**Deptt. Of Computer Science,**

**NIT Srinagar**

**Blockchain Development Internship Report**

Prepared by:

**Irfan Mehraj Wani**

**irfanwani347@gmail.com**

**+916005377974**

**August 05 , 2022**

**Chapter 0: Internship Description**

**Chapter 1: Blockchain**

**1.1: Introduction**

**1.2: How a Blockchain works**

**1.2.1: Understanding Hash and Hashing Functions**

**1.2.2: What is a Block?**

**1.2.3: Blockchain**

**1.2.4: Distributed Blockchain**

**1.2.5: Private key and Public key**

**1.3: Concensus**

**1.3.1: Proof of Work**

**1.3.2: Proof of stake**

**Chapter 2: Smart Contracts and Solidity**

**2.1: Introduction**

**2.2: Solidity**

**2.2.1: What is Solidity?**

**2.2.2: Creating a smart contract in Solidity**

**2.2.3: Deploying and Interacting with a contract**

**Internship Description**

This Internship was intented to teach me the basics of blockchain development. In this internship, i learned about blockchain, how it works and different components which make it to work and blockchain development (i.e, decentralized applications over a blockchain). I learned about different techlologies used to build an application over a blockchain. Some of them are;

Blockchain (what is it and how it works),

Ethereum blockchain, What it is and how to interact with it,

MetaMask,

smart contracts,

Solidity (A programming language to create smart contracts),

Deploying a smart contract and interacting with it,

setting a local blockchain using Ganache (ganache-cli),

Chainlink (providing data to the blockchain in a decentralized manner)

And so on...

**Chapter 1:** **Blockchain**

**1.1: Introduction**

A blockchain is a distributed database or ledger that is shared among the nodes of a computer network. As a database, a blockchain stores information electronically in digital format.

Bitcoin was the first protocol to use this technology. This network is powered by cryptography. Blockchain brings in the concept of decentralisation where there is no intermediatory and there occur only peer to peer transactions in this system.

E.g, Ethereum uses blockchain to create decentralised applications, organisations etc.

Blockchain is a type of shared database that differs from a typical database in the way that it stores information; blockchains store data in blocks that are then linked together via cryptography.

As new data comes in, it is entered into a fresh block. Once the block is filled with data, it is chained onto the previous block, which makes the data chained together in chronological order.

Different types of information can be stored on a blockchain, but the most common use so far has been as a ledger for transactions.

In Bitcoin’s case, blockchain is used in a decentralized way so that no single person or group has control—rather, all users collectively retain control.

Decentralized blockchains are immutable, which means that the data entered is irreversible. For Bitcoin, this means that transactions are permanently recorded and viewable to anyone.

**1.2: How a Blockchain Works?**

The goal of blockchain is to allow digital information to be recorded and distributed, but not edited. In this way, a blockchain is the foundation for immutable ledgers, or records of transactions that cannot be altered, deleted, or destroyed. This is why blockchains are also known as a distributed ledger technology (DLT).

First proposed as a research project in 1991, the blockchain concept predated its first widespread application in use: Bitcoin, in 2009. In the years since, the use of blockchains has exploded via the creation of various cryptocurrencies, decentralized finance (DeFi) applications, non-fungible tokens (NFTs), and smart contracts.

**1.2.1: Understanding Hash and Hashing Functions**

Hash Function can be defined as a function which takes an input (some data) and applies some algorithm to produce an output of fixed length and the output is known as hash.

They are pseudo-random functions because, though they produce random outputs after we provide an input but the output always remains same with same input provided.

There are many examples of hash functions used these days in cryptography like, SHA256, SHA1, SHA2, Keccak256 (used by Ehtereum).

SHA256 is one of the most used hash functions these days. It takes an input of any length and produces an output of size 256 bits. In this

function, output has a size of 256 bits, i.e, it has 256 poisitions to be filled with either 0 or 1.

Thus total number of possible hashes which can be produced using this function is 2256 , thus making it nearly impossible to crack a hash.

There are only two ways to break a hashing algorithm;

1) Reverse Engineering the hash: One can reverse engineer the hashing algorithm (i.e, take the output andd retrace the path to the input) and extract the input from the output, but till date, no one has done this and seems impossible.

