Airport Baggage Claim

Decentralized System

https://youtu.be/nKLNoE0kOGU

Project Phase 2

Shubhangi Mishra - 50373871
Akshay Sahai - 50419149

User Interface

The project consists of two sections:

1. ***Baggage Upload and Airline Register***
This is the more usual part of the platform where airports are able to register and upload the unclaimed baggage that have not been claimed by the customers and have been lost in the long itinerary. For this, airports will first register themselves on the platform and the airports will register themselves

2. ***Baggage Claim***
This is a more usual part of the platform where customers are able to claim the baggage they want to claim and can select the number of bags they want. For this customers have to wait for the response of the airport that their request has been updated and then they can pay the airport once the request is approved

The BGK consortium authority will have a monitor, whom you will refer to as the airport authority of the consortium. This consortium does not mean centralization, because this monitoring or chairperson role is periodically circulated among the consortium members. The chairperson does not manage any central database. The database of each airport is safe within its firewalls. The shared data is stored in the distributed ledger of the shared blockchain.

Assets

The data assets are the airport's unclaimed baggage and the funds held by the claiming customers. We know that the fundamental tenet of a decentralized system is that peer participants—not an intermediary—hold their own assets.

ROLES

The following are the roles:
1. Agents acting on behalf of the airports can enroll or self-register by using the register() function with an escrow/deposit this action makes them (airports) members of BaggageClaim.
2. Agents (of members only) can request upload baggages for claim.
3. Customers can see the bag number and per bag price and can ask the number of bags he wants to claim
4. The request only passes the system if the customer has sufficient funds in the account
5. Agents acting on behalf of the airports can upload the baggage and once claim by the customer have the authority to approve all the claims if they want to in case they have a prebooking or some exceptional condition occurring.
6. Customers settle the payment once the response status is allowed and payment is being deducted from their account and added to the airport account.
7. The airport authority of the consortium has the sole authority to unregister members and return leftover deposits back to the airport if they want to unregister themselves.

The definition of roles is critical in an automated decentralized system, so you want to be certain that authorized participants initiate the requests. Agents can be human or software applications.

DATA TYPES

The new Solidity data types are introduced by this smart contract. These data types include

- address to refer to the identity of the airport authority.
- struct to collectively define the data of the airports, airport baggage, baggage-claim, including the escrow or the deposit.
- mapping to map account addresses (identities) of members to their details. (A mapping is like a hash table.)
- modifier definitions for airportOnly, customerOnly and chairpersonOnly.

These data types are followed by the function definitions: constructor(), registerAirports(),addBaggage(),requestToClaimBaggage(),responseToClaimBaggage(),settlePayment(), and unregisterAirports(). It's important to note that the airports have to execute their regular functions and checks by using their existing systems. Note the use of the msg.sender, msg.value, and payable features. The smart contract only takes care of the extra functionality needed for the decentralized interaction with airports and customers.

TRANSACTIONS

A typical process of buying unclaimed baggages may involve many operations and interactions with various subsystems, such as databases.
Let's refer to operations that need to be verified, validated, confirmed, and recorded by all parties as the transactions, or simply Txs.

RULES

The contract diagram shows new elements that did not appear in the counter use case:
 Modifiers are special elements of a smart contract, representing the rules that act as gatekeepers to control access to data and functions.
Only valid airports (onlyAirport) can transact on the system, Only valid customers (onlyCustomers) can transact on the system, ,and only the airportAuthority (onlyAirportAuthority) can unregister any airport.

The BGK codes demonstrate the use of modifiers in a smart contract with three modifiers: onlyCustomer, onlyAirport and onlyAirportAuthority

The airports representatives (on behalf of the airoprts) can initiate the trades proactively or reactively in response to customer demand or as warranted by circum- stances such as baggage lost. In this use case, you'll limit the scope to the elemental operation of peer-to-peer baggage claim of unclaimed baggage among airports and customers. Enforcement of agreed rules of engagement and a seamless payment system is also enabled by the blockchain, alleviating the traditional business concerns of airports way of handling unclaimed baggage.

Sequence of operations

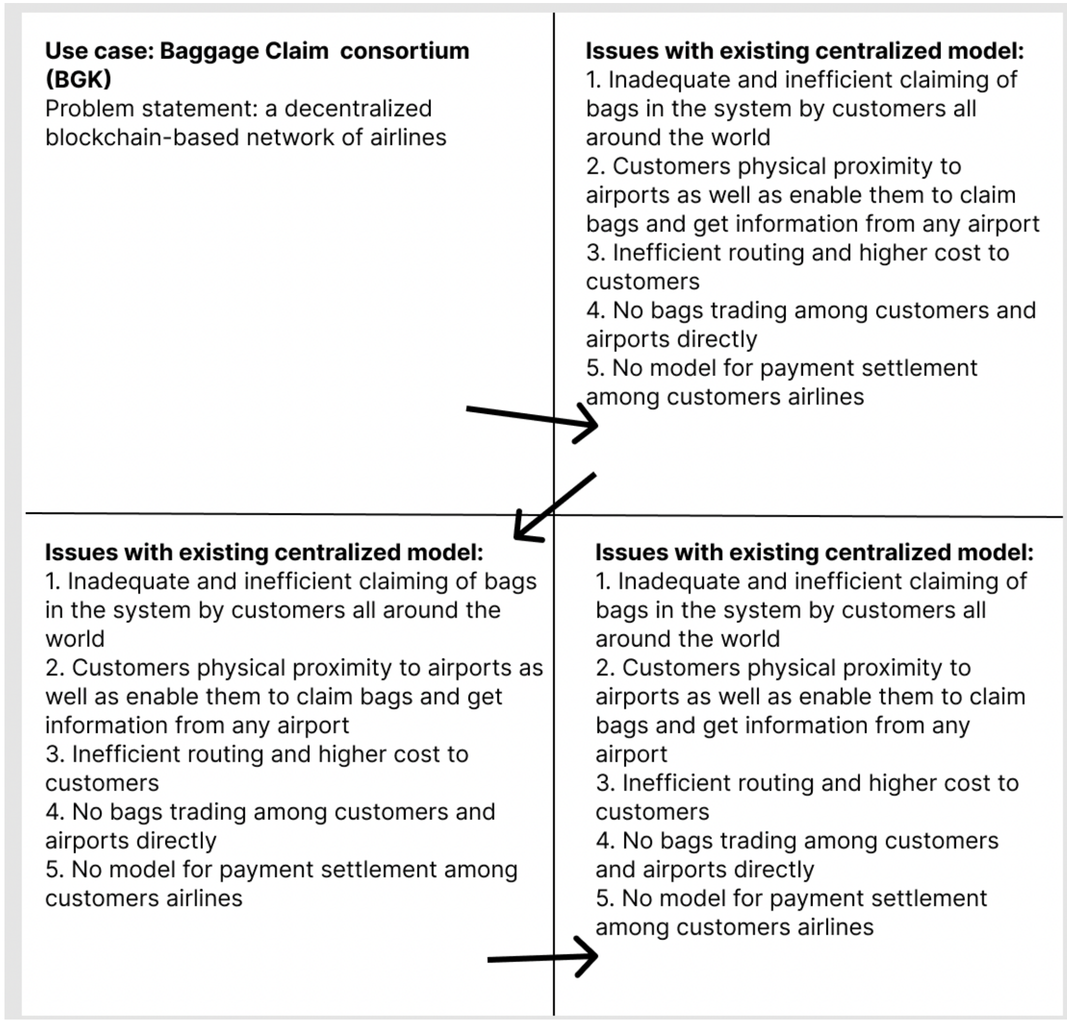
Here are the steps followed by two airports and customers that are not necessarily known to each other and that operate beyond the boundaries of traditional trust. In other words, they don't have a conventional business partnership such as code-sharing and alliances. How can the airports and customers trust each other to verify and record the transactions?

Consider this everyday situation: your sister wants to borrow $10, all in $1 bills. You take the bills out of your wallet, count them, verify there are 10 $1 bills, and hand them to her. She pockets the bills and leaves. She does not count them again because she trusts you.

