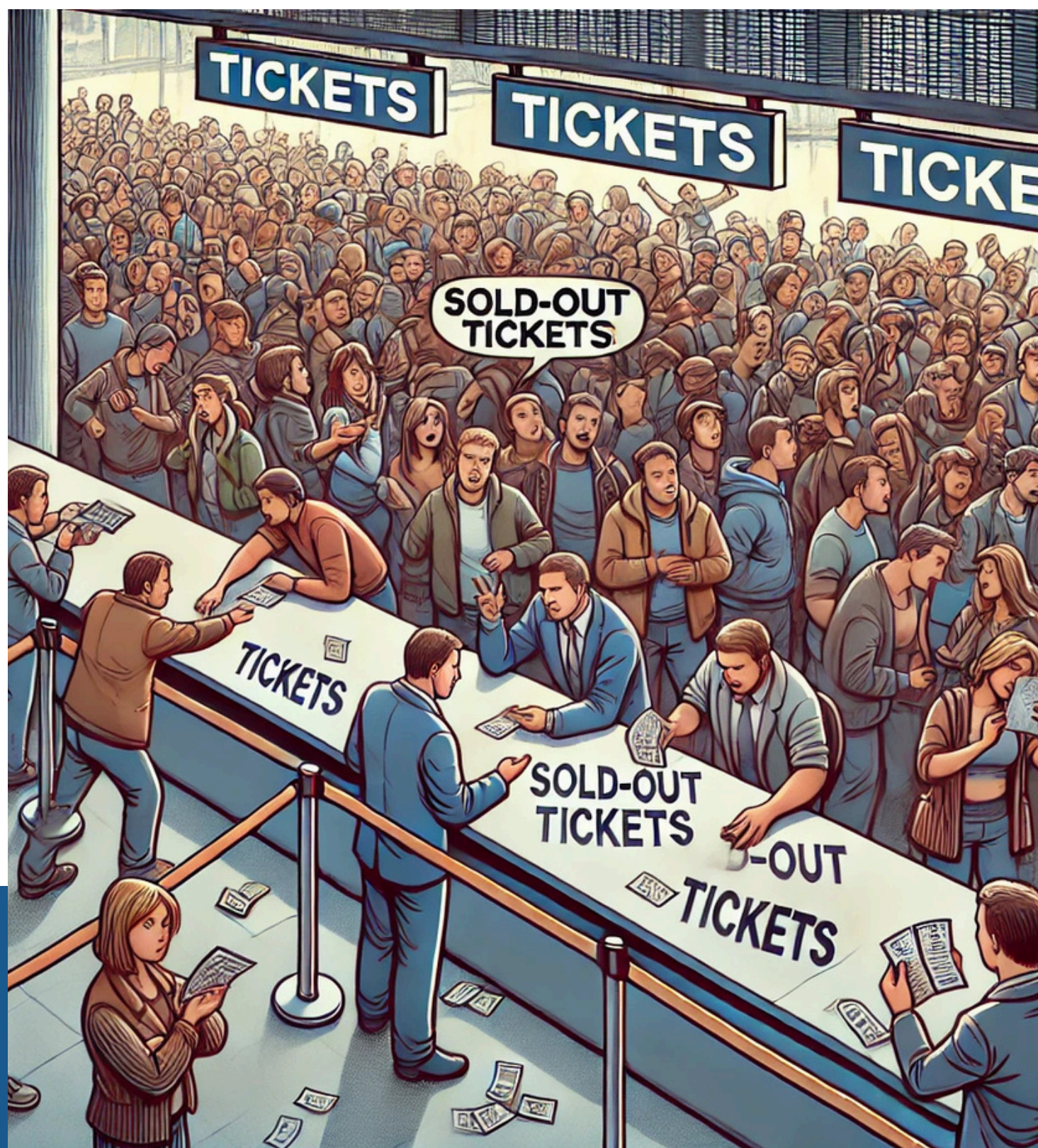


Tickechain

Un sistema decentralizzato
per la gestione di eventi e
biglietti NFT



Ticketing Tradizionale

Il ticketing tradizionale è soggetto a falsificazioni, scalping e prezzi gonfiati, con alte commissioni e poca trasparenza, penalizzando utenti e organizzatori.

