Phase 2 and Phase 3 write-ups are merged, scroll down for phase 3 write-up

Phase 2

Abstract:
The idea of sharing economy gives rise to unique ideas and develops innovative businesses. This project aims to relate the concept of smart city by introducing the smart service system and explores the opportunities of adopting blockchain technology in service industry. Currently, as we know companies like Urban-Clap, House-Joy and many more are the top leaders in providing on-demand home services. They basically provide the trust needed for a customer to use their platform and order the desired service. Our idea aims to provide the same trust with the help of blockchain technology resulting in a decentralized infrastructure. We propose blockchain-based framework for the existing centralized framework.

Background:
Initially, inspired from the companies like Urban-clap and House-Joy we thought of providing the services through "by the users – for the users" model. The challenging here was providing the trust which encapsulates validation, verification, authentication and integrity of the user of both the sides i.e., provider and consumer. Narrowing it down form the general public to the students at a particular university we thought of providing the trust by verifying a particular student by their unique college ID. Since this was a valid solution to apply but we did not consider the future scenarios, so our phase I submission is based on the model for students and not for the general public. It was realized that if we verify students by their unique college id the model becomes dependent on the university to provide the information and the legal issues made it complicated. So, we turned around to our previous model for the general public.

Existing Centralized Model:

1. **Trust**: It is provided by verifying the information of a given user digitally by accessing their legal documents providing them a user account storing all their information in a centralized storage which is expensive in terms of cost of the infrastructure.
2. **Lack of Transparency**: As the account is provided by a centralized model the authority can access and modify the details like reviews, ratings, count of orders etc.
3. **Pricing**: The prices are usually high because the revenue gets divided into the user and the company providing the platform. So, either the user takes the subscription, or a fixed share is given to the company due to which the pricing is high.
4. **Authenticity**: The reviews and ratings based on which a consumer may select a particular provider can be manipulated either by the company itself or by the consumers. E.g., A fraud order placed with a cash on delivery option and the consumer asks the user to put a fake review and rating. Even if the transaction takes place the transaction can go out of the system reverting the money sent.

Proposed Model:

As mentioned earlier the trust is the challenging part in this idea. To provide this trust essential use of smart contract and the block chain technology is being done. The proposed model works in following way:

1. The service provider registers his/her information including (legal documents & service-related details) with a minimum deposit, so that if a provider does some unusual activity, we can deduct the fine from deposit amount. For simplicity in this project, we are only collecting basic information excl. legal documents
2. After registering the service will be displayed in the marketplace section to all the users (consumers) who are on the blockchain network.
3. The consumer books a service they need and the status of the service for a given time will changed to book. As a matter of security some percent of the rate will be deducted from the consumers account to state that the service is booked and if cancelled the amount will not be returned (or can cancel before a specific time). As these features are out of scope of blockchain technology, for simplicity we are just showing the service in the marketplace keeping our focus on the blockchain methods.
4. After the service is completed, the consumer needs to pay from their wallet. Here, there is a constraint that the consumer should first review or rate the provider before moving to the payment otherwise no transaction will take place. This constraint can be placed by various business strategy but as it is out of scope, we are just keeping it mandatory to rate the provider.
5. After rating the provider, the consumer just needs to click on the Pay button and the specified amount will be send from consumers wallet to the providers wallet.

Smart Contract Functionality:

1. **Variables:**
    a. address chairperson: the one who deploys the smart contract and can remove a provider from the system. (Next phase implementation)
    b. uint120 minDeposit: amount required to store the deposit amount.
    c. uint8 minAge: to declare a constraint on age.
    d. uint256 transactionCount: to keep a count of the transactions
2. **Data Structures:**
    a. ProviderStruct: Structure to store provider-related information.
    b. TransferStruct: Structure to store transaction details.
    c. TransferStruct[] transactions: Array to store all the transactions
    d. Mapping(address=>ProviderStruct) services: A hash map to store all the registered services.