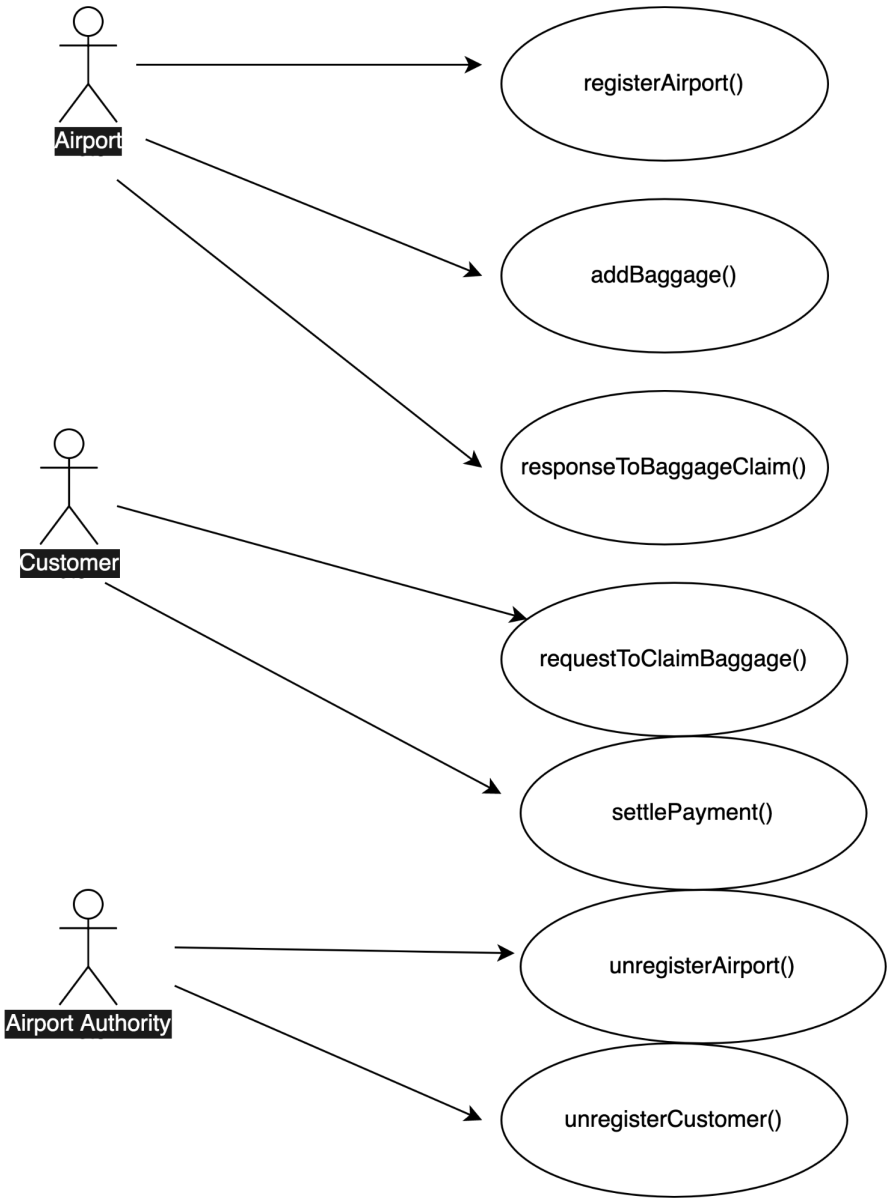
Now imagine the same transaction at a checkout counter. You count out (verify) 10 $1 bills right in front of the checkout clerk and hand the money to her. In this case, the clerk verifies that you have given her the right number of bills by counting them again. Why? You two are decentralized entities that don't know each other well enough to trust each other.

That is the situation with airport and customer, which are not known to each other. In BGK, they rely on the trust established by the blockchain by verifying and recording the transactions. In addition to the two airport and customer, other stakeholders may veriify the transactions and record the transactions as witnesses

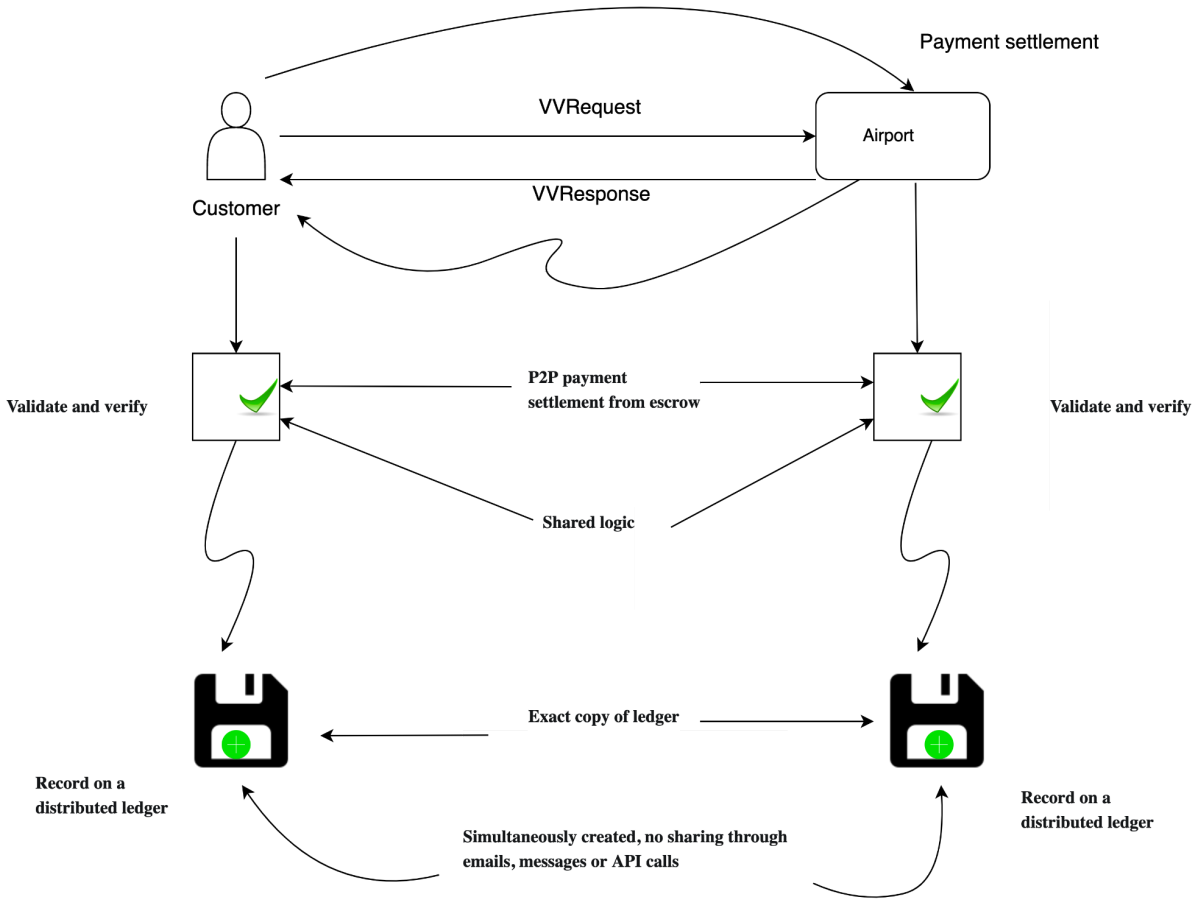
Let's analyze the operations indicated in figure to understand the role that a blockchain plays in a decentralized system as a verifier, validator, and recorder. Figure shows the numbered sequence of operations. Follow the operations in the sequence numbered to get the operational details discussed.