2) Brute Force: Another way to break a hash is to try different inputs until we get the desired hash thus giving us the desired input. Though this method doesn’t seem impossible but is very close to that as we already saw that there are a lot of possible hashes which makes it very difficult to guess that many hashes and it requires a lot of computational power.

Thus above two points justify the fact that hash functions are nearly impossible to crack making them secure, which is the reason they are used for managing passwords in databases and also in blockchain to validate a block.

E.g,

“superstrongpassword” if provided as an input string to SHA256 algorithm will produce a result as shown below;

9ca80cbce084b692e0b78adf4dc3f6b8883221d1fd59c86162c6f97bf4b75b8e

**1.2.2: What is a Block?**

Blocks are data structures within the blockchain database, where transaction data in a cryptocurrency blockchain are permanently recorded. A block records some or all of the most recent transactions not yet validated by the network. Once the data are validated, the block is closed. Then, a new block is created for new transactions to be entered into and validated.

A block is thus a permanent store of records that, once written, cannot be altered or removed.

A block is a place in a blockchain where information is stored and encrypted.

Blocks are identified by long numbers that include encrypted transaction information from previous blocks and new transaction information.

Blocks and the information within them must be verified by a network before new blocks can be created.

A block is represented by four parts;

1) Block Number: It is the number of the block in the blockchain.

2) Nonce: Stands for numbe used once, is a unique number assigned to a block which also represents the block. It acts as a solution to the block, as mining means finding a proper nonce which satisfies the condition for the hash (E.g, Hash should start with 4 zeroes).

3) Data: This represents the actual data to be stored in the block.

4) Hash: Hash is the output of the hash function where the first three (block number, nonce and data) collectively act as input.

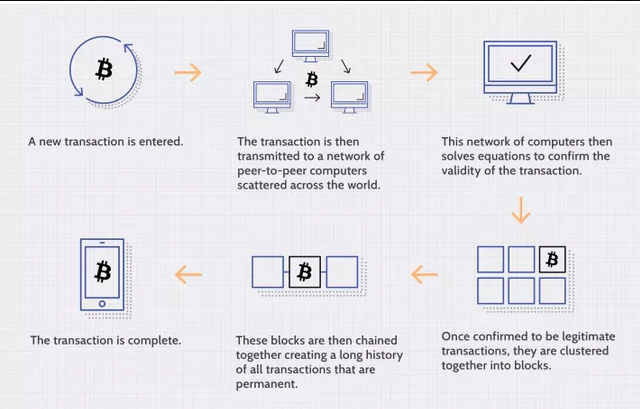
Besides these 4 components, when a block is in a blockchain, it also contains the hash of the previous block thus, getting connected to it, forming a chain, thus known as blockchain.

**1.2.3: Blockchain**

When multiple blocks are connected with each other where each block has the hash of previous block too, it is known as a blockchain.

First block in a block chain, which doesn’t have a previous hash, is known as Genesis Block.

Below is a diagram explaining the transaction process in a blockchain;



Simply put, a blockchain is a shared database or ledger. Pieces of data are stored in data structures known as blocks, and each node of the network has an exact replica of the entire database.

Security is ensured since if somebody tries to edit or delete an entry in one copy of the ledger, the majority will not reflect this change and it will be rejected.

**1.2.4: Distributed Blockchain**

When same blockchain is on different nodes in a network, it is known as Distributed Blockchain.

This reflects the decentralized nature of the blockchain as it is not controlled by a single node, but is present on every node in the network.

**1.2.5: Private key and Public key**

**Private Key**

A private key is a secret number that is used in cryptography and cryptocurrency.

A private key is a large, randomly-generated number with hundreds of digits. For simplicity, they are usually represented as strings of alphanumeric characters.

A cryptocurrency wallet consists of a set of public addresses and private keys. Anyone can deposit cryptocurrency in a public address, but funds cannot be removed from an address without the corresponding private key.

Private keys represent final control and ownership of cryptocurrency. It is vitally important to prevent one's private keys from being lost or compromised.

**Public Key**

A public key allows you to receive cryptocurrency transactions. It’s a cryptographic code that’s paired to a private key.

While anyone can send transactions to the public key, you need the private key to “unlock” them and prove that you are the owner of the cryptocurrency received in the transaction.

The public key that can receive transactions is usually an address, which is simply a shortened form of your public key.

Therefore, you can freely share your public key without worry.