3. **Events:**
    a. Event Transfer (): used to emit so that the blockchain networks knows that transaction took place
4. **Modifiers:**
    a. validRegistration(): ensures the if the registration can be done by checking the deposit and age of the provider.
    b. onlyOnce(): ensures that the provides registers only once and rejects if the same address is encountered again.
5. **Functions:**
    a. registerService(): this function registers i.e. stores the information of service provider in the smart contract using above mentioned variables. It returns true if success else false.
    b. pay(): stores the transaction details, increases the transaction count, updates the ratings given and finally emit the Transfer event so that the transfer is made.
    c. updateRatings(): used to calculate the average rating based upon the count of the ratings and update the rating.
    d. getAllTransactions(): returns the transaction array to render it on the front-end.
    e. getAllServices(): returns the providers array.
6. **Working:**
    a. First step is to connect the wallet through the Metamask and register for a service.
    b. The smart contract is invoked while registering a particular servicer which ensures that the registration takes place only once by checking with the help of modifier. It also ensures the deposit amount is correct else revert the transaction.
    c. The information received is mapped to a particular address with all the associated details and store it in an array of providers.
    d. To book a particular service log into different account, click on the Book button (assuming real life situation that the provider has completed the service) to pay the provider.
    e. The Metamask wallet is invoked to transfer the mentioned amount of ether in the provider's profile. Without rating the payment will not proceed, as discussed earlier to establish the trust based on ratings this step is mandatory.

Web App functionality:

1. **Folder Structure:**
    a. Client:
        i. We have used ReactJS as framework to create the front-end components and to contact with the smart contract.
        ii. All the components are stored inside the folder **src/components**.
        iii. To interact with the smart contract, React Context is being used to avoid the prop passing through multiple components. The context file is under the **src/context** folder.
        iv. Basic util functions, information and the contract ABI are under the folder **src/utils**
    b. Smart Contract:
        i. We are using hardhat as a framework to compile and deploy the smart contract, like truffle.
        ii. We are also using alchemy as a platform to deploy our node on the Ropsten network, this will also help to migrate the application from test network to main network.
        iii. Ethers library for interacting with the Ethereum Blockchain Ecosystem.
2. **Implementation:**

Once the contract is deployed and the ABI is copy pasted into client folder under **src/utils/Services.json** the Client side can interact with the contract with the help of **src/context/ServiceContext.js** file. The context file working goes as following:

getEthereumContract(): This function fetches the web3 provider, signer and creates a contract using ethers.Contract() function and returns it.

```
//we get access to the ethereum object by metamask
const { ethereum } = window;

//fetch the ethereum contract
const getEthereumContract = () => {
    const provider = new ethers.providers.Web3Provider(ethereum);
    const signer = provider.getSigner();
    //get the contract
    const serviceContract = new ethers.Contract(
        contractAddress,
        contractABI,
        signer
    );

    console.log({
        provider,
        signer,
        serviceContract,
    });

    return serviceContract;
};
```

The next is the react functional component which is being served as the Provider for this context. Important functions within that component are discussed below.

registerService(): the function first checks if the Ethereum provider is provided if not then then alert is thrown. Next, it gets the Smart Contract by calling the above function and makes a call to the smart contract function to register a service.

```
const registerService = async(name, deposit, serviceName, age, serviceFee) => {
    console.log("called register")
    try {
        if(!ethereum)
            return alert("Please install metamask");
        const serviceContract = getEthereumContract();

        //make sure wallet is connected (future)

        //call register service form contract
        const status = await serviceContract.registerService(
            name, deposit, serviceName, age, serviceFee);

        status.wait();

        if(status){
            console.log("Registration success");
        }


    } catch (error) {
        console.log(error);
        alert("Registration failed");
        throw new Error("SMART CONTRACT, registration failed");
    }
}
```

Throws error if the registration fails due to some error.