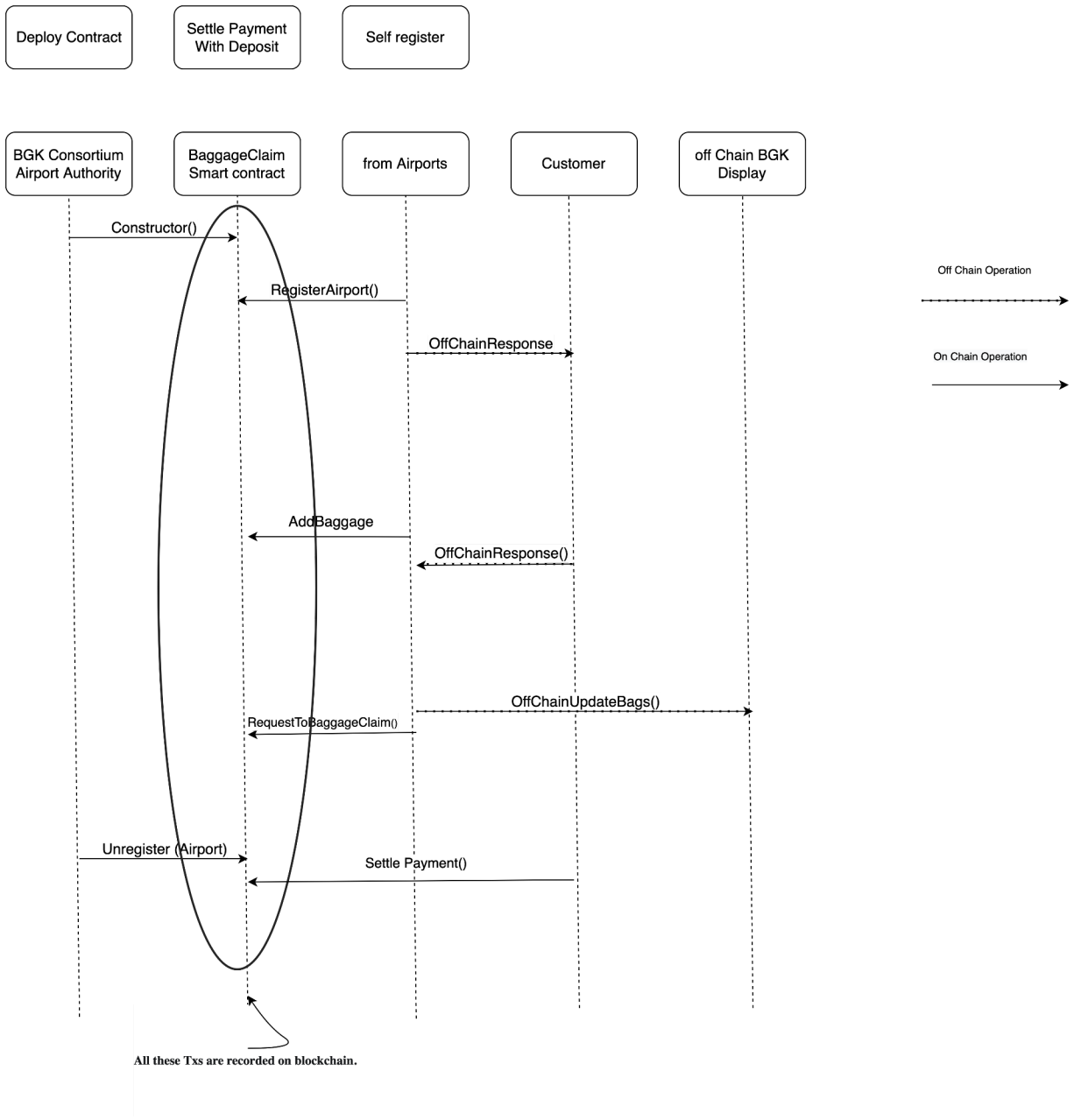
Quad Chart Diagram



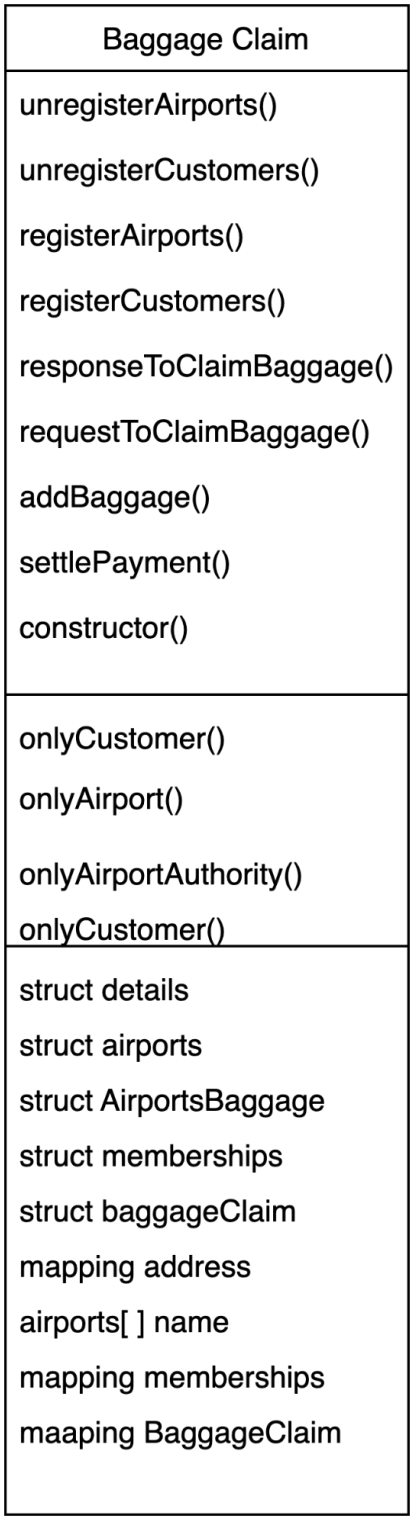
Use Case Diagram



Operations of participants in a decentralized baggage claim system



BGK sequence diagram



Contract Diagram

Smart Contract Code

```
pragma solidity >=0.4.24;

contract BaggageClaim{

   address airportAuthority;
   uint public myBalance;

   struct details{
       uint escrow;
       uint status;
       uint hashOfDetails;
   }

   struct Memeberships{
       uint memebershipStatus;//0 new, //1-airport,//2-authority//3-customer
   }

   struct Airports{
       address airportAddress;
       string airportName;
   }

   struct AirportBaggage{
       uint timestamp;
       uint pricePerbag;
       uint totalBaggage;
       uint totalUnclaimedBaggage;
       uint totalBaggageClaimed;
   }

   struct ClaimRequest{
       uint requestStatus;
       uint requestQuantity;
   }

   mapping (address => details) balanceDetails;
   mapping(address => AirportBaggage) airportBaggage;
   mapping(uint => ClaimRequest) claimRequest;
   mapping(address => Memeberships) memberships;
```

```
        Airports[] airportNames;

        modifier onlyAirportAuthority(){
            require(msg.sender == airportAuthority);
            _;
        }

        modifier onlyAirport(){
            require(memberships[msg.sender].memebershipStatus==1);
            _;
        }

        modifier onlyCustomer(){
            require(memberships[msg.sender].memebershipStatus!=1);
            require(memberships[msg.sender].memebershipStatus!=2);
            _;
        }

        constructor () payable public{
            airportAuthority = msg.sender;
            balanceDetails[msg.sender].escrow = msg.value;
            memberships[msg.sender].memebershipStatus = 2;
            myBalance = address(this).balance;
        }

        event LogMessage(string message, uint value);



                                                    // Airport Functions //

        function registerAirports(string memory airportsName, uint pricePerbag)
public{
            if(msg.sender==airportAuthority){
                revert();
            }
            if(memberships[msg.sender].memebershipStatus==3){
                revert();
            }
            address newAirports =msg.sender;
            airportBaggage[newAirports].timestamp = block.timestamp;
            memberships[newAirports].memebershipStatus = 1;
            airportBaggage[newAirports].pricePerbag = pricePerbag;
```

```
            airportBaggage[newAirports].totalBaggage = 0;
            airportBaggage[newAirports].totalBaggageClaimed = 0;
            airportBaggage[newAirports].totalUnclaimedBaggage = 0;
            balanceDetails[newAirports].escrow = 0;
            airportNames.push(Airports(newAirports, airportsName));
        }

        function unregisterAirports(address payable toAirports) public
onlyAirportAuthority{
            if(memberships[toAirports].memebershipStatus!=1){
                revert();
            }
            memberships[toAirports].memebershipStatus = 0;
            balanceDetails[toAirports].escrow = 0;

        }

        function addBaggaage(uint noOfBaggage) public onlyAirport{
            airportBaggage[msg.sender].totalBaggage+=noOfBaggage;
            airportBaggage[msg.sender].totalUnclaimedBaggage+=noOfBaggage;

        }


        function responseToClaimBaggage(uint done, uint hashOfDetails) public
onlyAirport{
            if(claimRequest[hashOfDetails].requestStatus!=0){
                revert();
            }
            balanceDetails[msg.sender].status = done;
            balanceDetails[msg.sender].hashOfDetails = hashOfDetails;
            claimRequest[hashOfDetails].requestStatus = done;
        }



                                                    // Customer Functions //
        function registerCustomer() public{
            if(msg.sender==airportAuthority){
                revert();
            }
            if(memberships[msg.sender].memebershipStatus==1){
```

```
                revert();
            }
            address customer =msg.sender;
            memberships[customer].memebershipStatus = 3;
            balanceDetails[customer].escrow = 0;
        }

        function unregisterCustomer(address payable customer) public
onlyAirportAuthority{
            if(memberships[customer].memebershipStatus!=3){
                revert();
            }
            memberships[customer].memebershipStatus = 0;
            customer.transfer(balanceDetails[customer].escrow);
            balanceDetails[customer].escrow = 0;

        }