- ✗ Biglietti non verificabili e rischio di truffe per gli acquirenti.
- ✗ Piattaforme chiuse che limitano il controllo degli utenti sugli acquisti.

Obiettivi

TickeChain rivoluziona il ticketing con un sistema **trasparente, sicuro e decentralizzato**, eliminando le criticità del mercato tradizionale. Grazie alla blockchain, migliora l'affidabilità degli eventi, riducendo il rischio di frodi e garantendo un'esperienza più efficiente.

Autenticità e Sicurezza

Ogni biglietto è un NFT unico su blockchain, risultando così impossibile da falsificare o duplicare.

Eliminazione Intermediari

Gli utenti possono acquistare biglietti senza piattaforme centralizzate, riducendo costi e commissioni.

Rimborsi Efficienti

Gli utenti possono richiedere rimborsi verificati in caso di eventi annullati, senza rischi di abuso.

Verifica Affidabile

I biglietti sono validati tramite QR code e blockchain, così da garantire autenticità e sicurezza agli ingressi.

Architettura Generale del Sistema

Blockchain e Decentralizzazione

Tutte le operazioni sono registrate su **Ethereum**, eliminando intermediari e prevenendo manomissioni.

Smart Contract Sicuri

Gli smart contract in **Solidity** gestiscono eventi, biglietti e pagamenti, garantendo sicurezza e trasparenza su blockchain.

Integrazione con Wallet

Gli utenti si connettono al sistema tramite **MetaMask**, permettendo acquisti e verifiche direttamente dalla loro identità blockchain.

Frontend Interattivo

Il sistema utilizza **React + Vite** per offrire un'interfaccia veloce e intuitiva, con gestione delle transazioni tramite MetaMask.

Blockchain e Decentralizzazione

La blockchain è una tecnologia che consente di registrare transazioni in modo **immutabile**, **trasparente** e **decentralizzato**, eliminando la necessità di intermediari. Ogni operazione viene verificata da una rete distribuita di nodi, rendendo i dati incorruttibili e accessibili a tutti. Questo sistema garantisce **sicurezza** e **fiducia**, riducendo il rischio di frodi e manipolazioni.

TickeChain applica questi principi al settore del ticketing, sfruttando la blockchain di Ethereum per registrare ogni biglietto come **NFT** unico. Questo assicura che i biglietti non possano essere falsificati, duplicati o modificati, mentre le transazioni rimangono pubbliche e verificabili da chiunque. Inoltre, grazie alla decentralizzazione, nessuna entità centrale può controllare o limitare l'accesso ai biglietti, offrendo agli utenti piena proprietà e libertà di scambio.

Panoramica degli Smart Contract

Gli smart contract sono programmi auto-eseguibili registrati sulla blockchain, progettati per garantire sicurezza, trasparenza e automazione nelle transazioni. Una volta implementati, funzionano senza bisogno di intermediari, eseguendo le operazioni solo se vengono soddisfatte condizioni prestabilite. Questo riduce il rischio di frodi, errori e manipolazioni, rendendo il sistema affidabile e verificabile da chiunque.

In **TickeChain**, gli smart contract gestiscono l'intero ecosistema del ticketing in modo sicuro e decentralizzato. Ogni componente chiave è regolato da un contratto specifico:

- **EventFactory.sol** per la creazione e gestione dei singoli eventi.
- **EventRegistry.sol** per l'archiviazione e la consultazione degli eventi registrati.
- **PaymentManager.sol** per la gestione dei pagamenti e rimborsi in maniera protetta.
- **TicketManager.sol** per la creazione, validazione e rimborso dei biglietti NFT.