Private key is used to generate public key using some algorithm.

Ethereum uses Elliptic Curve Digital Signature Algorithm (ECDSA) to generate public keys from private keys.

**1.3: Concensus**

A consensus mechanism refers to any number of methodologies used to achieve agreement, trust, and security across a decentralized computer network.

In the context of blockchains and cryptocurrencies, proof-of-work (PoW) and proof-of-stake (PoS) are two of the most prevalent consensus mechanisms.

Critics of Bitcoin miners have argued that PoW is overly energy-intensive, which has sparked the creation of new and more efficient mechanisms.

**1.3.1: Proof of Work**

Proof of work (PoW) is a decentralized consensus mechanism that requires members of a network to expend effort solving an arbitrary mathematical puzzle to prevent anybody from gaming the system.

Proof of work is used widely in cryptocurrency mining, for validating transactions and mining new tokens.

Due to proof of work, Bitcoin and other cryptocurrency transactions can be processed peer-to-peer in a secure manner without the need for a trusted third party.

Proof of work at scale requires huge amounts of energy, which only increases as more miners join the network.

The way that users detect tampering in practice is through hashes, long strings of numbers that serve as proof of work. Put a given set of data through a hash function (bitcoin uses SHA-256, Ethereum uses Keccak-256), and it will only ever generate one hash.

Due to the "avalanche effect," however, even a tiny change to any portion of the original data will result in a totally unrecognizable hash. Whatever the size of the original data set, the hash generated by a given function will be the same length. The hash is a one-way function: it cannot be used to obtain the original data.

Generating just any hash for a set of bitcoin transactions would be trivial for a modern computer, so in order to turn the process into "work," the bitcoin network sets a certain level of "difficulty."

This setting is adjusted so that a new block is "mined", added to the blockchain by generating a valid hash, approximately every 10 minutes.

Setting difficulty is accomplished by establishing a ‘target’ for the hash: the lower the target, the smaller the set of valid hashes, and the

harder it is to generate one. In practice, this means a hash that starts with a very long string of zeros.

Since a given set of data can only generate one hash, how do miners make sure they generate a hash below the target? They alter the input by adding an integer, called a nonce("number used once"). Once a valid hash is found, it is broadcast to the network, and the block is added to the blockchain.

Proof of work makes it extremely difficult to alter any aspect of the blockchain, since such an alteration would require re-mining all subsequent blocks.

It also makes it difficult for a user or pool of users to monopolize the network's computing power, since the machinery and power required to complete the hash functions are expensive.

**1.3.2: Proof of Stake**

With proof-of-stake (POS), cryptocurrency owners validate block transactions based on the number of coins a validator stakes.

Proof-of-stake (POS) was created as an alternative to Proof-of-work (POW), the original consensus mechanism used to validate a blockchain and add new blocks.

While PoW mechanisms require miners to solve cryptographic puzzles, PoS mechanisms require validators to simply hold and stake tokens.

Proof-of-stake (POS) is seen as less risky in terms of the potential for an attack on the network, as it structures compensation in a way that makes an attack less advantageous.

The next block writer on the blockchain is selected at random, with higher odds being assigned to nodes with larger stake positions.

Proof-of-stake reduces the amount of computational work needed to verify blocks and transactions that keep the blockchain, and thus a cryptocurrency, secure. Proof-of-stake changes the way blocks are verified using the machines of coin owners.

The owners offer their coins as collateral for the chance to validate blocks. Coin owners with staked coins become "validators.

Validators are then selected randomly to "mine," or validate the block. This system randomizes who gets to "mine" rather than using a competition-based mechanism like proof-of-work.

To become a validator, a coin owner must "stake" a specific amount of coins. For instance, Ethereum will require 32 ETH to be staked before a user can become a validator. Blocks are validated by more than one validator, and when a specific number of the validators verify that the block is accurate, it is finalized and closed.

**Chapter 2: Smart Contracts and Solidity**

**2.1: Introduction**

**Smart contracts** are simply programs stored on a blockchain that run when predetermined conditions are met. They typically are used to automate the execution of an agreement so that all participants can be immediately certain of the outcome, without any intermediary’s involvement or time loss. They can also automate a workflow, triggering the next action when conditions are met.

