ProteaV2Contracts Event Manager Contract

LinumLabs is a team of blockchain developers who provide blockchain solutions as form of service.

This is one of the contracts created by the LinumLabs in their ProteaV2Contracts. As the name of the contract signifies, the contract is actually a contract that manages different events and keeps record of the participant.

Imports

There are four imported contracts in the contracts, two of which the contract inherited. The following are the imported contracts :

```
import "./zeppelin-solidity/SafeMath.sol";
```

1. Zeppelin SafeMath contract : This is a library contract that is imported for the use of arithmetic operations. The following are the functions in the library and the use:
   - tryAdd(uint256 a, uint256 b) : this function returns the addition of two unsigned integers, with an overflow flag.
   - trySub(uint256 a, uint256 b): this function returns the subtraction of two unsigned integers, with an overflow flag.
   - tryMul(uint256 a, uint256 b): this function returns the multiplication of two unsigned integers, with an overflow flag.
   - tryDiv(uint256 a, uint256 b): this function returns the division of two unsigned integers, with a division by zero flag.
   - tryMod(uint256 a, uint256 b): this function returns the remainder of dividing two unsigned integers, with a division by zero flag.
   - add(uint256 a, uint256 b): this function returns the addition of two unsigned integers, reverting on overflow
   - sub(uint256 a, uint256 b): this function returns the subtraction of two unsigned integers, reverting on overflow (when the result is negative).
   - mul(uint256 a, uint256 b): this function Returns the multiplication of two unsigned integers, reverting on overflow.
   - div(uint256 a, uint256 b):this function returns the integer division of two unsigned integers, reverting on division by zero. The result is rounded towards zero.
   - mod(uint256 a, uint256 b):this function returns the remainder of dividing two unsigned integers. (unsigned integer modulo), reverting when dividing by zero.

```
import "./RewardIssuerInterface.sol";
```

2. Rewarder Issuer Interface: This is an interface the contract imported and inherited. This means the contract have to implement all the functions in the interface. This Rewarder Interface contract contains only one function which is the payout function.

- payout(address _member, uint256 _index): This is a function that pays out member based on their index.

```
import "./ERC223/ERC223Receiver.sol";
```

3. ERC233 Receiver: This contract is an interface imported and inherited by the **EventManager** contract. This interface contains just one function named **tokenFallback**.
    - tokenFallback(address _from, uint _value, bytes _data): this function takes in three parameters which includes the address, value and the data. This function contains the data that activates the fallback function.

```
import "./zeppelin-solidity/ERC20/ERC20Basic.sol";
```

4. ERC20Basic: This is a contract that acts as a simpler version of ERC20 interface. It includes about four functions from the ERC20 contract. The following are the functions in the ERC20 basic contract:
    - totalSupply(): This function returns the total supply of the ERC20 token.
    - balanceOf(address who): this function takes in an address and returns the token balance of the address.
    - transfer(address to, uint256 value): this function takes in two parameters which are the address of a recipient and the value of tokens to be sent to the recipient, and it is used to transfer a token from the current account to another account.
    - event Transfer(address indexed from, address indexed to, uint256 value) : This is an event also in the contract and the event basically emits out an event when a transfer function it called.

EventManager Contract

The **EventManager** is the name of the contract itself. The **EventManager** contact inherits two contracts namely; **RewardIssuerInterface** and the ERC223Receiver interface. This means the contract implemented all of the functions in the interfaces that was inherited.

State Variables

There are about 5 state variables in the contract. The following would highlight each state variable and what they stand for:

```
address public tokenManager;
address public rewardManager;

uint256 public creationCost = 20; // Defaulting for demo
uint256 public maxAttendanceBonus = 5;
```

tokenManager: This is the variable where the ERC20 token contract is stored .

rewardManager: This is the variable where the **rewardManager** address is stored, the value is defined inside the function **initContract**.

creationCost: A default value of 20 was given as the creation cost variable, but the value can be redefined inside the **initContract** function. This is basically the variable that stores the cost of an event and this is the amount that would be forwarded or transferred to the events account when an event is created.

maxAttendanceBonus: This is maximum attendance bonus as the name signifies is the variable that stores the value of the attendance bonus. The **maxAttendanceBonus** variable is used to calculate the Payout for the reward in an event.

```
address public admin;
```

admin: This is the address of the admin, this address is used inside the modifier onlyAdmin. The onlyAdmin modifier was used in the initcontract, this implies that only the admin address can call the initcontract function.