getAllServices() and getAllTransactions(): this two functions are similar in nature they do the initial check for the smart contract, call the appropriate functions, destructure and structure the received data and return it to render on the front-end.

```
const getAllServices = async () => {
    try {
        if(!ethereum) return alert("Please install metamask");
        const serviceContract = getEthereumContract();

        console.log("calling get all services", serviceContract);
        const availableServices = await serviceContract.getAllServices();

        console.log(availableServices)
        const structuredServices = availableServices.map((service) => ({
            name: service.name,
            deposit: service.deposit,
            serviceName: service.service,
            age: service.age,
            serviceFee: service.serviceFee,
            address: service.serviceFrom,
        }))

        setServices(structuredServices);
        console.log(structuredServices);

    } catch (error) {
        console.log("Failed to get the services", error);
    }
}
```

Below are some, we can call util functions, to do the routine checkup and render the data as available.
CheckIfTransactionExist() checks if any transaction exist for the connected wallet account and returns if any found.

checkIfWalletConnected() ensures that the wallet is connected to so that the account address is used as a current account.

connectWallet() connects the application to the metamask.

```
const checkIfTransactionsExist = async () => {
    try {
        const transactionContract = getEthereumContract();
        console.log("calling transactions count", transactionContract);
        const transactionCount = await transactionContract.getTransactionCount();
        console.log("transactions count");
        window.localStorage.setItem("transactionCount", transactionCount);

    } catch (error) {
        console.log(error);
        // throw new Error("No ethereum object.");
    }
}

//check if metamask is installed and get the connected accounts
const checkIfWalletIsConnected = async () => {
    try {
        //can use ethereum.isMetaMask
        if (!ethereum) return alert("Please install metamask");

        //request to get the accounts if meta mask connected
        const accounts = await ethereum.request({ method: "eth_accounts" });
        if (accounts.length) {
            setCurrentAccount(accounts[0]);
            getAllTransactions();
        } else {
            console.log("No accounts found");
        }
    } catch (error) {
        console.log(error);

        throw new Error("No ethereum object.");
```

sendTransaction() , this function is little lengthy so providing a brief description. It sends the amount of ether given to it via ethers.request() function. After the transaction we emit the event in the smart contract, we wait for the hash value, and increase the transaction count.

The functions are supposed to be executed on the page load are specified inside useEffect() hook call.

Finally, all the functions and state variables which are needed by child components are provided through the React context provider.

Phase 2 and Phase 3 write-ups are merged, scroll down for phase 3 write-up

Features:

Trust: Our aim is to provide the trust to the user through the rating and transaction system. The transaction ledger is transparent as we are providing the transactions occurred on the network. And as the ratings are mandatory due to which even if some rating is fake the transaction will be recorded and will be tracked and monitored by the chairperson if suspicious the provider will be removed from the network.

Transparency: Our model provided transparency in terms of making transaction record accessible in the application itself. Users can also see the transactions and ratings of the provider which again increases the amount of trust.

Pricing: As the infrastructure cost is less to store and manage the data and no other employees are needed to run the platform the charge applied to use the platform will be a minimal and far less than the centralized model.

Authenticity: As the ratings will be only given by the people on the network which are already identified by their unique address and the above transaction constraint the ratings/reviews are always guaranteed to be authentic.

Limitations:

For simplicity following functionalities are not included:

sign-up, login and data storage functionality.
Separate sections/views for users, providers and their respective profiles, everything on a single page.
Direct assumption that the order is complete, and the user pays.

