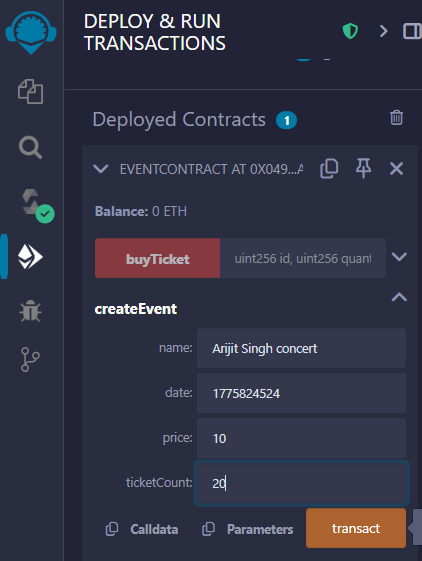
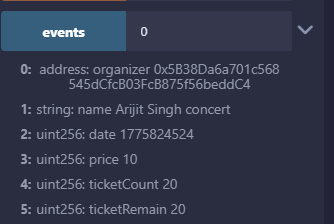
1. In compiler:Compile the event.sol
2. In deploy and run transaction:deploy the event contract
3. Create an event /show and add the event in transaction by clicking transact:
4. Then verify that the event /show is registered or not using,event ,

for index 0 - it will call first registered event details,

for index 1 - it will call second registered event details,

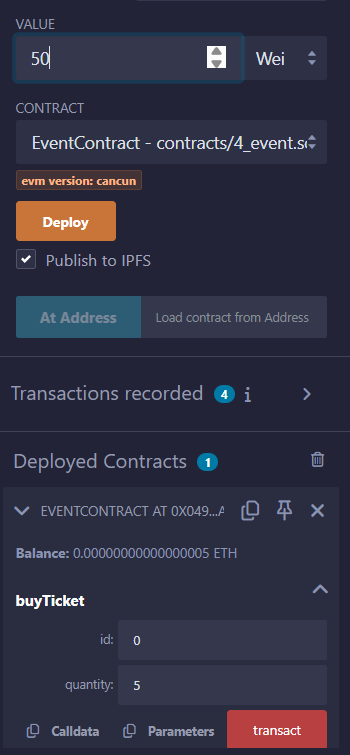
and so on..

e.g.

1. Then user will buy tickets with his address

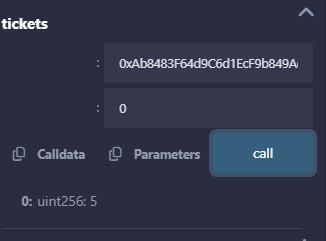
(which will be different from the address of event organizer)

* For that user have to change the address
* Set the id of event ,of which he wants to buy ticket
* Set the quantity of tickets
* Then set the value he is going to pay in value column





1. Now if user have to verify that how many tickets he have for which event that he has registered previously,than he can verify that also



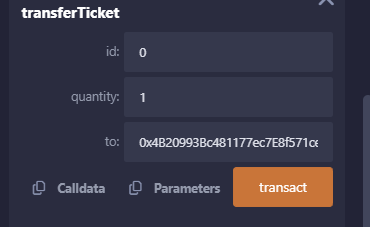
Look below user has currently registered to only index 0 event ,show it will show that 5 unit of ticket he has bought,if the user is registered for another event than it will show that also

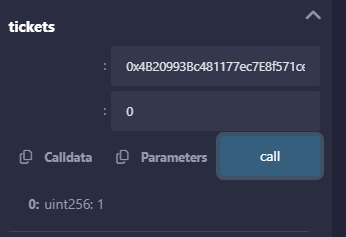
0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

1. Let suppose if the user want to gift his some ticket to his friend than he can do that also using transfer ticket function:

The user :

0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

Who is gifting ticket to his friend :

1. Now to verify that his friend has got tickets or not we will do ticket call

Look his friend has got one ticket!!

1. This is the code:

/\*

\* This smart contract, EventContract, is a blockchain-based solution for managing events and ticket sales.