**Solidity** is an object-oriented, high-level language for implementing smart contracts. Solidity is a curly-bracket language desiged to target the Ehtereum Virtual Machine (EVM). It is influenced by C++, Python and javascript.

**2.2: Solidity**

**2.2.1: What is Solidity?**

Solidity is an object-oriented programming language used to write smart contracts. Syntactically, it resembles javascript (mostly) and C++.

**2.2.2: Creating smart contracts in solidity**

Here is an example of a simple smart contract which alters the value of a variable which stores a number;

// SPDX-License-Identifier: MIT

pragma solidity >= 0.7.0 < 0.9.0;

contract AlterNumber {

uint256 number;

function setNumber(uint256 \_number) public {

number = \_number;

}

function showNumber() public view returns(uint256) {

return number;

}

}

Lets understand the above example step-by-step.

The first line tells us that the source code is licensed under MIT (i.e, the code is open source and can be used by others).

Second line tells about the comaptible versions of solidity which can be used to run this smart contract (which is defined after it).

In the third line, we define a smart contract using the keyword ‘contract’ and named the contract as ‘AlterNumber’, then, inside the curly braces, we wrote the logic of the contract.

The syntax of the contract resembles the syntax of class in javascript.

Inside the contract, the first line defines a variable ‘number’ with type as uint256 (i.e, unsigned integer with size 256 bits). The syntax resembles that of C++ (or C).

We need to use semi-colon (;) to terminate a line of code.

After this line, we defined a function. To define a function, we use ‘function’ keyword followed by the name of the function, followed by parentheses containing the arguments with their respective types.

Parentheses are followed by the type of function, i.e, whether the function is public , private or any other type (like internal).

Some of the types are defined below;

**external** − External functions are meant to be called by other contracts. They cannot be used for internal call. To call external function within contract this.function\_name() call is required. State variables cannot be marked as external.

**Public** − Public functions/ Variables can be used both externally and internally. For public state variable, Solidity automatically creates a getter function.

**Internal** − Internal functions/ Variables can only be used internally or by derived contracts.

**Private** − Private functions/ Variables can only be used internally and not even by derived contracts.

Inside the first function, we altered the value of ‘number’ variable and set it to the argument passed to the function .

After the first function, we defined another function ‘showNumber’ which is a view function, means it cannot alter any value but can only return a value.

For a function to return something, we need to provide the type of the data returned after the ‘returns’ keyword (e.g, returns (uint256) means function will return a variable of type uint256).

Then inside the function, we simply return the value as;

return variable\_name;

In general, a contract has following components;

**Constructor:** A special function declared with constructor keyword which will be executed once per contract and is invoked when a contract is created.

Syntax of constructor will look like this;

contract contractName {

contstructor() public {

...

}

}

**State variables:** Variables per Contract to store the state of the contract.

Syntax will look like this;

contract contractName {

data\_type variable\_name = Variable\_value;

}

**Functions:** Functions per Contract which can modify the state variables to alter the state of a contract.

Syntax will look like this;

contract contractName {

function function\_name(arg1\_type arg1, arg2\_type arg2, ..., argn\_type argn) function\_type returns (return\_type) {

...

}

}

Like other programming languages, we can do a lot in solidity (which was discussed during the internship programme).

**2.2.3: Deploying and interacting with a contract**

You need to deploy your smart contract for it to be available to users of an Ethereum network.

To deploy a smart contract, you merely send an Ethereum transaction containing the compiled code of the smart contract without specifying any recipient.

To deploy a contract, we need following things;

1) Byte code of our contract , which is generated after compiling our contract.

2) ETH for gas – we'll set our gas limit like other transactions. Note that contract deployment needs a lot more gas than a simple ETH transfer.

3) A deployment script or plugin.

4) access to an Ethereum node, either by running your own, connecting to a public node, or via an API key using a node service like Infura or Alchemy.

For development purposes, we also use local blockchains, like Ganache to test our contracts.

Now , we know what we need to deploy a contract to an Ethereum network. But How do we do that?

One way to do that is using python, where we use web3py to deploy our contracts and interact with them.

Web3py is a python package build to deploy and interact with smart contracts.

