Lab:1 Write a Java program to Create and Record Transactions, Organise Transactions into Blocks, Securely Link Blocks Together, Mine New Blocks , and Validate the Blockchain.

```
import java.util.ArrayList;

import java.util.Date;

class Block {

   public String hash;

   public String previousHash;

   private String data; // For simplicity, data is a string in this example.

   private long timeStamp;

   private int nonce;

   // Constructor

   public Block(String data, String previousHash) {

      this.data = data;

      this.previousHash = previousHash;

      this.timeStamp = new Date().getTime();

      this.hash = calculateHash();

   }

   // Calculate hash based on block's data

   public String calculateHash() {

      String calculatedHash = StringUtil.applySha256(

            previousHash +

                  Long.toString(timeStamp) +

                  Integer.toString(nonce) +

                  data

      );

      return calculatedHash;
```

```java
    }
    // Mine the block to find a hash with leading zeros
    public void mineBlock(int difficulty) {
        String target = new String(new char[difficulty]).replace('\0', '0'); // Create a string with difficulty
* "0"
        while (!hash.substring(0, difficulty).equals(target)) {
            nonce++;
            hash = calculateHash();
        }
        System.out.println("Block Mined: " + hash);
    }
}
class Blockchain {
     static int difficulty = 5;
    ArrayList<Block> blockchain = new ArrayList<>();
    // Add block to the blockchain
    public void addBlock(Block newBlock) {
        newBlock.mineBlock(difficulty);
        blockchain.add(newBlock);
    }
    // Validate the blockchain
    public boolean isChainValid() {
        Block currentBlock;
        Block previousBlock;
        String hashTarget = new String(new char[difficulty]).replace('\0', '0');
        // Loop through blockchain to check hashes:
        for (int i = 1; i < blockchain.size(); i++) {
            currentBlock = blockchain.get(i);
            previousBlock = blockchain.get(i - 1);
            // Compare registered hash and calculated hash:
            if (!currentBlock.hash.equals(currentBlock.calculateHash())) {
```

```
                System.out.println("Current Hashes not equal");
                return false;
            }
            // Compare previous hash and registered previous hash:
            if (!previousBlock.hash.equals(currentBlock.previousHash)) {
                System.out.println("Previous Hashes not equal");
                return false;
            }
            // Check if hash is solved
            if (!currentBlock.hash.substring(0, difficulty).equals(hashTarget)) {
                System.out.println("This block hasn't been mined");
                return false;
            }
        }
        return true;
    }
}

public class Main {
    public static void main(String[] args) {
        Blockchain blockchain = new Blockchain();
        // Add some sample blocks to the blockchain
        blockchain.addBlock(new Block("First block", "0"));
        blockchain.addBlock(new Block("Second block",
blockchain.blockchain.get(blockchain.blockchain.size() - 1).hash));
        blockchain.addBlock(new Block("Third block",
blockchain.blockchain.get(blockchain.blockchain.size() - 1).hash));
        // Validate the blockchain
        System.out.println("Is blockchain valid? " + blockchain.isChainValid());
    }
}
class StringUtil {
```

```
    // Applies Sha256 to a string and returns the result.
    public static String applySha256(String input) {
        try {
            java.security.MessageDigest digest = java.security.MessageDigest.getInstance("SHA-256");
            // Applies sha256 to our input,
            byte[] hash = digest.digest(input.getBytes("UTF-8"));
            StringBuffer hexString = new StringBuffer(); // This will contain hash as hexidecimal
            for (int i = 0; i < hash.length; i++) {
                String hex = Integer.toHexString(0xff & hash[i]);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }
            return hexString.toString();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Output:

Block Mined: 00000c3a7a3a1a219d613469aa82659557c3507c53d16c2e7ac9aa341b5fa11a

Block Mined: 00000e78b81b232584fb25e44698ece74bd2c7206b3f02a4e785bafb0bd8bd32

Block Mined: 00000e0577cbdc30e34ce00caacd5c9df840a4ec58e2fee03f306cb58920bd1b

Is blockchain valid? True

Lab 2: Write a program that defines the roles of token holders, delegates, and also the consensus process. Make sure to simulate a few rounds of block creation.