\* It allows organizers to create events by specifying details such as the event name, date, ticket price,

\* and the number of tickets available. Each event is uniquely identified by an event ID. Users can browse

\* events, purchase tickets, and transfer tickets to others, all while ensuring transparency and security

\* through the blockchain.

\* Key functionalities of the contract include:

\* 1. Creating events: Event organizers can schedule future events and define the number of tickets available for sale.

\* 2. Buying tickets: Users can purchase tickets for available events by sending the exact amount of Ether required for their purchase.

\* 3. Transferring tickets: Ticket holders can transfer their tickets to other users before the event occurs, enabling flexibility.

\* The contract uses mappings to store event details and track the tickets owned by each user for a specific event.

\* It incorporates safeguards, such as verifying event dates, ensuring sufficient Ether for ticket purchases, and

\* preventing transactions for past events. This project demonstrates how Ethereum smart contracts can facilitate

\* secure and decentralized event management, ensuring fairness and transparency for all participants.

\*/

// SPDX-License-Identifier: Unlicense

pragma solidity >=0.5.0 <0.9.0;

contract EventContract {

//structure to represent event

struct Event {

address organizer; // Address of the event organizer

string name; // Name of the event

uint date; // Date of the event as a UNIX timestamp

uint price; // Price of one ticket (in Wei)

uint ticketCount; // Total number of tickets available

uint ticketRemain; // Remaining tickets that can still be sold

}

// Mapping to store events using an event ID (uint) as the key

mapping(uint => Event) public events;

// Nested mapping to track the number of tickets each user owns for a specific event

mapping(address => mapping(uint => uint)) public tickets;

// Counter to assign unique IDs to events

uint public nextId;

/\*\*

\* @dev Create a new event

\* @param name Name of the event

\* @param date Date of the event (must be in the future)

\* @param price Price of one ticket (in Wei)

\* @param ticketCount Total number of tickets to be made available

\*/

function createEvent(string memory name, uint date, uint price, uint ticketCount) external {

// Ensure the event date is in the future

require(date > block.timestamp, "You can organize an event only for a future date");

// Ensure at least one ticket is being created

require(ticketCount > 0, "You must create more than 0 tickets");

// Create and store the event in the mapping

events[nextId] = Event(msg.sender, name, date, price, ticketCount, ticketCount);

// Increment the event ID for the next event

nextId++;

}

/\*\*

\* @dev Buy tickets for an event

\* @param id The ID of the event

\* @param quantity The number of tickets to buy

\*/

function buyTicket(uint id, uint quantity) external payable {

// Check if the event exists

require(events[id].date != 0, "Event does not exist");

// Ensure the event hasn't already occurred

require(events[id].date > block.timestamp, "Event has already occurred");

// Fetch the event from the mapping

Event storage \_event = events[id];

// Ensure the buyer has sent enough Ether to pay for the tickets

require(msg.value == (\_event.price \* quantity), "Insufficient Ether sent");

// Ensure there are enough tickets remaining

require(\_event.ticketRemain >= quantity, "Not enough tickets available");

// Deduct the purchased tickets from the remaining tickets

\_event.ticketRemain -= quantity;

// Record the purchased tickets in the buyer's account

tickets[msg.sender][id] += quantity;

}

/\*\*

\* @dev Transfer tickets to another user

\* @param id The ID of the event

\* @param quantity The number of tickets to transfer

\* @param to The address of the recipient

\*/

function transferTicket(uint id, uint quantity, address to) external {

// Check if the event exists

require(events[id].date != 0, "Event does not exist");

// Ensure the event hasn't already occurred

require(events[id].date > block.timestamp, "Event has already occurred");

// Ensure the sender has enough tickets to transfer

require(tickets[msg.sender][id] >= quantity, "You do not have enough tickets");

// Deduct tickets from the sender's balance

tickets[msg.sender][id] -= quantity;

// Add tickets to the recipient's balance

tickets[to][id] += quantity;

}

}