A basic example will look like this;

import os

from solcx import compile\_standard, install\_solc

import json

from web3 import Web3

from dotenv import load\_dotenv

load\_dotenv()

with open("./SimpleStorage.sol", "r") as file:

simple\_storage\_file = file.read()

# install\_solc("0.8.15") # uncomment this line to install solidity version 0.8.1

5.

compiled\_sol = compile\_standard(

{

"language": "Solidity",

"sources": {"SimpleStorage.sol": {"content": simple\_storage\_file}},

"settings": {

"outputSelection": {

"\*": {"\*": ["abi", "metadata", "evm.bytecode", "evm.sourceMap"]}

}

},

},

solc\_version="0.8.15",

)

with open("compiled\_code.json", "w") as file:

json.dump(compiled\_sol, file)

# To deploy the cotract, we need to get the bytecode and abi of the contract

# get bytecode

bytecode = compiled\_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["evm"

][

"bytecode"

]["object"]

# get abi

abi = compiled\_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["abi"]

# Ganache is a fake (simulated) blockchain which we are going to use for our project deployment.

# connecting to rinkeby

w3 = Web3(

Web3.HTTPProvider("https://rinkeby.infura.io/v3/9cc547604894475986c383c22414

1c42")

)

chain\_id = 4

my\_address = "address\_goes\_here"

private\_key = os.getenv("private\_key")

# create the contract in python

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

# now we need to build a transaction, then sign it and send it

# get latest trnsaction

nonce = w3.eth.getTransactionCount(my\_address)

# 1. build a transaction

# 2. sign the transaction

# 3. send the transaction

transaction = SimpleStorage.constructor().buildTransaction(

{

"gasPrice": w3.eth.gas\_price,

"chainId": chain\_id,

"from": my\_address,

"nonce": nonce,

}

)

signed\_transaction = w3.eth.account.sign\_transaction(

transaction, private\_key=private\_key

)

# send the signed transaction

tx\_hash = w3.eth.send\_raw\_transaction(signed\_transaction.rawTransaction)

# waiting till the transaction goes through

tx\_reciept = w3.eth.wait\_for\_transaction\_receipt(tx\_hash)

# working with the contract (i.e, interacting with it). For that we need :

# contract address

# contract ABI

simple\_storage = w3.eth.contract(address=tx\_reciept.contractAddress, abi=abi)

# Two ways to interact with a transaction, call(simulate making a call and getti

ng a return value i.e, they dont make any state changes) and transact (actually

makes a state change)

print(

simple\_storage.functions.retrieve().call()

) # will print the initial value of the number used in the simplestorage contract

print("creating contract...")

store\_transaction = simple\_storage.functions.store(15).buildTransaction(

{

"gasPrice": w3.eth.gas\_price,

"chainId": chain\_id,

"from": my\_address,

"nonce": nonce + 1,

}

)

print("signing contract...")

signed\_store\_tx = w3.eth.account.sign\_transaction(

store\_transaction, private\_key=private\_key

)

print("sending the contract...")

send\_store\_tx = w3.eth.send\_raw\_transaction(signed\_store\_tx.rawTransaction)

print("waiting for the contract to complete...")

tx\_reciept = w3.eth.wait\_for\_transaction\_receipt(send\_store\_tx)

print("Deployed!!!")

Different steps are explained using comments in the above code.

We also have another package in python known as brownie, built on web3py which is much easier to use than web3py.

A simple example of brownie is given below;

# we can import the created contracts directly from brownie as shown below (SimpleStorage is an example contract created here)

from brownie import accounts, SimpleStorage

def deploy\_simple\_storage():

account = accounts[0]

# brownie spins up 10 accounts for us and we are using the first one here

simple\_storage = SimpleStorage.deploy({"from": account})

stored\_value = simple\_storage.retrieve()

print(stored\_value)

transaction = simple\_storage.store(15, {"from": account})

transaction.wait(1)

new\_val = simple\_storage.retrieve()

print(new\_val)

def main():

deploy\_simple\_storage()

In Nutshell, we learned about the decentralized systems, how they work and how we can create apps over them. Thus, the only difference between centralized systems and decentralized systems is how they are managed. Centralized Systems are managed by a centralized authority and decentralized systems are controlled by everyone in the network, secured by hashing algorithms.

Thus, we can do a lot with this technology like, create decentralized apps, decentralized Fincance (DeFi) etc where everthing will be controlled by everyone.

Here, I conclude my Internship report, Thanking my Trainer **Dr. Sparsh Sharma** for helping me thoughout this journey.

Thank You