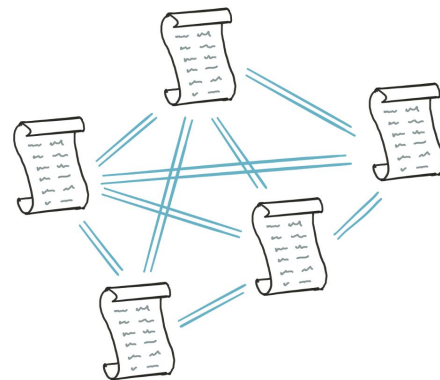
Contract State and Networks

Several networks but only one “mainnet”

Test Networks: Ropsten, Kovan and Rinkeby

Local Testing: testrpc, remix

A deployed contract = instance with 1 network and 1 state



A single scroll icon representing a contract state. It contains a table with two columns: 'Account' and 'Balance'.

Account	Balance
A	3
B	25
C	4
D	8
E	15
F	16
G	23
H	42
Ξ	10

A scroll icon representing a contract state with conditional annotations. It contains a table with two columns: 'Account' and 'Balance'.

Account	Balance
A	3
B	25 *
C	4
D	8
E	15 *
F	16
G	23 *
H	42
Ξ	10

Annotations:

- If this, then that (pointing to B)
- If this, then that (pointing to E)
- If this, then that (pointing to G)

A scroll icon representing a contract state with multiple conditional annotations. It contains a table with two columns: 'Account' and 'Balance'.

Account	Balance
A	3
B	25 *
C	4
D	8
E	15 *
F	16
G	23 *
H	42
Ξ	10

Annotations:

- If this, then that (pointing to B)
- If this, then that (pointing to E)
- If this, then that (pointing to G)

3. Development Environment

Truffle, Truffle Develop & Ganache CLI



Truffle

Installation and First Project

Truffle comes with:

- Solidity compiler
- Test harness
- Development blockchain
- Module (box) loader

Install and load the metacoin project



```
npm i -g truffle
...
mkdir metacoin
cd metacoin
truffle unbox metacoin
truffle test
```

Test Output

```
Compiling your contracts...
```

```
=====
```

```
> Compiling ./contracts/ConvertLib.sol  
> Compiling ./contracts/MetaCoin.sol  
> Compiling ./contracts/Migrations.sol  
> Compiling ./test/TestMetacoin.sol
```

```
TestMetacoin
```

```
✓ testInitialBalanceUsingDeployedContract (193ms)
```

```
✓ testInitialBalanceWithNewMetaCoin (123ms)
```

```
Contract: MetaCoin
```

```
✓ should put 10000 MetaCoin in the first account (46ms)
```

```
✓ should call a function that depends on a linked library (71ms)
```

```
✓ should send coin correctly (173ms)
```

```
5 passing (8s)
```

Truffle Develop

Truffle Develop and Migrations

Inside a “test” blockchain

Similar to an ssh/psql

Useful commands

- test
- compile
- migrate
- migrate --reset

More options (exit first)

- truffle --help



```
truffle develop
...
Truffle Develop started at http://127.0.0.1:9545/
...
truffle(develop)> test
...
truffle(develop)> migrate
...
truffle(develop)> migrate --reset
```

Ganache CLI

Another “test” blockchain?

Ganache ~= local testnet

Like a local webserver

Start in a screen / other terminal

On top of truffle-develop

- Local / wide network access
- More granular settings e.g. block times

More options (exit first)

- `ganache-cli --help`



```
ganache-cli
```

```
...  
Listening on 127.0.0.1:8545
```


Deploying to Ganache

Truffle has a nice config file

- truffle-config.js


ganache-cli is **not** running on port **7545**

We need to set that port to **8545**

Now from another screen / console

- truffle test

Go watch the magic in ganache-cli



```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*"
    },
    test: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*"
    }
  }
};
```