        function requestToClaimBaggage(address fromAirport, uint hashOfDetails, uint
noOfBaggage) public payable onlyCustomer{
            uint totalAmountNeeded =
(airportBaggage[fromAirport].pricePerbag)*noOfBaggage;
            if(airportBaggage[fromAirport].totalUnclaimedBaggage<noOfBaggage){
                revert();
            }
            if(memberships[fromAirport].memebershipStatus!=1){
                revert();
            }
            if(totalAmountNeeded > msg.value){
                revert();
            }

            address newBuyer = msg.sender;
            balanceDetails[newBuyer].status = 1;
            balanceDetails[newBuyer].escrow = msg.value;
            emit LogMessage("Pmsg.value ", msg.value);
            balanceDetails[msg.sender].hashOfDetails = hashOfDetails;
            claimRequest[hashOfDetails].requestStatus = 0;
            claimRequest[hashOfDetails].requestQuantity = noOfBaggage;

        }
```

```
        function settlePayment(address payable toAirport, uint hashOfDetails) public
payable onlyCustomer{
            uint quantity = claimRequest[hashOfDetails].requestQuantity;
            uint amt = quantity * airportBaggage[toAirport].pricePerbag;
            emit LogMessage("Payment of ", amt);
            emit LogMessage("Payment of airport sent ", amt);
            if(balanceDetails[msg.sender].escrow<amt){
                revert();
            }
            if(claimRequest[hashOfDetails].requestStatus!=1){
                revert();
            }

            address customer = msg.sender;

            balanceDetails[toAirport].escrow = balanceDetails[toAirport].escrow +
amt;
            balanceDetails[customer].escrow = balanceDetails[customer].escrow-amt;
            airportBaggage[toAirport].totalBaggage =
airportBaggage[toAirport].totalBaggage-quantity;
            airportBaggage[toAirport].totalUnclaimedBaggage =
airportBaggage[toAirport].totalUnclaimedBaggage-quantity;
            airportBaggage[toAirport].totalBaggageClaimed+=
airportBaggage[toAirport].totalBaggageClaimed+quantity;
            balanceDetails[toAirport].hashOfDetails = hashOfDetails;
            emit LogMessage("Customer Balance before ",
balanceDetails[customer].escrow);
            toAirport.transfer(amt);
            emit LogMessage("Balance after ", balanceDetails[toAirport].escrow);

        }

        function getTotalUnclaimedBaggage() public view returns(uint){
            return airportBaggage[msg.sender].totalUnclaimedBaggage;
        }

        function getAirportClaimedBagagge() public view returns(uint){
            return airportBaggage[msg.sender].totalBaggageClaimed;
        }

        function getBalance() public view returns(uint){
            return balanceDetails[msg.sender].escrow;
```

```
    }

    function getAirportsCount() public view returns(uint){
        return airportNames.length;
    }

    function getAirportsName(uint index) public view returns(string memory){
        return airportNames[index].airportName;
    }

    function getAirportsAddress(uint index) public view returns(address){
        return airportNames[index].airportAddress;
    }

    function getAirportPricePerBag() public view returns(uint){
        return airportBaggage[msg.sender].pricePerbag;
    }

    function getMembershipStatus() public view returns(uint){
        return memberships[msg.sender].memebershipStatus;
    }

}
```

BGK smart contract deployment and testing

Now create a Solidity file in remix contract folder called BaggageClaim.sol, and enter the code from listing , available in the codebase of thisreport. Compile it, using the Compile command on the menu in Remix; select JavaScript VM as the environment; and click Deploy & Run transactions icon. Now you are ready to deploy and test the application on the simulated VM provided by the Remix IDE.

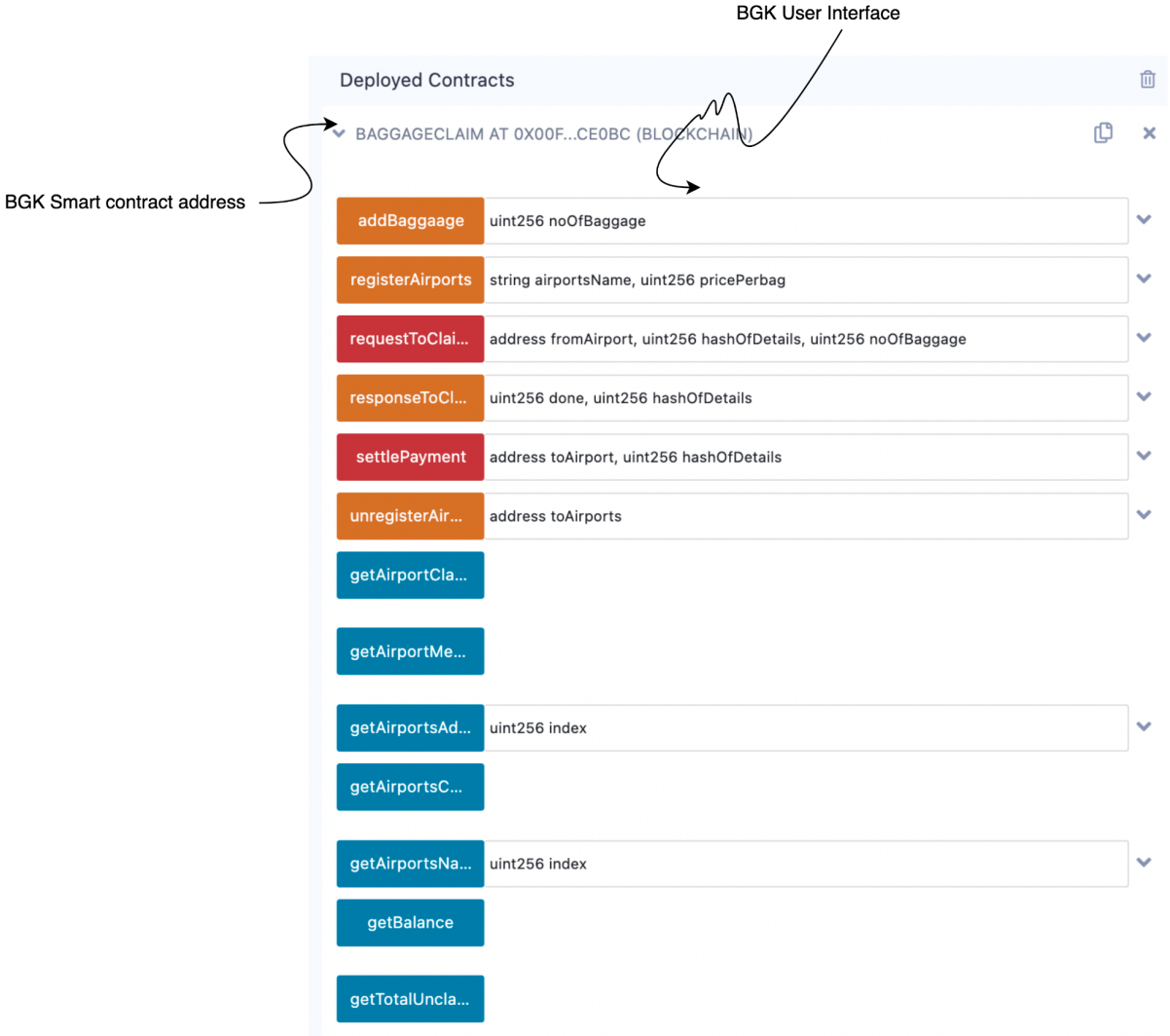
The airportAuthority is a legitimate peer airport, so choose the airportAuthority address in the left panel below the VM simulator, enter a value for escrow of 50 ether, Click Deploy and Run transactions icon, and click the Deploy button at the center of the left panel. The bottom pane shows a deployed smart contract with its address and a down arrow. When you click the down arrow, you expand the web interface to the deployed application, displaying all the public functions and data for you to interact with, and you'll be able to observe the output from the function execution. All these items are shown in figure below In the Remix IDE, you'll observe that the functions of the user interface are color-coded:

- Orange for a public function with no validation rules.
- Red for functions with rules coded by modifiers.
- Blue for access functions that are for viewing any public data. All public functions are available for viewing through blue button

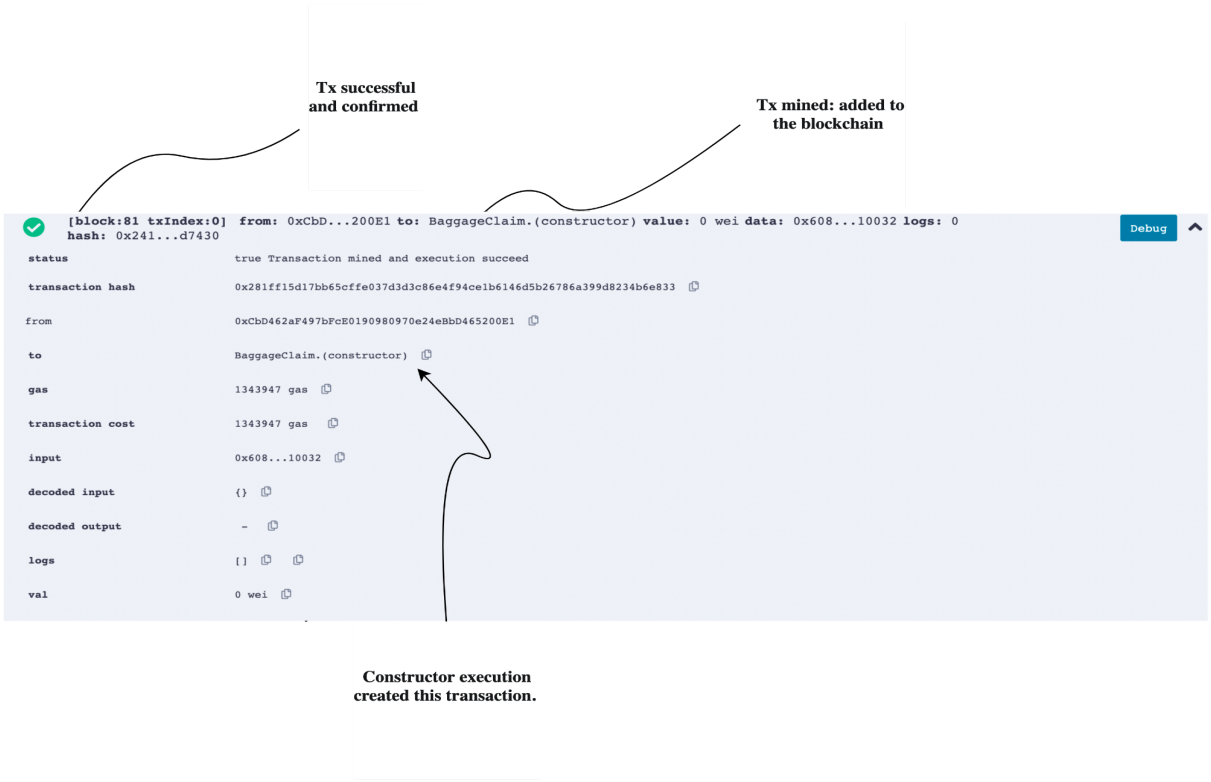
NOTE Please note that the color scheme of the interface keeps changing. The colors may not be the same as the ones displayed here.

The constructor is used for the deployment of the contract when you click the Deploy button. The transaction created is displayed in the console of your Remix IDE, as shown in figure

Now you're ready to test the other functions: registerAirports(),addBaggage(),requestToClaimBaggage(),responseToClaimBaggage(),settlePayment(), and unregisterAirports(). You'll observe that the simulated VM has many accounts for testing purposes, which are listed in the Account drop-down box at the top of the left panel. Some account numbers (five from the bottom) from the list are repeated in table , along with their allocations to the roles you identified: the airportAuthority of the BGK consortium, customer, and airport.



Deployed BaggageClaim smart contract and its UI



Transaction for BaggageClaim constructor (recorded/mined)

TEST PLAN DESCRIPTION

Here's a simple test plan to verify the execution of your functions in the IDE:

- ***constructor()*** *function* —The constructor is executed upon deployment of the contract. Set the Value field in the blockchain simulator panel at the top left to 50 ether, as shown in figure above. The selected account is that of the airport authority . The meaning of this initialization is that the airport authority deposits a value of 50 ether. Now click the Deploy button. You'll see the account balance go down by this amount after the constructor is executed.
- ***registerAirports()*** *function*—Any airport can self-register with a deposit. Make sure that the airport account is selected in the Account drop-down list in the top-left panel. This function needs four parameters: the account address and the escrow value. The airport name and the price per bag The account address is *implicitly provided* by msg.sender. Enter 50 ether for the escrow value; then click the Register button to execute the function.
- ***addBaggage()*** *function* —Make sure that the airport account is selected in the Account box. Then, in the function parameter box, paste in the total number of bags, and supply any number (say, 123) to represent the hash of the off- chain details of data and the total number of bags you want to claim.Click the Request button.
- ***requestToClaimBaggage()*** *function* —Make sure that the customer account is selected in the Account box. Then, in the function parameter box, paste in the fromAirport address, and supply any number (say, 123) to represent the hash of the off- chain details of data and the total number of bags you want to claim.Click the Request button.

- ***responseToClaimedBaggage()*** *function* —Make sure that the airport account is selected in the Account box. In the function parameter box, paste in the fromAirport address, and supply any number (say, 345) to represent the hash of off-chain details of data and a third value to indicate whether the request was accepted (1) or declined (0) (based on the availability of baggage, of course). Click the Response button.
- ***settlePayment()*** *function*—Make sure that the customer account is selected in the Account box. In the function parameter box, paste in the airport address, and specify an amount to be paid (say, 2 ether) for settle- ment. Click the settlePayment button.

- ***balanceDetails()*** *function*—Click the balanceDetails button, with the customer address as a parameter. You see all the details you entered there, but with the escrow reduced by 2 because it paid for bags. If you repeat this process for the airport address, you'll see 2 more ethers in its (airport) escrow, which verifies that all your functions worked as expected. You should see this verification reflected in the balances in the blockchain emulator panel's accounts.
- ***unregisterAirport()*** *function*—Unregistering can be done only by the airport authority,because there may be conditions to be checked before the escrow is returned to the airports.