Phase 3

ERC-20 smart contract:

Overview:

- The smart contract has three main sections viz., 1. SafeMath contract 2. ERC20 interface and 3. Token contract. As contracts are similar to classes, we are inheriting the Safe-Math contract and ERC20 interface in the Tokens contract. By inheriting we use or override the methods defined in the respective contracts/interfaces.
- Safe-Math contract has methods defined to perform basic mathematical operations like add, subtract, divide and multiply.
- ERC-20 interface has methods defined according to the ERC-20 standards which are used in the Tokens contract to operate on the actual tokens.

```
interface ERC20Interface{
    function totalSupply() external  returns(uint);
    function balanceOf(address tokenOwner) external  returns(uint balance);
    function allowance(address tokenOwner, address spender) external  returns(uint remaining);
    function transfer(address to, uint tokens) external returns (bool success);
    function approve(address spender, uint tokens) external returns (bool success);
    function transferFrom(address from, address to, uint tokens)  external returns (bool success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
}
```

Tokens Contract:

Variables and Data Structure:

- The variables and data structures needed are:

```
contract SDToken is ERC20Interface, SafeMath{
    string public symbol;
    string public name;
    uint8 public decimals;
    uint public _totalSupply;
    address public owner;

    mapping(address => uint) balances;
    mapping(address => mapping(address => uint)) allowed;
    mapping(address => bool) statusList;
    address[] airDropList;
```

- Balances is a mapping of address to uint to store/track balances of a particular address.

- Allowed is a mapping used to track who issued the request for which address and how many tokens.
- Status-list is used to track the addresses who have received the airdrop so that duplicate entries and re-sending the tokens are avoided.
- Airdrop-list is used to track the addresses who have requested for the airdrop.

Constructor and Modifiers:

- The constructor generates the token related information and stores the initial total supply of the tokens in the deployers wallet i.e. transferring tokens to the deployer's address.

```
constructor(){
    owner = msg.sender;
    symbol = "SD";
    name = "ServiceDoor Coin";
    decimals = 2;
    _totalSupply = 100000;
    balances[0xfBf0197035FFa3a31DFCe02DFdB0f897eF1A0D8a] = _totalSupply;
    emit Transfer(address(0), 0xfBf0197035FFa3a31DFCe02DFdB0f897eF1A0D8a, _totalSupply);
}

modifier onlyOwner (address from) {
    require(from == owner);
    _;
}

 modifier sufficientBalance(address sender, uint tokens){
     if(sender == owner){
         require(balances[sender] > tokens);
     }
    else{
        require(balances[sender] >= tokens);
    }
    _;
}
```

- Modifier onlyOwner ensures the request received is from the owner.
- Modifier sufficientBalance ensures the balance is enough to transfer the tokens.

Utils Methods:

- These methods are helper methods in the process of token operations.
- Function totalSupply returns the total number of tokens initially transferred to the deployers address.
- Function balanceOf returns the current balance of a particular address which is holding the tokens.

- Function addRecipients is used to add the address who requested airdrop.

```
function totalSupply () override  public view  returns (uint){
    return _totalSupply - balances[address(0)];
}

function balanceOf(address tokenOwner) override  public view  returns(uint balance){
    return balances[tokenOwner];
}

function addRecipients(address to) public returns(bool success) {
    if(statusList[to]) {
        return false;
    }
    airDropList.push(to);
    statusList[to] = false;
    return true;
}

function getAirDropList() public view returns (address[] memory){
   return airDropList;
}

function airDrop(address to, uint tokens) onlyOwner(msg.sender) public returns(bool succ
    if(statusList[to]) return false;
    transfer(to, tokens);
    statusList[to] = true;
    return true;
}

function cleanUp() public returns(bool) {
    for(uint i=0; i<airDropList.length; i++){
        airDropList.pop();
    }
    return true;
}
```

- Function getAirDropList returns the airdrop list so that we can operate on it.
- Function airdrop is used to initiate the airdrop to the receivers i.e., addresses in the airdropList. It first checks if the sender is owner because only the deployer is allowed to airdrop tokens. It also checks the statusList to ensure that no address receives the token again, if found return false, else it in-turn calls transfer function to transfer the tokens.
- Function cleanUp is used to empty the list of the receivers. It is called from the front-end after the airdrop is completed. This way every time we initiate an airdrop, we have new addresses to process.