Output

Contract Creation

Constructor Call

- Owner balance

Function Call

- Send coin

```
eth_blockNumber  
eth_sendTransaction
```

```
Transaction: 0xad40d09c7c110a0fba4b0388fa7d7b3645528ab59a98bcce8b2df4042beb3c8a  
Contract created: 0xbb1023400043e6c68c6dff3c7ee0bc6564203138  
Gas usage: 338285  
Block Number: 4  
Block Time: Mon Aug 26 2019 08:25:00 GMT-0700 (Pacific Daylight Time)
```

```
eth_getTransactionReceipt
```

```
eth_sendTransaction
```

```
Transaction: 0xb1a5eed47a8289fa77089e801115dfc81696d189860fe9c125b611ed0e4fdf0d  
Gas usage: 27023  
Block Number: 5  
Block Time: Mon Aug 26 2019 08:25:00 GMT-0700 (Pacific Daylight Time)
```

```
eth_getTransactionReceipt
```

```
eth_sendTransaction
```

```
Transaction: 0xdbd0a5c1155050dbf0fe65ea9b534f463f0de95ac123a74c9da5201355e816fe  
Gas usage: 51008  
Block Number: 6  
Block Time: Mon Aug 26 2019 08:25:00 GMT-0700 (Pacific Daylight Time)
```

```
eth_getTransactionReceipt
```

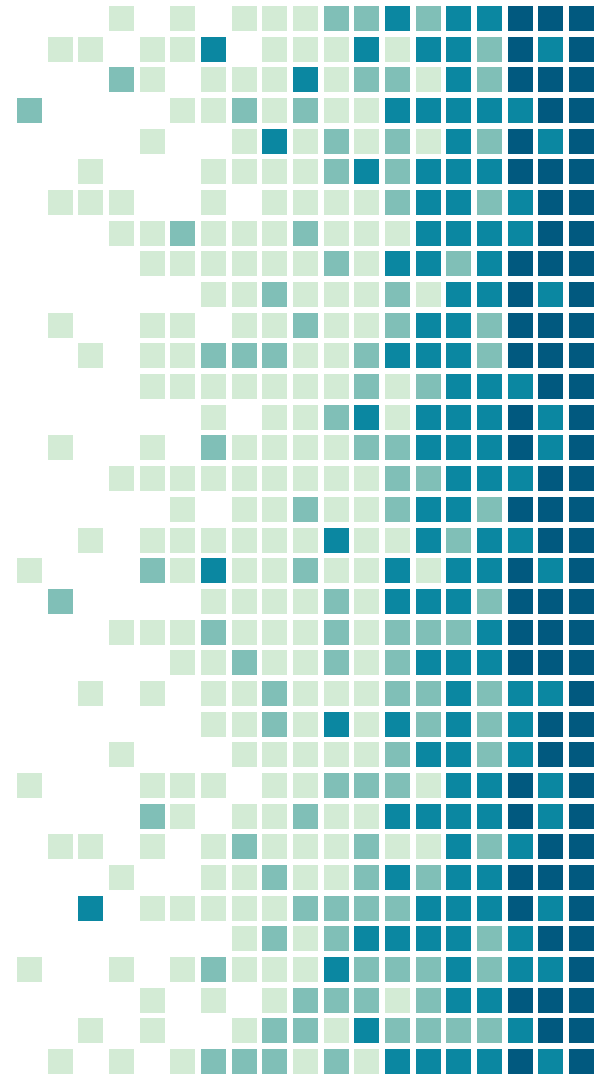
Part 2 Overview

1. Writing Tests
2. Deploying Contracts
3. Interacting with Contracts

1.