Mappings

```
mapping(uint256 => mapping(address => uint8)) public memberState;
```

memberState mapping: This is a 2D mapping that stores the attendance of each participant and he is rewarded when his attendance reaches 99. Each event is mapped to each participant and the participants are mapped to the amount of attendance they have.

```
mapping(uint256 => EventData) public events;
```

events mapping: This is a mapping data structure that stores the struct containing each event data. Each event is signified by an ID, and this ID points to the data location of the event Data in the mapping.

Events

This contract contains four events namely: EventCreated, EventConcluded, MemberRegistered, MemberAttended.

```
event EventCreated(uint256 indexed index, address publisher);
event EventConcluded(uint256 indexed index, address publisher, uint8 state);

event MemberRegistered(uint256 indexed index, address member);
event MemberAttended(uint256 indexed index, address member);
```

The following are the events emitted in the contract:

EventCreated: This event is used to emit newly created events in the contract. It takes in the index of the event and the organizer of the event created by the createEvent function.

EventConcluded: This event listener emits the index of an already created event, the organizer of the event and the current state of the event. This event is emitted when the endEvent function is called and also when the cancelEvent function is called. The basic highlight of the event entails that, it is triggered when an Event is canceled or ended.

MemberRegistered: This event takes in two parameters, the index of the member been registered and the address of the member that is registered for a particular event. The MemberRegistered event is emitted when the rsvp or the rsvpDebug function is called.

MemberAttended: This event takes in two parameters which includes, the index of a participant and the address of a participant. But this event was never used in the contract.

Struct

```
struct EventData{
    string name;
    string date;
    string location;
    uint24 participantLimit;
    uint8 state;
    uint24 registered;
    uint24 attended;
    address organiser;
    // 0: Not created
    // 1: Created
    // 2: Concluded
    // 3: Cancelled
    uint256 requiredStake;
    uint256 totalStaked;
    uint256 payout;
}
```

The above shows the struct which is one of the data structure used in the contract. The EventData struct is the reference type that gives the layout of the variables inside the Event Data struct. The value types contained in the struct includes, the name of the event, the date of the event, the location of the event, the participantLimit, the state which signifies if the event is on, ended of canceled.

Constructor

The constructor basically runs the moment the contract is deployed. It sets the address of the admin as the msg.sender.

Modifier

There are five modofiers in the contact, the modifiers restrict functions from certain conditions. The folowing outlines the modifiers in the contract and how it is used:

```
modifier onlyAdmin() {
    require(msg.sender == admin, "Not authoriesd");
    _;
}
```

onlyAdmin() : This is a modifier that gives the condition to allow only the msg.sender who is the address admin to call the initContract function.

```
modifier onlyMember(address _member, uint256 _index){
    require(memberState[_index][_member] >= 1, "User not registered");
    _;
}
```

OnlyMember(address _member, uint256 _index) : this modifier requires that only the members who have been registered for a particular event can call the function. The onlyMember modifier takes in two parameters, _member and _index.

```
modifier onlyToken(){
    require(msg.sender == address(tokenManager), "Not registered token address");
    _;
}
```

OnlyToken(): this modifier requires that only the tokenManager contract address is the msg.sender of the function otherwise it reverts with the message “Not registered token address”. This modifier was used in the createEventDebug function which signifies that only the tokenManager contract can call the function.

```
modifier onlyRewardManager() {
    require(msg.sender == rewardManager, "Only reward manager authorised");
    _;
}
```

onlyRewardManager(): This modifier signifies that only the rewardManager addesss can call the function where it is been used. It is used in the payout function. This implies that only the rewardMananger can payout.

```
modifier onlyManager(uint256 _index){
    require(events[_index].organiser == msg.sender, "Account not organiser");
    _;
}
```

onlyManager(): This is a modifier that requires that the account calling the function is the organizer of the event address (EOA). Otherwise, it reverts if the msg.sender calling the function isn't the organizer of the event. This modifier was used in three functions, namely: increaseParticipantLimit, endEvent and lastly the cancelEvent.