TEST INSTRUCTIONS

- Following are step-by-step instructions for the test sequence discussed in the previous plan. This list tests all the items of the baggage claim smart contract that you see in the (Remix) UI. The execution details are displayed in the output console. You should be able to see whether the function was executed successfully (a green checkmark on the console) and many other details related to transaction execution and confirmation. Follow these steps to run the tests:
- Restart the Remix IDE. This action resets the blockchain environment to its starting point. You can restart any time you make a mistake during the learning process. Clear the console, using the O symbol in the top-left corner of the console (bottom panel)
- Copy (BaggageClaim.sol) into the editor window. Make sure to use the correct version of the compiler in the pragma line.
- Click Compile in the menu in the top-right corner; then click Deploy.
- Configure the following settings in the blockchain emulator panel and then click Deploy:

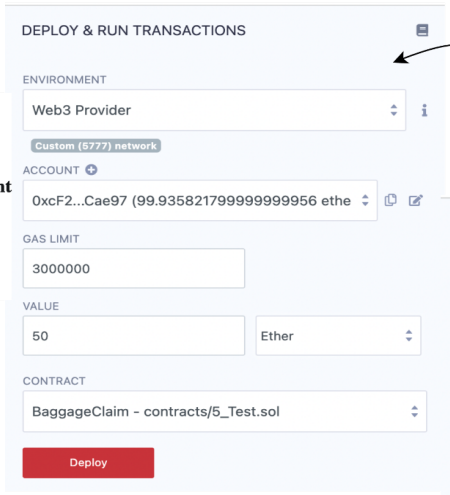
- *Environment* —Web3 Provider

- *Account*—the first address (the airport authority account, such as 0xca3 . . .)

- *Value* —50 ether (not Wei)

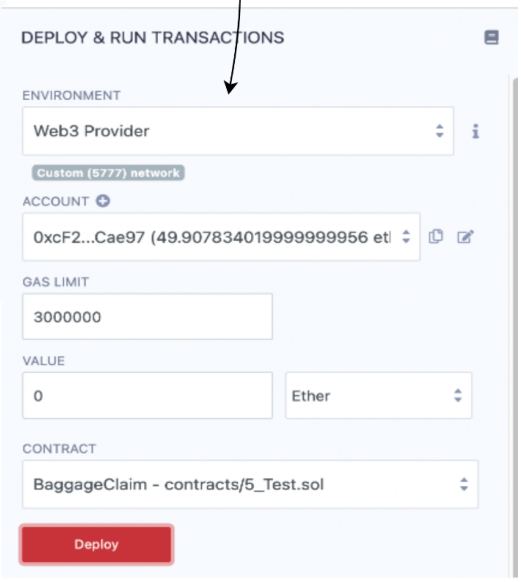
  Refer to the figure for details.
- Open the smart contract by clicking the down arrow next to Deployed Contracts
- (Self-)Register airport
  - Set Account to the airport (address (0x147 . . .).
  - Set Value to 50 ether (not Wei).
  - Select the name of the airport
  - Select price per bag
  - Click Register.
  - Transaction between airport and customer
- With airport A's address (0x147 . . .) selected in the Account box, in the addbaggage() function's parameter box, paste in the bag quantity enter 123 for the hash details and click Request.
- With customer A's address (0x4b0 . . .) selected in the Account box, in the requestToClaimBaggage() function's parameter box, paste in the bag quantity enters 123 for the hash details and the airport from which you want the bag click Request.
- With airports address (0x147 . . .) selected in the Account box, in the responseToBaggageClaim() function's parameter box, enter the status as approve 1 and disapprove 0, enter 123 for the hash details and 1 for success, and then click Response.
- To test the settlePayment() function, with the address of customer in the Account box, enter 2 ether in the Value box, paste the address of airport in the function's parameter box, and click settlePayment.
- Unregister, using airport address (0x147 . . .) as a parameter. The address of the airport authority (0xca3 . . .) must be selected in the Account box; otherwise, it will revert.
- Click balanceDetails, and use customer address (0x147 . . .) as a parameter to see the internal balance of this account. You can also check all the account bal- ances in the top-left panel.

BGK chairperson's account with balance 100 ether



Web3

simulated test environment

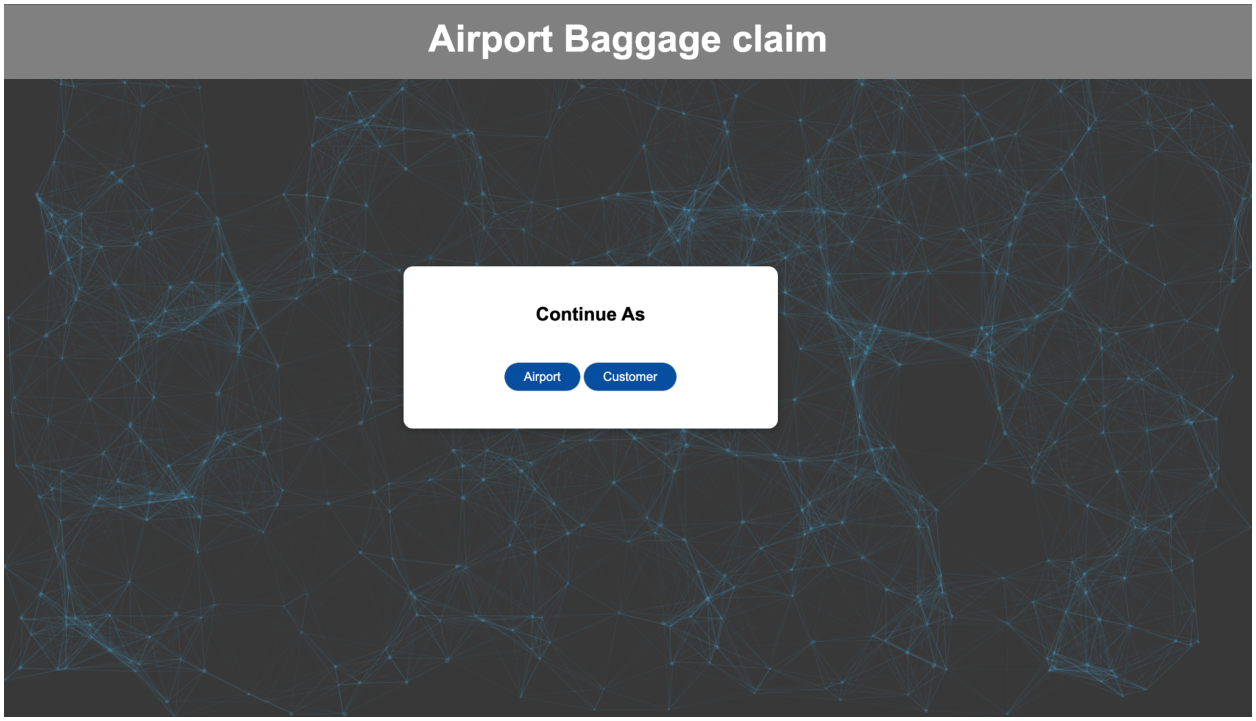


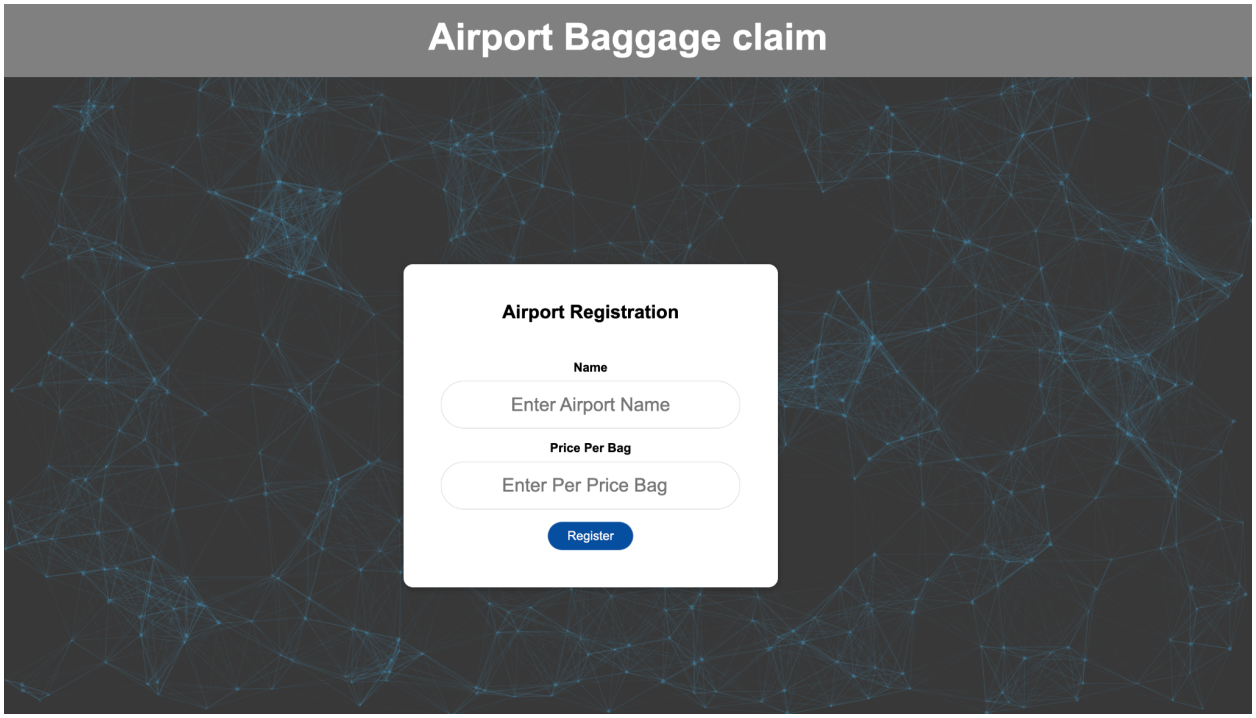
BGK chairperson's account with balance 100 ether