Writing Tests

JavaScript Testing in Depth



Setting Up Tests

```
const MetaCoin = artifacts.require("MetaCoin");

contract('MetaCoin', (accounts) => {
  it('should put 10000 MetaCoin in the first account', async () => {
    const metaCoinInstance = await MetaCoin.deployed();
    const balance = await metaCoinInstance.getBalance.call(accounts[0]);
    assert.equal(balance.valueOf(), 10000, "10000 wasn't in the first account");
  });

  ...

});
```

Contract Instances

```
const MetaCoin = artifacts.require("MetaCoin");
```

Truffle creates an “artifact” that stores the ABI and a lot of other information

When Truffle deploys, an “artifacts” JavaScript object is created (interface)

```
const metaCoinInstance = await MetaCoin.deployed();
```

Now we need to connect to the actual deployed contract instance

This instance is what we will use to sign and send our transactions

The Test Harness

```
contract('MetaCoin', (accounts) => {...});
```

Start a contract test with a callback (receives accounts)

Truffle Develop / Ganache would provide 10 unlocked accounts

```
it('should put 10000 MetaCoin in the first account', async () => {...});
```

Start a test with a descriptive title

The test function should return a promise

Truffle uses Mocha as a test runner and Chai for assertions

The MetaCoin Test

```
it('should put 10000 MetaCoin in the first account', async () => {  
  const metaCoinInstance = await MetaCoin.deployed();  
  const balance = await metaCoinInstance.getBalance.call(accounts[0]);  
  assert.equal(balance.valueOf(), 10000, "10000 wasn't in the first account");  
});
```

Exceptions?

```
let tx;  
try {  
  tx = await contract.iAmNotAllowed({ from: randomAcct });  
} catch (e) { ... }  
assert(tx === undefined, 'transaction occurred when it should NOT have');
```


Big Numbers (bn.js)

```
const oneEther = new web3.utils.BN(web3.utils.toWei('1', 'ether'));  
//100000000000000000000  
assert(oneEther.eq(oneEther), "1 ether != 1 ether");  
assert.equal(oneEther, 100000000000000000000, "1 ether != 1 ether");  
assert.equal(oneEther.toString(), '100000000000000000000', "1 ether != 1 ether");
```

Web3.utils has a number of useful numerical operators

JavaScript has some support for large integers, but...

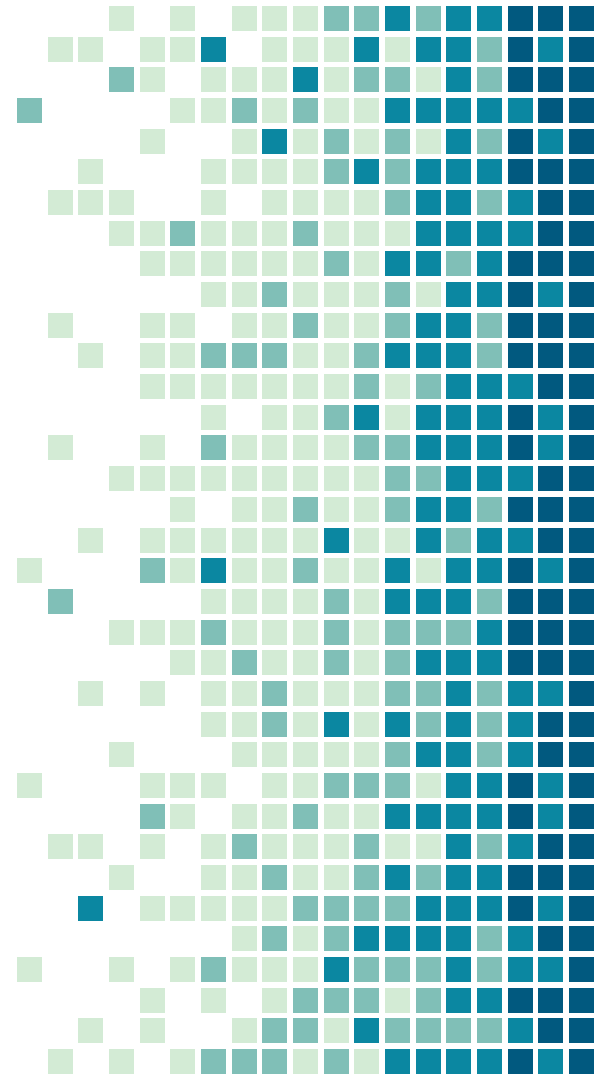
- use bn.js for testing equality (no headaches)
- e.g. `new web3.utils.BN(...)`

<https://github.com/indutny/bn.js/>

2.