```
import java.util.*;
// Token holder class
class TokenHolder {
    private int tokens;
    public TokenHolder(int tokens) {
```

```
        this.tokens = tokens;
    }
    public int getTokens() {
        return tokens;
    }
    public void setTokens(int tokens) {
        this.tokens = tokens;
    }
}
// Delegate class
class Delegate {
    private String name;
    private int votes;
    public Delegate(String name) {
        this.name = name;
        this.votes = 0;
    }
    public String getName() {
        return name;
    }
    public int getVotes() {
        return votes;
    }
    public void addVote() {
        votes++;
    }
}
// Blockchain class
class Blockchain {
    private List<Block> blockchain;
    private List<Delegate> delegates;
```

```
    private int round;
    public Blockchain() {
        this.blockchain = new ArrayList<>();
        this.delegates = new ArrayList<>();
        this.round = 1;
    }
    public void addDelegate(Delegate delegate) {
        delegates.add(delegate);
    }
    public void addTokenHolderVote(TokenHolder tokenHolder, Delegate delegate) {
        if (tokenHolder.getTokens() > 0) {
            delegate.addVote();
            tokenHolder.setTokens(tokenHolder.getTokens() - 1);
            System.out.println("Token holder voted for delegate: " + delegate.getName());
        } else {
            System.out.println("Token holder does not have enough tokens to vote.");
        }
    }
    public void createBlock(Delegate delegate) {
        if (delegate.getVotes() > 0) {
            System.out.println("Creating block for round " + round + " with delegate: " +
delegate.getName());
            blockchain.add(new Block(round, delegate.getName()));
            round++;
        } else {
            System.out.println("Delegate does not have enough votes to create a block.");
        }
    }
    public void printBlockchain() {
        System.out.println("\nBlockchain:");
        for (Block block : blockchain) {
```

```
            System.out.println(block);
        }
    }
}
// Block class
class Block {
    private int round;
    private String delegate;
    public Block(int round, String delegate) {
        this.round = round;
        this.delegate = delegate;
    }
    @Override
    public String toString() {
        return "Block{" +
                "round=" + round +
                ", delegate='" + delegate + '\'' +
                '}';
    }
}
public class Main {
    public static void main(String[] args) {
        Blockchain blockchain = new Blockchain();
        // Create delegates
        Delegate delegate1 = new Delegate("Delegate 1");
        Delegate delegate2 = new Delegate("Delegate 2");
        Delegate delegate3 = new Delegate("Delegate 3");
        // Add delegates to the blockchain
        blockchain.addDelegate(delegate1);
        blockchain.addDelegate(delegate2);
        blockchain.addDelegate(delegate3);
```

```
        // Create token holders
        TokenHolder tokenHolder1 = new TokenHolder(5);
        TokenHolder tokenHolder2 = new TokenHolder(3);
        // Token holders vote for delegates
        blockchain.addTokenHolderVote(tokenHolder1, delegate1);
        blockchain.addTokenHolderVote(tokenHolder1, delegate2);
        blockchain.addTokenHolderVote(tokenHolder2, delegate3);
        // Simulate block creation
        blockchain.createBlock(delegate1);
        blockchain.createBlock(delegate2);
        blockchain.createBlock(delegate3);
        // Print the blockchain
        blockchain.printBlockchain();
    }
}
```