Main Methods:

- These methods are used in the process of transferring the tokens.
- Function transfer is used to transfer a given number of tokens to a given receiver's address. It emits an even which is caught by the nodes on the network.

```
function transfer(address to, uint tokens) override public returns(bool success){
    // if(statusList[to]) return false;
    balances[msg.sender] = safeSub(balances[msg.sender], tokens);
    balances[to] = safeAdd(balances[to], tokens);
    // statusList[to] = true;
    emit Transfer(msg.sender, to, tokens);
    return true;
}

function approve(address spender, uint tokens) sufficientBalance(spender, tokens) ove
    allowed[msg.sender][spender] = tokens;
    emit Approval(msg.sender, spender, tokens);
    return true;
}

function transferFrom(address from, address to, uint tokens) override public returns
    balances[from] = safeSub(balances[from], tokens);
    allowed[from][msg.sender] = safeSub(allowed[from][msg.sender], tokens);
    balances[to] = safeAdd(balances[to], tokens);
    emit Transfer(from, to, tokens);
    return true;
}

function allowance(address tokenOwner, address spender) override public view returns
    return allowed[tokenOwner][spender];
}
```

- Function approve is used to ask for the approval of the tokens to be sent. Currently to approve tokens only balance is checked. In ideal scenario, it will also check if the transaction being sent is a valid transaction, the receiver and the sender belong to the application and much more. For simplicity purpose we are limiting this functionality.
- Function transferFrom is a similar function we created for a situation when a transaction will be requested on behalf of the sender. For e.g., receiver requests tokens from a sender and sender just needs to sign the transaction.
- Function allowance returns the allowed number of tokens for a particular transaction. Depending on the nature of the transaction the number of tokens approved may vary.

Connection:

- To connect the Token contract in such a way that the Services contract will be able to call the functions in the token contract we are using React context feature. As mentioned in earlier phase we are using React as framework so to connect with a contract we are using a context file which connects to the smart contract and interacts with its functions.
- Similarly, we have created a context file for Tokens contract and are interacting with it in the context file. This context file can be imported in the context file of Services contract and can be used to call the functions defined in the context file which in-turn will interact with the Tokens contract.
- This way the Services contract can interact with the Tokens contract. Though we had other way to approve and connect the contract we thought this can be the efficient way to connect and interact with the other contract without spending gas or transaction fee for the interaction.

Deployment:

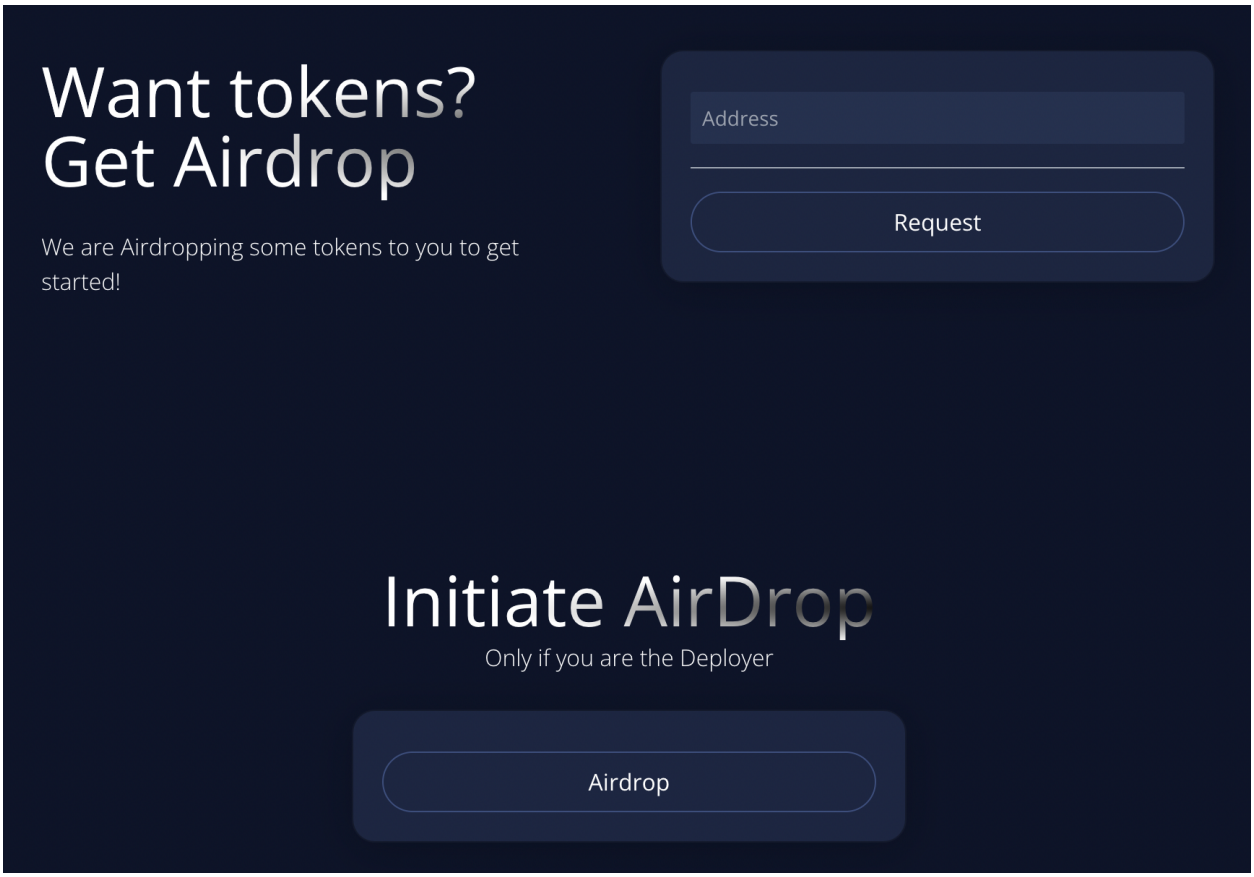
- The deployment process is like what we did in the first phase i.e., use hardhat to deploy on the specified network i.e., ropsten. After successfully deploying the contract, we get the address of the contract where it has been deployed.
- We need to copy that address in the following locations:
    - **Client/src/utils/constants.js** so that we can use the contract address in the front-end.
    - **Import into wallet,** this will let a user to import the tokens and see the token balance and activity in the Meta-mask.
- Last step is to copy the ABI file from artifacts to the **client/src/utils/Tokens.json** this will give us access to the ABI.

Working:

- Working will be same as the previous phase the only difference is that now the transactions will be done in-terms of Tokens and not in ETH.
- To be able to transact with Tokens one first needs tokens in the wallet, for that a user needs to submit their address for the airdrop facility on the homepage. After submitting the deployer or the admin will initiate the airdrop and the respected receivers will get 10 tokens as a demo. In future implementation, to get more tokens there will be either a subscription model or some payment method required.
- Now that users have some tokens in account, we follow the same procedure to interact with the application. On clicking the Pay button, the call goes to the Tokens contract, the tokens are first approved and then transferred to the designated address.

