**Cryptocurrency**

**1.Write a program to create the chain with Genesis block and adding block into**

**blockchain and validating the chain for any alteration. (Part-I)**

**Code:-**

const SHA256=require('crypto-js/sha256');

class Block

{

constructor(index,timestamp,data,previousHash='')

{

this.index=index;

this.timestamp=timestamp;

this.data=data;

this.previousHash=previousHash;

this.hash=this.calculateHash();

}

calculateHash()

{

return SHA256(this.index+this.previousHash+this.timestamp+JSON.stringify(this.data)+this.nonce).toString();

}

}

class Blockchain

{

constructor()

{

this.chain=[this.createGenesisBlock()];

}

createGenesisBlock()

{

return new Block(0,"02/11/2021","Genesis Block","0");

}

getLatestBlock()

{

return this.chain[this.chain.length-1];

}

addBlock(newBlock)

{

newBlock.previousHash=this.getLatestBlock().hash;

newBlock.hash=newBlock.calculateHash();

this.chain.push(newBlock);

}

isChainValid()

{

for(let i=1;i<this.chain.length;i++)

{

const currenytBlock = this.chain[i];

const previousBlock=this.chain[i-1];

if(currenytBlock.hash!==currenytBlock.calculateHash())

{

return false;

}

if(currenytBlock.previousHash !== previousBlock.hash)

{

return false;

}

}

return true;

}

}

let MyCoin =new Blockchain();

console.log("Adding Blocks...");

MyCoin.addBlock(new Block(1,"05/08/2022",{amount:4000}));

MyCoin.addBlock(new Block(2,"07/08/2022",{amount:1000}));

console.log(JSON.stringify(MyCoin,null,4));

console.log('Is blockchain valid?'+MyCoin.isChainValid());

**Output:-**



**2. Write a program to implementing proof of work for blockchain. (Part-II)**

**Code:**

//proof of work addeded

this.chain=[this.createGenesisBlock()];

this.difficulty=4;

}

createGenesisBlock()

{

return new Block(0,"02/11/2021","Genesis Block","0");

}

getLatestBlock()

{

return this.chain[this.chain.length-1];

}

addBlock(newBlock)

{

newBlock.previousHash=this.getLatestBlock().hash;

newBlock.mineBlock(this.difficulty);

this.chain.push(newBlock);

}

isChainValid()

{

for(let i=1;i<this.chain.length;i++)

{

const currenytBlock = this.chain[i];

const previousBlock=this.chain[i-1];

if(currenytBlock.hash!==currenytBlock.calculateHash())

{

return false;

}

if(currenytBlock.previousHash !== previousBlock.hash)

{

return false;

}

}

return true;

}

}

let MyCoin =new Blockchain();

console.log("Mining The Bloc1...");

MyCoin.addBlock(new Block(1,"05/08/2022",{amount:4000}));

console.log("Mining The Bloc2...");

MyCoin.addBlock(new Block(2,"07/08/2022",{amount:1000}));

console.log(JSON.stringify(MyCoin,null,4));

**Output:-**



**3. Write a program to add multiple transactions into block and give reward to**

**miner for successful mining of block in blockchain. (Part-III)**

**Code:**

//giving reward to miners and multiple transaction can be added

const SHA256=require('crypto-js/sha256');

class Transaction

{

constructor(fromAddress,toAddress,amount)

{

this.fromAddress=fromAddress;

this.toAddress=toAddress;

this.amount=amount;

}

}

class Block

{

constructor(timestamp,transactions,previousHash='')

{

this.timestamp=timestamp;

this.transactions=transactions;

this.previousHash=previousHash;

this.hash=this.calculateHash();

this.nonce=0;

}

calculateHash()

{

return SHA256(this.index+this.previousHash+this.timestamp+

JSON.stringify(this.data)+this.nonce).toString();

}

mineBlock(difficulty)

{

while(this.hash.substring(0,difficulty)!==Array(difficulty+1).join("0"))

{

this.nonce++;

this.hash=this.calculateHash();

}

console.log("Block Mined!"+this.hash);

}

}

class Blockchain

{

constructor()

{

this.chain=[this.createGenesisBlock()];

this.difficulty=4;

this.pendingTransaction=[];

this.miningReward=100;

}

createGenesisBlock()

{

return new Block(0,"02/11/2021","Genesis Block","0");

}

getLatestBlock()

{

return this.chain[this.chain.length-1];

}

minePendingTransaction(miningRewardAddress)

{

let block = new Block(Date.now(),this.pendingTransaction);

block.mineBlock(this.difficulty);

console.log("Block mined Successfully");

this.chain.push(block);

this.pendingTransaction=[new Transaction(null,miningRewardAddress,this.miningReward)];

}

createTransaction(transaction)

{

this.pendingTransaction.push(transaction);

}

getBalancedOfAddress(address)

{

let balance=0;

for(const block of this.chain)

{

for(const trans of block.transactions)

{

if(trans.fromAddress==address)

{

balance-=trans.amount;

}

if(trans.toAddress==address)

{

balance+=trans.amount;

}

}

}

return balance;

}

isChainValid()

{

for(let i=1;i<this.chain.length;i++)

{

const currenytBlock = this.chain[i];

const previousBlock=this.chain[i-1];

if(currenytBlock.hash!==currenytBlock.calculateHash())

{

return false;

}

if(currenytBlock.previousHash !== previousBlock.hash)

{

return false;

}

}

return true;

}

}

let MyCoin =new Blockchain();