```
Token holder voted for delegate: Delegate 1
Token holder voted for delegate: Delegate 2
Token holder voted for delegate: Delegate 3
Creating block for round 1 with delegate: Delegate 1
Creating block for round 2 with delegate: Delegate 2
Creating block for round 3 with delegate: Delegate 3
Blockchain:
Block{round=1, delegate='Delegate 1'}
Block{round=2, delegate='Delegate 2'}
Block{round=3, delegate='Delegate 3'}
```

Lab 3: Understand different consensus mechanisms in blockchain.

Steps: a. Research and present information on consensus mechanisms like Proof of Work (PoW), Proof of Stake (PoS), etc.

b. Discuss the advantages and disadvantages of each consensus mechanism.

c. Simulate a simple consensus scenario in a controlled environment.

a. Consensus Mechanisms:

Proof of Work (PoW):

Definition: PoW is a consensus algorithm in which miners compete to solve complex mathematical puzzles to validate and add blocks to the blockchain.

How it works: Miners use computational power to solve cryptographic puzzles, and the first one to solve it gets the right to add the next block. This process is energy-intensive.

Examples: Bitcoin, Ethereum (currently transitioning to PoS).

Proof of Stake (PoS):

Definition: PoS is a consensus algorithm where validators are chosen to create new blocks based on the number of coins they hold and are willing to "stake" as collateral.

How it works: Validators are selected to create new blocks based on their stake in the network. The probability of being chosen is proportional to the amount of cryptocurrency staked.

Examples: Ethereum 2.0 (after transition), Cardano, Algorand.

Delegated Proof of Stake (DPoS):

Definition: DPoS is a variation of PoS where coin holders vote for a limited number of delegates who are responsible for validating transactions and creating new blocks.

How it works: Coin holders vote for delegates who represent them in the consensus process. Delegates with the most votes become block producers.

Examples: EOS, Tron, BitShares.

Practical Byzantine Fault Tolerance (PBFT):

Definition: PBFT is a consensus algorithm designed for permissioned blockchains where participants are known and trusted.

How it works: Nodes in the network communicate with each other to agree on the validity of transactions. Consensus is reached through a series of rounds of message exchanges.

Examples: Hyperledger Fabric, Ripple.

b. Advantages and Disadvantages:

Proof of Work (PoW):

Advantages:

Proven security: It has been battle-tested and is highly secure.

Decentralization: Anyone with computational power can participate.

Disadvantages:

Energy-intensive: Requires massive amounts of computational power, leading to high energy consumption.

Centralization of mining: Mining pools can consolidate power, leading to centralization concerns.

Proof of Stake (PoS):

Advantages:

Energy-efficient: Doesn't require the same level of computational power as PoW.

More decentralized: Encourages widespread participation as anyone with coins can become a validator.

Disadvantages:

Potential for centralization: Wealthier participants have more influence, potentially leading to centralization.

Nothing-at-stake problem: Validators may have an incentive to support multiple forks.

Delegated Proof of Stake (DPoS):

Advantages:

Scalability: Faster block times and transaction throughput compared to PoW and PoS.

More energy-efficient: Doesn't require extensive computational resources.

Disadvantages:

Centralization risk: Relies on a limited number of delegates, which can lead to centralization if not properly managed.

Governance challenges: Voting mechanisms and delegate selection can be complex.

Practical Byzantine Fault Tolerance (PBFT):

Advantages:

Fast transactions: Can achieve high throughput and low latency.

Finality: Provides immediate finality once a decision is reached.

Disadvantages:

Permissioned: Suitable only for permissioned blockchains with known and trusted participants.

Limited scalability: Can face scalability challenges with a large number of participants.

c. Simple Consensus Scenario Simulation:

```
import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

import java.util.Random;

class Block {

    private int index;

    private long timestamp;

    private String data;
```

```java
private String previousHash;
private String hash;
private int nonce;
public Block(int index, String data, String previousHash) {
    this.index = index;
    this.timestamp = System.currentTimeMillis();
    this.data = data;
    this.previousHash = previousHash;
    this.nonce = 0;
    this.hash = calculateHash();
}
public String calculateHash() {
    String dataToHash = index + timestamp + data + previousHash + nonce;
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hashBytes = digest.digest(dataToHash.getBytes());
        StringBuilder hexString = new StringBuilder();
        for (byte hashByte : hashBytes) {
            String hex = Integer.toHexString(0xff & hashByte);
            if (hex.length() == 1)
                hexString.append('0');
            hexString.append(hex);
        }
        return hexString.toString();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
        return null;
    }
}
public void mineBlock(int difficulty) {
    String target = new String(new char[difficulty]).replace('\0', '0');
```

```
        while (!hash.substring(0, difficulty).equals(target)) {
            nonce++;
            hash = calculateHash();
        }
        System.out.println("Block mined: " + hash);
    }
    public int getIndex() {
        return index;
    }
    public String getHash() {
        return hash;
    }
    public String getPreviousHash() {
        return previousHash;
    }
    @Override
    public String toString() {
        return "Block{" +
                "index=" + index +
                ", timestamp=" + timestamp +
                ", data='" + data + '\'' +
                ", previousHash='" + previousHash + '\'' +
                ", hash='" + hash + '\'' +
                ", nonce=" + nonce +
                '}';
    }
}
public class Main {
    public static void main(String[] args) {
        int difficulty = 4;
        String genesisPreviousHash = "0";
```