Deploying Contracts

Network Configurations, Relayers



What Network am I on?

For sanity and while developing your frontend, stay local

```
'http://localhost:8545' //ganache-cli (testrpc)
```

```
'http://localhost:9545' //truffle develop
```

Connecting MetaMask to Truffle

- Use the mnemonic to connect MetaMask with Truffle develop

- `candy maple cake sugar pudding cream honey rich smooth crumble sweet treat`

When connecting to a real Network with MetaMask

- web3 and the provider will be injected into the page
- *** In some JS frameworks, race conditions with web3 can occur

3.

Connecting to Contracts

Introduction to Web3.js, ethers.js, ???



Truffle Develop

Connecting Directly

```
migrate
```

```
...
```

```
let instance = await MetaCoin.deployed()
```

```
let accounts = await web3.eth.getAccounts()
```

```
let ether = await instance.getBalanceInEth(accounts[0])
```

```
ether.toNumber()
```

Interacting with the contract using JavaScript

TruffleContract.js (used here) is not a typical production library

Connecting with Web3.js

Provide Provider, ABI and Deployed Address

```
const web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
const artifact = await fetch('./../build/contracts/MetaCoin.json').then((res) =>
res.json())
console.log(artifact)
const deployedAddress = artifact.networks[xxxxx].address
const instance = new web3.eth.Contract(artifact.abi, deployedAddress)
console.log(instance)
let accounts = await web3.eth.getAccounts()
console.log(accounts)
qs('#accounts').innerHTML = accounts.map((account) => `

${account}</div>`).join('')
let ether = await instance.methods.getBalanceInEth(accounts[0])
console.log(ether.toNumber())


```


Creating the Frontend

A Sample App Page

```
<body>
  <h1>Accounts</h1>
  <div id="accounts"></div>
  <h1>MetaCoin Balance in Ether</h1>
  <div id="balance"></div>

  <script
src="https://cdnjs.cloudflare.com/ajax/libs/babel-polyfill/7.4.4/polyfill.min.js"></script>
  <!-- https://raw.githubusercontent.com/ethereum/web3.js/1.x/dist/web3.min.js -->
  <script src="./web3.min.js"></script>
  <script src="./app.js"></script>

</body>
</html>
```

App.js

```
const qs = (sel) => document.querySelector(sel)

async function init() {
  const web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
  const artifact = await fetch('./../build/contracts/MetaCoin.json')
    .then((res) => res.json())
  const deployedAddress = artifact.networks[xxxxxx].address
  const instance = new web3.eth.Contract(artifact.abi, deployedAddress)
  let accounts = await web3.eth.getAccounts()
  qs('#accounts').innerHTML = accounts.map((account) => `<div>${account}</div>`).join('')
  const alice = accounts[0]
  let ether = await instance.methods.getBalanceInEth(alice)
    .call({from: alice, gas: 100000})
  qs('#balance').innerHTML = `<div>${ether}</div>`
}

window.onload = init
```

Integrating with Frameworks

JS Frameworks + Web3

Highly asynchronous calls to blockchain can be tricky

Make sure you understand the lifecycle of your components

- <https://reactjs.org/docs/react-component.html>
- <https://vuejs.org/v2/guide/instance.html>
- ...

Personal recommendations / opinions

- Import web3 helpers and libraries in html / app root
- Connect to contract ASAP store instance at root
- Interact + sync with state via Redux / Single Store pattern

Other Frameworks For Web3

Other Libraries and Frameworks for Web3

Frameworks

- Embark, Truffle

Web3 Libraries

- ethers.js
- ethjs
- Web3j (java / android)

Editors

- Remix Online Editor

Libraries

- Open Zeppelin, ...

Challenges