MyCoin.createTransaction(new Transaction('address1','address2',200));

console.log("\n Starting the mining by miner....");

MyCoin.minePendingTransaction('Tata-address');

console.log("\n Balance of Tata-address is= "+MyCoin.getBalancedOfAddress('Tata-address'));

console.log(JSON.stringify(MyCoin,null,4));

**Output:**



**4. Write a program to sign the transaction with private key and verify the**

**signed transactions for blockchain. (Part-IV)**

**Code:**

Blockchain.js

const EC=require('elliptic').ec;

const ec=new EC('secp256k1');

const SHA256=require('crypto-js/sha256');

class Transaction

{

constructor(fromAddress,toAddress,amount)

{

this.fromAddress=fromAddress;

this.toAddress=toAddress;

this.amount=amount;

}

calculateHash()

{

return SHA256(this.fromAddress+this.toAddress+this.amount).toString();

}

signTransaction(signingKey)

{

if(signingKey.getPublic('hex')!==this.fromAddress)

{

throw new Error('You cannot sign transactions for other wallets');

}

const hashTx=this.calculateHash();

const sig=signingKey.sign(hashTx,'base64');

this.signature=sig.toDER('hex');

}

isValid()

{

if(this.fromAddress==null)return true;

if(!this.signature || this.signature.length==0)

{

throw new Error('No signature in this transaction!');

}

const publicKey=ec.keyFromPublic(this.fromAddress,'hex');

return publicKey.verify(this.calculateHash(),this.signature);

}

}

class Block

{

constructor(timestamp,transactions,previousHash='')

{

this.timestamp=timestamp;

this.transactions=transactions;

this.previousHash=previousHash;

this.hash=this.calculateHash();

this.nonce=0;

}

calculateHash()

{

return

SHA256(this.previousHash+this.timestamp+JSON.stringify(this.transactions)+this.nonce).toString();

}

mineBlock(difficulty)

{

while(this.hash.substring(0,difficulty)!==Array(difficulty+1).join("0"))

{

this.nonce++;

this.hash=this.calculateHash();

}

console.log("Block Mined!"+this.hash);

}

hasValidTransactions()

{

for(const tx of this.transactions)

{

if(!tx.isValid())

{

return false;

}

}

return true;

}

}

class Blockchain

{

constructor()

{

this.chain=[this.createGenesisBlock()];

this.difficulty=4;

this.pendingTransaction=[];

this.miningReward=100;

}

createGenesisBlock()

{

return new Block(0,"02/11/2021","Genesis Block","0");

}

getLatestBlock()

{

return this.chain[this.chain.length-1];

}

minePendingTransaction(miningRewardAddress)

{

const rewardTx=new

Transaction(null,miningRewardAddress,this.miningReward);

this.pendingTransaction.push(rewardTx);

let block = new

Block(Date.now(),this.pendingTransaction,this.getLatestBlock().hash);

block.mineBlock(this.difficulty);

console.log("Block mined Successfully");

this.chain.push(block);

this.pendingTransaction=[];

}

addTransaction(transaction)

{

if(!transaction.fromAddress||!transaction.toAddress)

{

throw new Error('Transaction must include from and to address! ');

}

if(!transaction.isValid())

{

throw new Error("Cannot add invalid transaction to chain!");

}

this.pendingTransaction.push(transaction);

}

getBalancedOfAddress(address)

{

let balance=0;

for(const block of this.chain)

{

for(const trans of block.transactions)

{

if(trans.fromAddress==address)

{

balance-=trans.amount;

}

if(trans.toAddress==address)

{

balance+=trans.amount;

}

}

}

return balance;

}

isChainValid()

{

for(let i=1;i<this.chain.length;i++)

{

const currenytBlock = this.chain[i];

const previousBlock=this.chain[i-1];

if(currenytBlock.hash!==currenytBlock.calculateHash())

{

return false;

}

if(currenytBlock.previousHash !== previousBlock.hash)

{

return false;

}

}

return true;

}

}

module.exports.Blockchain=Blockchain;

module.exports.Transaction=Transaction;

**Main.js:-**

const {Blockchain,Transaction}=require("./Blockchain");

const EC=require('elliptic').ec;

const ec=new EC('secp256k1');

const

myKey=ec.keyFromPrivate('046992ca6c22bf0bbeaf55eb86d663b60158690502016b5c6c1c5f7e0d52fa3a701155ce1de50b4014e532b95737f62be92f2164a593170102a4413e3189b5ef3d');

const myWalletAddress=myKey.getPublic('hex');

let MyCoin =new Blockchain();

const tx1=new Transaction(myWalletAddress,'public key goes here',10);

tx1.signTransaction(myKey);

MyCoin.addTransaction(tx1);

console.log("\n Starting the miner....");

MyCoin.minePendingTransaction(myWalletAddress);

console.log("\n Balance of myWalletAddresss is="+MyCoin.getBalancedOfAddress(myWalletAddress));

console.log("\n Is chain valid= "+MyCoin.isChainValid());

MyCoin.chain[1].transactions[0].amount=15;

console.log("\n Is chain valid= "+MyCoin.isChainValid());

**KeyGenerator.js:**

const EC=require('elliptic').ec;

const ec=new EC('secp256k1');

const key=ec.genKeyPair();

const publicKey=key.getPublic('hex');

const privateKey=key.getPrivate('hex');

console.log("\n Public Key",privateKey);

console.log("\n Private Key",publicKey);

**Output-**