```
        Block genesisBlock = new Block(0, "Genesis Block", genesisPreviousHash);
        genesisBlock.mineBlock(difficulty);
        String previousHash = genesisBlock.getHash();
        for (int i = 1; i < 5; i++) {
            Block block = new Block(i, "Block " + i, previousHash);
            block.mineBlock(difficulty);
            previousHash = block.getHash();
        }
    }
}
```

Output:

Block mined: 00004fb75caf54f41e1f7c8dc637b91bbb2e5a5e63083b02c0f1f08494c9974a

Block mined: 0000540be8246fc2a27fd6ab4d6a75c5c82c0f226524501bb35acd6c7bf952d1

Block mined: 0000a9c330377e6000cbecb6df06864848826086be962e437636226c8b2d1d24

Block mined: 00009ea48b9d7d65700ee6dd01ecdef4ff7254b4ca4bfa916560c842f67e8b8f

Block mined: 0000f94d13b6b7bc07dfa27dc86822c431c60ee71b859ccd46053a5d9bbdf919

Lab 5:

Understand the basic components of a blockchain network and set up a local blockchain using tools like Ganache.

Steps:

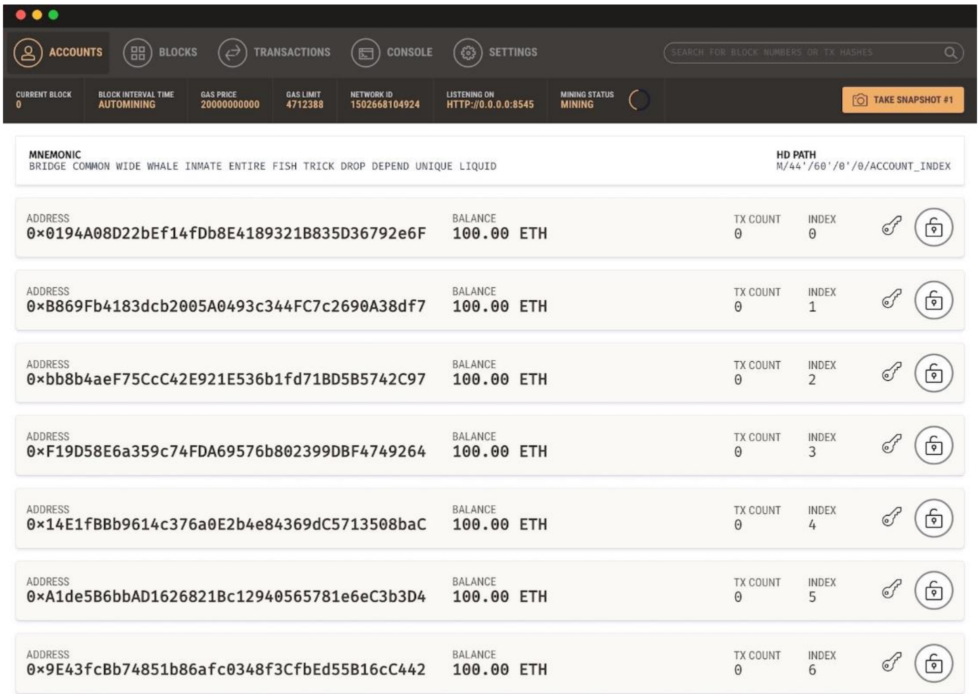
a. Install Ganache on your machine.

b. Create a new blockchain project.

c. Explore the blocks, transactions, and accounts in the local blockchain.

d. Perform simple transactions between accounts.

a. Install Ganache on your machine:



d. Perform simple transactions between accounts:

1. Install Truffle globally if you haven't already:

```
npm install -g truffle
```

2. Create a new directory for your Truffle project:

```
mkdir myBlockchainProject
cd myBlockchainProject
```

3. Initialize a new Truffle project:

```
truffle init
```

4. Create a JavaScript file to interact with the blockchain (e.g., app.js):