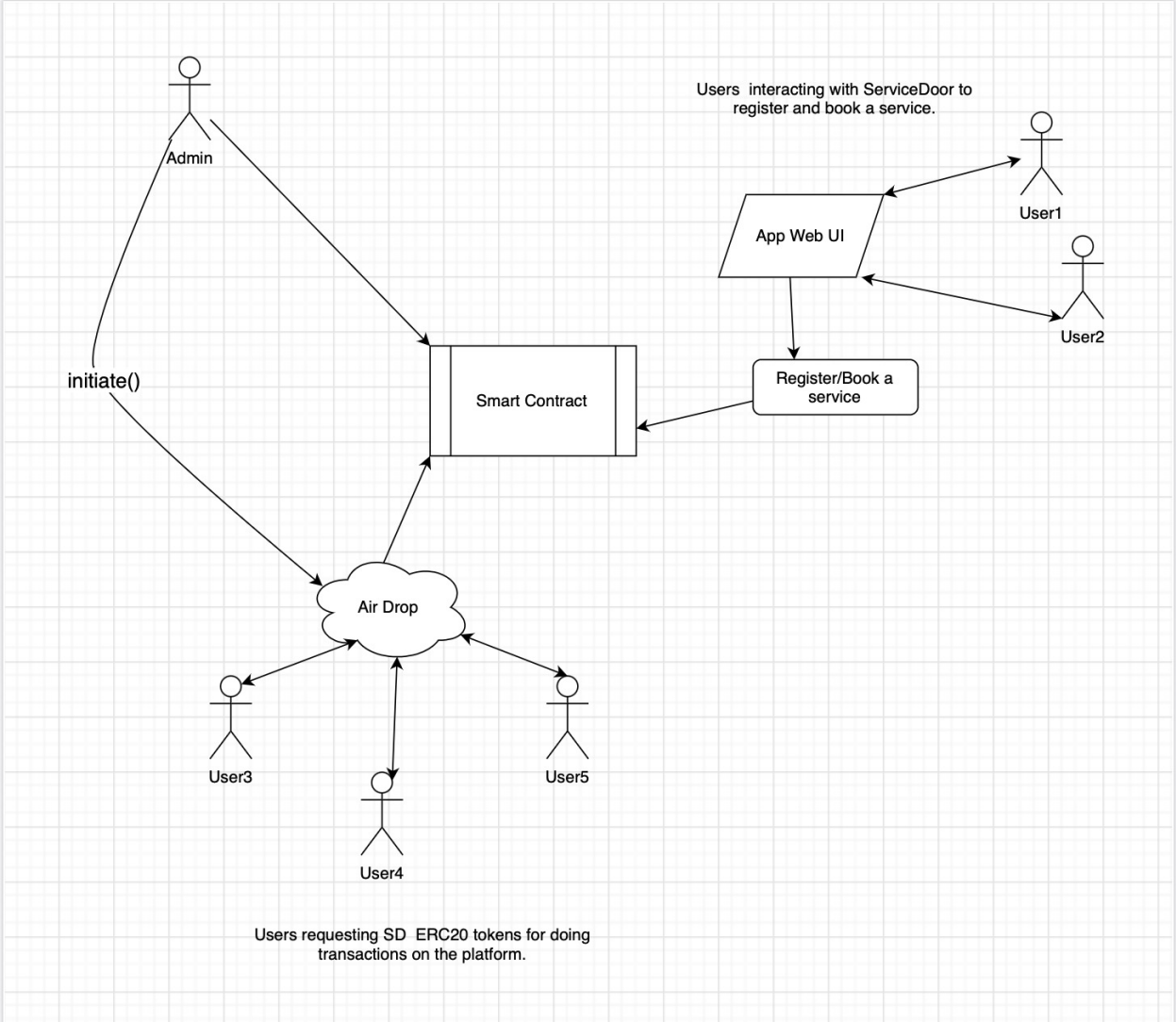
Airdrop:

- To perform airdrop from the homepage, there is a section which contains an input box where the users can register their addresses to receive the airdrop.
- If you are the deployer of the smart contract, then you can initiate the airdrop by clicking the Airdrop button on the homepage. Registered addresses will receive the tokens via airdrop.
- Initially, we offer 10 tokens to the users to get started. For future implementation user should request for the number of tokens in exchange of specific Ethers. This way the business-revenue model runs efficiently.



Phase 2 and Phase 3 write-ups are merged, scroll down for phase 3 write-up

Architectural Diagram:



Note:

- Node modules are removed, please run npm init before running the project.
- Please refer to the phase 2 demo video and report for more details which are skipped in the phase 3 demo video to adjust the video recording in time.
- To deploy the Token contract, you need to copy paste your account address as the deployer address so that the initial token supply will be in your address.
- While deploying you can either deploy both the smart contracts or either one.
- Please refer to Alchemy documentation for more information on the client-node service.