Functions

```
function initContract(address _tokenManager, address _rewardManager, uint256 _creationCost, uint256 _maxAttendanceBonus) public onlyAdmin {
    tokenManager = _tokenManager;
    rewardManager = _rewardManager;
    creationCost = _creationCost;
    maxAttendanceBonus = _maxAttendanceBonus;
}
```

initContract: This function sets the four state variables which are tokenManager, rewardManager, creationCost, maxAttendanceBonus. Only the admin of the contract can call this function to set how much each event created on the contract is paid. It also sets the reward manager for the contract.

```
function updateStakes(uint256 _creationCost, uint256 _maxAttendanceBonus) public onlyAdmin{
    creationCost = _creationCost;
    maxAttendanceBonus = _maxAttendanceBonus;
}
```

updateStakes: This function basically updates the creation cost for the event and the maximum attendance bonus to be paid out.

```
function createEvent(
    string _name, string _date, string _location,
    uint24 _participantLimit, address _organiser, uint256 _requiredStake)
    private {
    require(forwardStake(creationCost, _organiser), "Must forward funds to the reward manager");
    numberOfEvents += 1;
    events[numberOfEvents].name = _name;
    events[numberOfEvents].date = _date;
    events[numberOfEvents].location = _location;
    events[numberOfEvents].participantLimit = _participantLimit;
    events[numberOfEvents].organiser = _organiser;
    events[numberOfEvents].requiredStake = _requiredStake;
    emit EventCreated(numberOfEvents, _organiser);
}
```

createEvent: The create event is responsible for creating an event and it saves all created events in a mapping of index to the event Data which is a struct. To call this function, a require condition is used whereby the function reverts if the forwardstake function when called is false. This implies that funds must be forwarded to the reward manager before an event is created. It also increases the index of the events created every time the function is called by 1.

The EventCreated event listener also emits the index of the event created and the address of the organize that created the event.

Note: This function was made private and was never used inside the contract. Reasons for this remains unknown.

```
function getEvent(uint256 _index) public view returns(string, string, string, uint24, uint8, uint24, uint256){
    return (events[_index].name, events[_index].date, events[_index].location, events[_index].participantLimit,
        events[_index].state, events[_index].registered, events[_index].requiredStake);
}
```

getEvent: The get event function is a view functions that returns the variables inside the struct of a particular index of the events mapping.

```
function getEventStake(uint256 _index) public view returns(uint256){
    return events[_index].requiredStake;
}
```

getEventStake : This is a view function that returns the required amount staked for a particular event in the mapping.

```
function createEventDebug(
    string _name, string _date, string _location,
    uint24 _participantLimit, address _organiser, uint256 _requiredStake)
    external onlyToken {
    numberOfEvents += 1;
    events[numberOfEvents].name = _name;
    events[numberOfEvents].date = _date;
    events[numberOfEvents].location = _location;
    events[numberOfEvents].participantLimit = _participantLimit;
    events[numberOfEvents].organiser = _organiser;
    events[numberOfEvents].requiredStake = _requiredStake;
    emit EventCreated(numberOfEvents, _organiser);
}
```

createEventDebug: This functions serves the same function as the createEvent function but in this case, the function isn't made private instead it was made external, also an onlyToken modifier was used to ensure that the only msg.sender that can call that function is the token contract address registered in the state variable.

```
function increaseParticipantLimit(uint256 _index, uint24 _limit) public onlyManager(_index){
    require(events[_index].participantLimit < _limit, "Limit can only be increased");
    events[_index].participantLimit = _limit;
}
```

increaseParticipantLimit: This function allows only the manager of that particular event increase the participant limit that was initially set when the even was created. This function takes a require statement which acts as a guard from reducing the participants in an event, meaning the participants can only be increased.

```
function endEvent(uint256 _index) public onlyManager(_index){
    require(events[_index].state == 1, "Event not active");
    events[_index].state = 2;
    calcPayout(_index);
    emit EventConcluded(_index, msg.sender, events[_index].state);
}
```