After deploying ASK smart contract

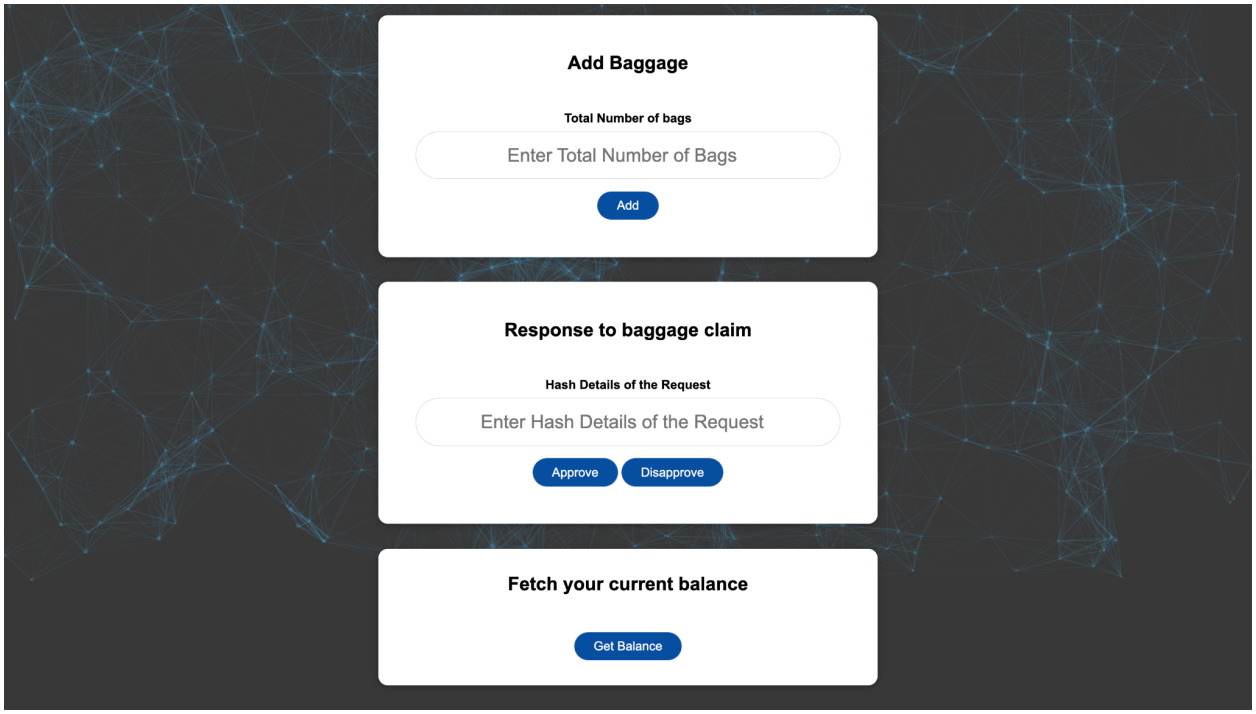
User Interaction Screenshots

User Type Selection Screen

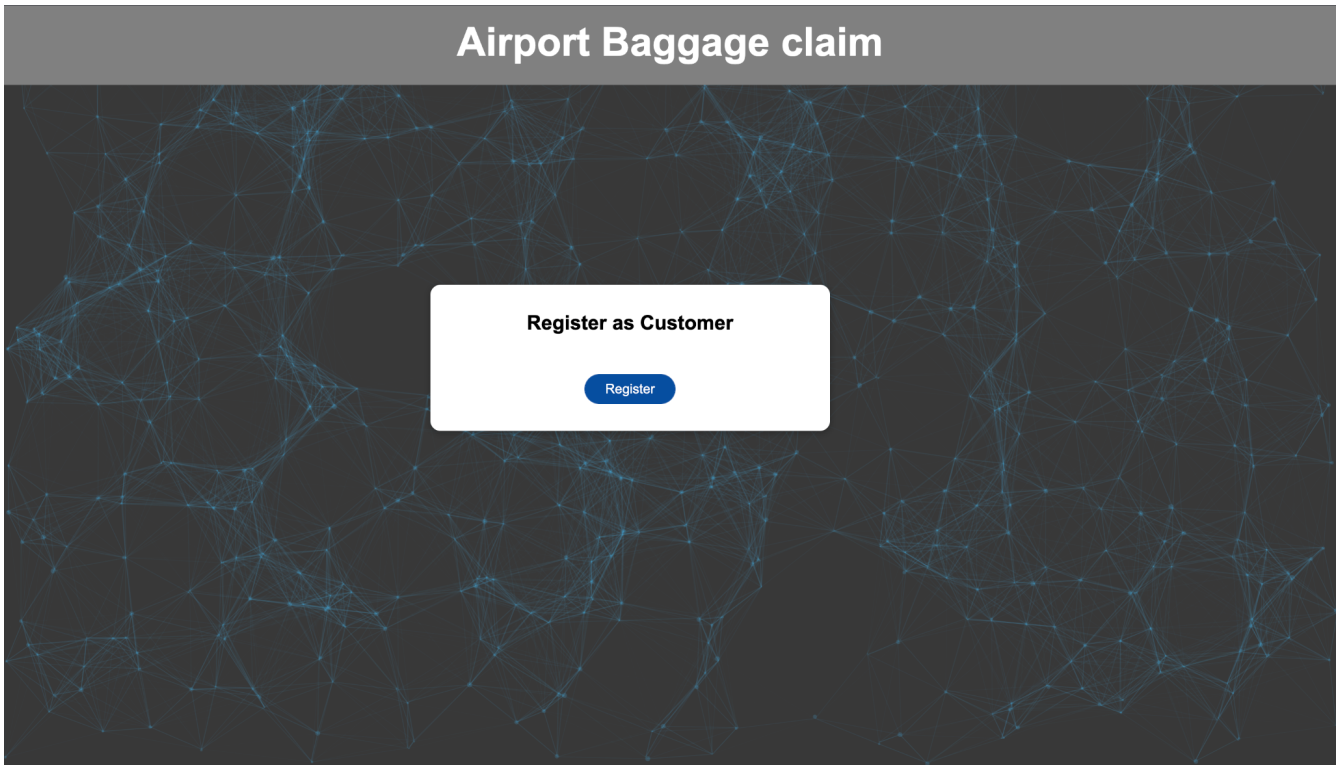




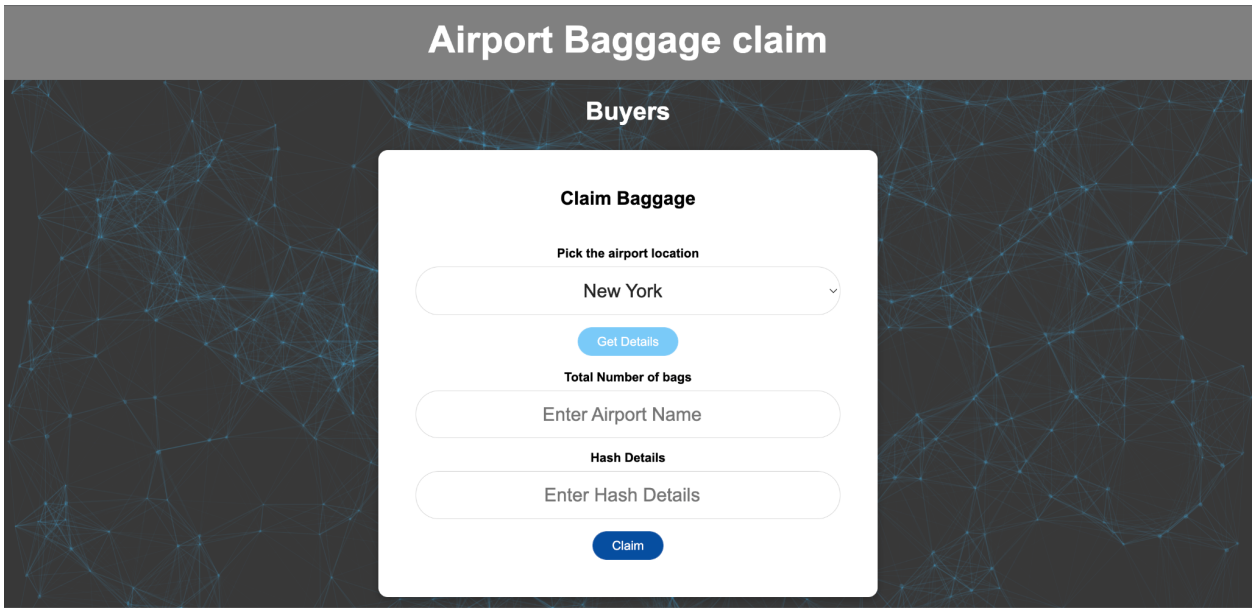
Airport Registration Screen

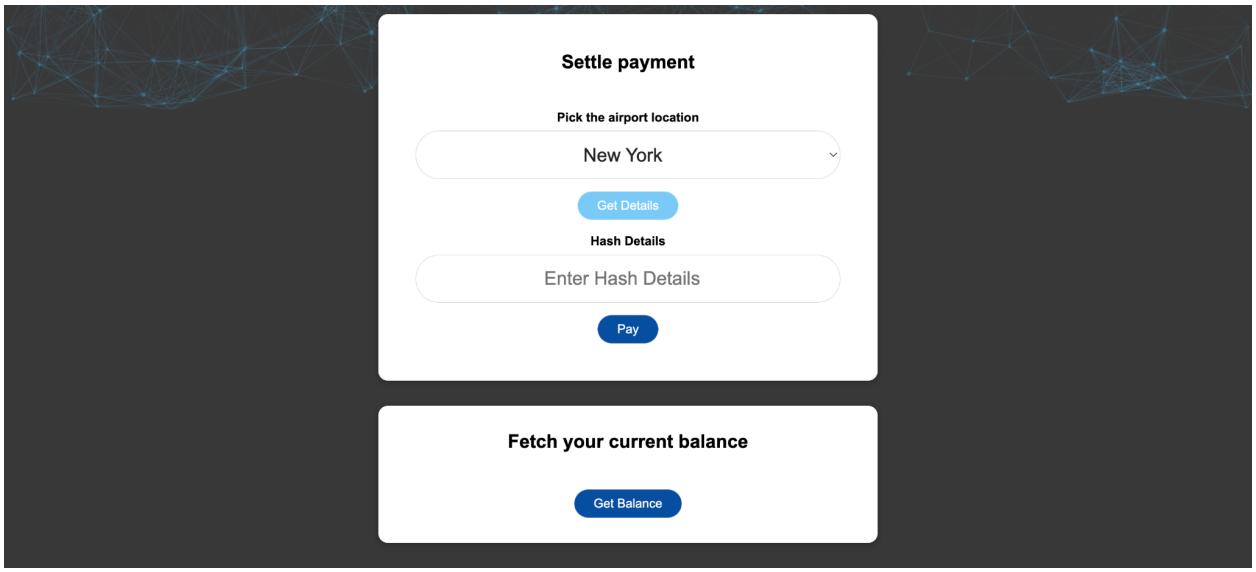


Airport Functions Screen

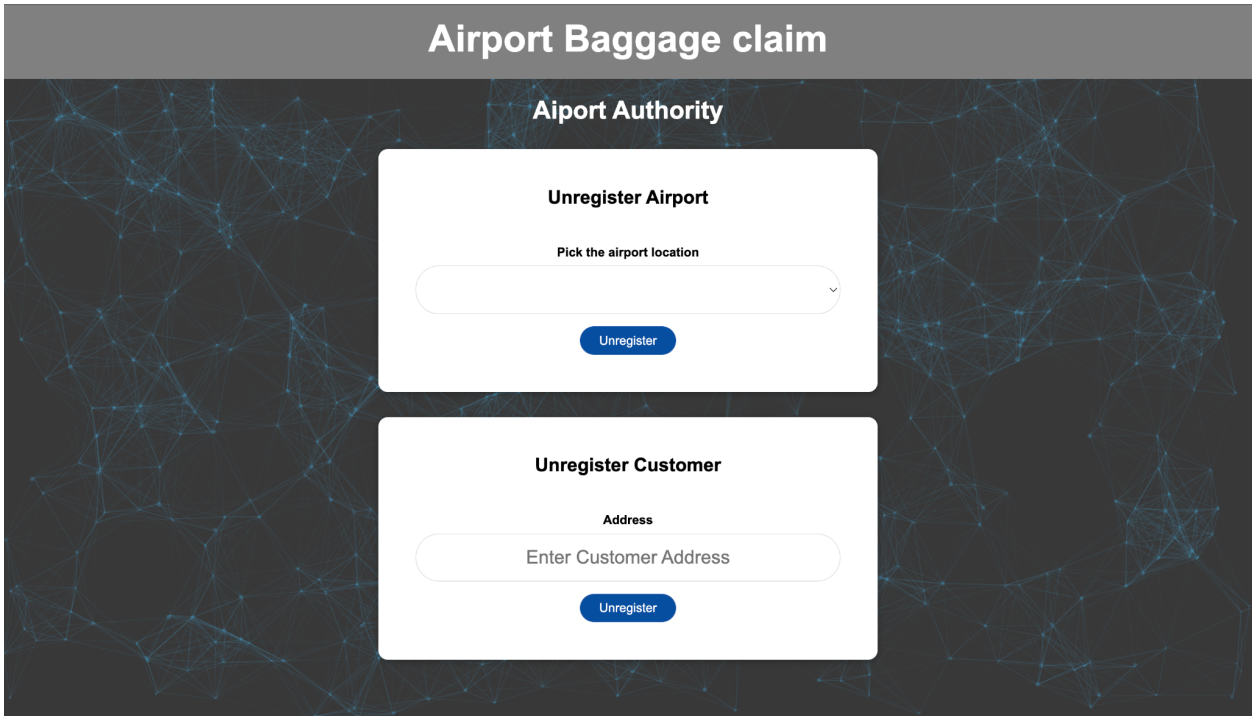


Customer Registration Screen





Customer Functions Screen



Airport Authority Function Screen

Deployment Steps

Following are the steps that you need to perform to deploy the dapp in ropsten network:

- Move to the contract folder, where smart contracts are placed.
- Get the API Key from infura.
- Get the Mnemonics key from MetaMask
- Enter the API keys and the Mnemonic in tuffle-config.js

Run the following commands inside the contracts path:

- npm install --save truffle-hdwallet-provider
- truffle deploy --network ropsten
- truffle console --network ropsten
- BaggageClaim.deployed().then(function(instance){return instance });
- BaggaeClaim.deployed().then(function(instance){return instance.getBalance() });