```
const Web3 = require('web3');

const web3 = new Web3('http://localhost:7545'); // Connect to Ganache's RPC server


async function performTransactions() {

  // Get accounts from Ganache

  const accounts = await web3.eth.getAccounts();

  console.log('Accounts:', accounts);
```

```
  // Get initial balances
  const initialBalances = await Promise.all(accounts.map(account =>
web3.eth.getBalance(account)));
  console.log('Initial Balances:', initialBalances);
  // Perform a transaction from account[0] to account[1]
  const amountToSend = web3.utils.toWei('0.1', 'ether');
  const txReceipt = await web3.eth.sendTransaction({
    from: accounts[0],
    to: accounts[1],
    value: amountToSend
  });
  console.log('Transaction Receipt:', txReceipt);
  // Get updated balances
  const updatedBalances = await Promise.all(accounts.map(account =>
web3.eth.getBalance(account)));
  console.log('Updated Balances:', updatedBalances);
}
performTransactions();
```

5. Run the script using Node.js:

```
node app.js
```

Lab 7: Understand the concept of wallets in blockchain and create and manage wallets using tools like MetaMask.

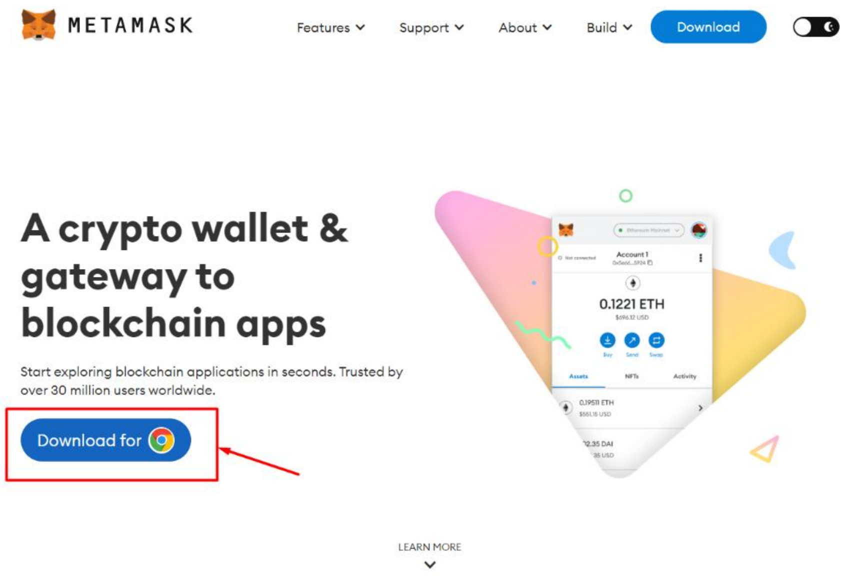
a. Install MetaMask browser extension.

b. Create a new wallet.

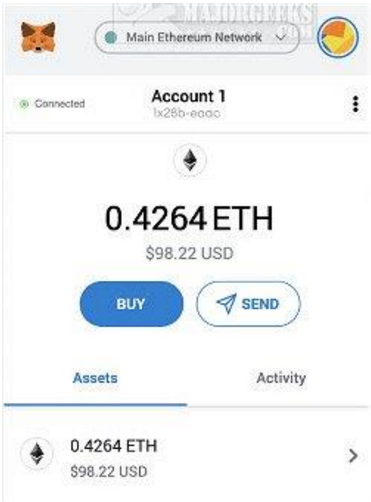
c. Fund the wallet using test Ether from a faucet.

d. Connect the wallet to the local blockchain and perform transactions.

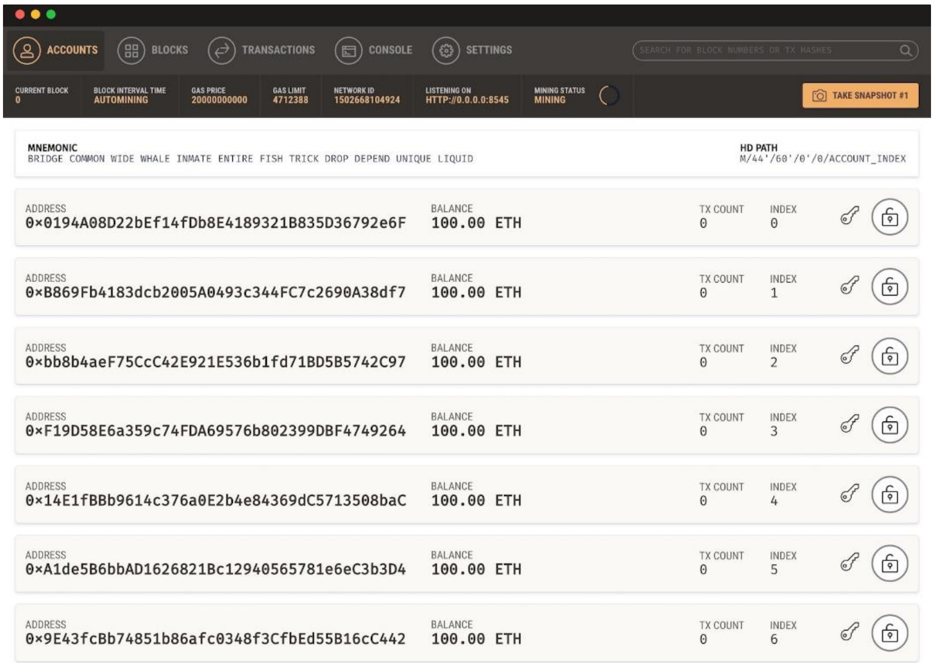
A: Install MetaMask browser extension:



b. Create a new wallet:



C: Connect the wallet to the local blockchain and perform transactions:



Lab 8: Write a simple smart contract in Solidity that prints "Hello, World!" when executed.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;
contract hello
{
string enter;
function set(string memory value) public
{
enter= value;
}
function get() public view returns(string memory)
{
return enter;
}
}
```