endEvent: This function is responsible for closing an event started. Only the creator of the event who is the organizer in the struct can call it. It uses a require statement that ensures that the event is still on

before ending it. The state of the event is between 1,2,3. 1 means the event is in progress, 2 means the event ended already, 3 means the event was cancelled. In this case, the require statement ensures that the event state is in 1 which means the event must be in progress before it can be ended, which hen sets it to 2. It then calls the calcPayout function that calculates the reward. Lastly, the EventConcluded event is emited.

```
function cancelEvent(uint256 _index) public onlyManager(_index){
    require(events[_index].state == 1, "Event not active");
    events[_index].state = 3;
    emit EventConcluded(_index, msg.sender, events[_index].state);
}
```

cancelEvent: Just like the endEvent function, the cancelEvent function also ends an event but it doesn't calculate any payout. Instead it sets the state of that particular event to 3. Then emits the EventConcluded event listener. It can also only be called by the organizer of that particular event.

```
 function forwardStake(uint256 _amount, address _member) internal returns(bool) {
     ERC20Basic token = ERC20Basic(tokenManager);
     token.transfer(_member, _amount);
     return true;
 }
```

forwardStake: This is basically in internal function that takes in the amount and the address of the an EOA or contract and uses the ERC20Basic interface to call the transfer function to the contract or the EOA, then it returns true.

```
function rsvp(uint256 _index, address _member) private {
    require(forwardStake(events[_index].requiredStake, _member), "Must forward funds to the reward manager");
    /// Send stake to reward manager
    /// Updated state
    memberState[_index][_member] = 1;
    events[_index].totalStaked.add(events[_index].requiredStake);
    events[_index].registered += 1;
    emit MemberRegistered(_index, _member);
}
```

Rsvp: this is a private function that requires that a member that joins an event forwards the required stake to the organizer. Then the function uses safeMath function **add()** to add new stakes to the total Stakes staked. It then registers and increases the total registered members by 1. Lastly, emits the MemberRegistered event.

Note: This function was made private with no reasons given as it wasn't used in the contract. The function is deemed redundant.

```
function rsvpDebug(uint256 _index, address _member) external onlyToken {
    /// Send stake to reward manager
    /// Updated state
    memberState[_index][_member] = 1;
    events[_index].totalStaked.add(events[_index].requiredStake);
    events[_index].registered += 1;
    emit MemberRegistered(_index, _member);
}
```

rsvpDebug: This function performs same function as the rsvp function but in this case it is made external and an onlyToken modifier was used. Also the require statement was removed compared to the rsvp function.

```
function confirmAttendance(uint256 _index) public {
    require(memberState[_index][msg.sender] == 1, "User not registered");
    memberState[_index][msg.sender] = 99;
}
```

confirmAttendance: This function checks if the user attended the event. If not , it reverts with the error message" User not registered".

```
function calcPayout(uint256 _index) internal {
    uint256 reward = events[_index].totalStaked / events[_index].attended;
    events[_index].payout = reward.add(maxAttendanceBonus.mul(events[_index].attended).div(events[_index].participantLimit));
}
```

calcPayout: this is a function that performs an arithmetic of calculating the total staked token and divides it by the total number of members that attended that event, then stores it in the variable called reward. This reward then perform another calculation by using safemath functions to multiplty the attendance bonus by the number of attendees then divides it by the participant limit of that event, then sums the result up by adding the reward. The result of all calculations is then stored in the payout of that particular event.

```
function payout(address _member, uint256 _index) external view onlyRewardManager onlyMember(_member, _index) returns(uint256){
    require(memberState[_index][_member] == 99, "User not attended");
    require(events[_index].state > 1, "Event not ended");
    if(events[_index].state == 3){
        return events[_index].requiredStake;
    } else if(events[_index].state == 3) {
        return events[_index].payout;
    }
}
```

payout: This function goes through different conditional statements starting from requiring that the a particular member attended a particular event and it reverts if the user never attended that event. Another require condition spurns up if the first was passed, and this requires that the event has ended by making sure that the state of the event is not 1 but at least greater than 1. Although the IF block that comes after the last require statement remains redundant as the condition stated also reflects same condition in the elseif block .

Recommendation Note: This contract contains a lot of bugs and redundant codes.