Lab 9: Write a simple smart contract in Solidity that prints "Hello, World!" when executed.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

contract hello

{

string enter;

function set(string memory value) public

{

enter= value;

}

function get() public view returns(string memory)

{

return enter;

}

}
```

Lab 10: //Implement an example of an ERC-20 token contract using Solidity, designed to run on the Ethereum blockchain.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ERC20Token {
    string public name;
    string public symbol;
    uint8 public decimals;
    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256
value);

    constructor(string memory _name, string memory _symbol, uint8 _decimals,
uint256 _initialSupply) {
```

```
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
        totalSupply = _initialSupply * (10 ** uint256(_decimals));
        balanceOf[msg.sender] = totalSupply;
    }

    function transfer(address _to, uint256 _value) public returns (bool
success) {
        require(_to != address(0), "Invalid address");
        require(balanceOf[msg.sender] >= _value, "Insufficient balance");

        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value;

        emit Transfer(msg.sender, _to, _value);
        return true;
    }

    function approve(address _spender, uint256 _value) public returns (bool
success) {
        allowance[msg.sender][_spender] = _value;

        emit Approval(msg.sender, _spender, _value);
        return true;
    }

    function transferFrom(address _from, address _to, uint256 _value) public
returns (bool success) {
        require(_from != address(0), "Invalid address");
        require(_to != address(0), "Invalid address");
        require(balanceOf[_from] >= _value, "Insufficient balance");
        require(allowance[_from][msg.sender] >= _value, "Allowance exceeded");

        balanceOf[_from] -= _value;
        balanceOf[_to] += _value;
        allowance[_from][msg.sender] -= _value;

        emit Transfer(_from, _to, _value);
        return true;
    }
}
```

Lab 11: Write a Solidity program to define a simple voting contract. The contract ensures that each user can vote only once and keeps a tally of votes received per candidate. It also emits an event each time a vote is cast.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleVoting {
    mapping(address => bool) public hasVoted;
    mapping(string => uint256) public votesReceived;

    event VoteCast(address indexed voter, string candidate);

    function vote(string memory _candidate) public {
        require(!hasVoted[msg.sender], "You have already voted");

        votesReceived[_candidate]++;
        hasVoted[msg.sender] = true;

        emit VoteCast(msg.sender, _candidate);
    }

    function getVotesForCandidate(string memory _candidate) public view
returns (uint256) {
        return votesReceived[_candidate];
    }
}
```