Grazie a questi contratti, TickeChain garantisce un ecosistema trasparente, in cui ogni operazione è verificabile on-chain, eliminando la necessità di fidarsi di piattaforme centralizzate.

EventFactory.sol

Gestisce la creazione, modifica e gestione degli eventi, regolando il loro stato tramite una State Machine e applicando meccanismi di sicurezza come Emergency Stop.

createEvent(string _name, string _location, uint256 _date, uint256 _price, uint256 _totalTickets): Crea un nuovo evento registrandone i dettagli.

updateEvent(uint256 _eventId, string _name, string _location, uint256 _date, uint256 _price): Permette all'organizzatore di modificare un evento prima che sia attivo.

changeEventState(uint256 _eventId, uint8 _newState): Modifica lo stato di un evento [CREATED → OPEN → CLOSED → CANCELLED].

cancelEvent(uint256 _eventId): Annulla un evento e avvia il processo di rimborso.

```
/**
 * @notice Crea un nuovo evento con i dettagli forniti.
 * @dev L'evento viene registrato sulla blockchain e inizialmente impostato nello stato 'CREATED'.
 * @dev La funzione è protetta dal modifier 'whenNotPaused', quindi non può essere eseguita se il contratto è in pausa.
 * @param _name Nome dell'evento.
 * @param _location Luogo dell'evento.
 * @param _description Descrizione dell'evento.
 * @param _date Data dell'evento in formato timestamp UNIX (deve essere futura).
 * @param _price Prezzo del biglietto in token ERC-20.
 * @param _ticketsAvailable Numero totale di biglietti disponibili per l'evento.
 */
function createEvent(
    string memory _name,
    string memory _location,
    string memory _description,
    uint256 _date,
    uint256 _price,
    uint256 _ticketsAvailable
) external whenNotPaused {
    // Controllo che la data dell'evento sia nel futuro
    require(_date > block.timestamp, "La data dell'evento deve essere nel futuro");

    // Verifica che ci siano biglietti disponibili
    require(_ticketsAvailable > 0, "Il numero di biglietti deve essere maggiore di zero");

    // Assegna un nuovo ID univoco all'evento
    uint256 eventId = eventIdCounter;

    // Memorizza i dettagli dell'evento nella mappatura
    events[eventId] = Event({
        name: _name,
        location: _location,
        description: _description,
        date: _date,
        price: _price,
        ticketsAvailable: _ticketsAvailable,
        creator: msg.sender, // L'utente che chiama la funzione diventa il creatore dell'evento
        state: EventState.CREATED // Stato iniziale dell'evento
    });

    // Incrementa il contatore per il prossimo evento
    eventIdCounter++;

    // Emette un evento sulla blockchain per segnalare la creazione dell'evento
    emit EventCreated(eventId, _name, msg.sender);
}

/**
 * @notice Modifica i dettagli di un evento già creato.
 * @dev Solo il creatore dell'evento può modificare i dettagli.
 * @dev L'evento deve essere ancora nello stato 'CREATED' per poter essere modificato.
 * @dev La funzione è protetta dal modifier 'whenNotPaused', quindi non può essere eseguita se il contratto è in pausa.
 * @param _eventId Identificativo univoco dell'evento da modificare.
 * @param _name Nuovo nome dell'evento.
 * @param _location Nuova posizione dell'evento.
 * @param _date Nuova data dell'evento (deve essere futura).
 * @param _price Nuovo prezzo dei biglietti.
 * @param _ticketsAvailable Nuova quantità di biglietti disponibili.
 */
function updateEvent(
    uint256 _eventId,
    string memory _name,
    string memory _location,
    uint256 _date,
    uint256 _price,
    uint256 _ticketsAvailable
) external onlyEventCreator(_eventId) whenNotPaused {
    // Recupera i dati dell'evento dalla mappatura
    Event storage eventToUpdate = events[_eventId];

    // Verifica che l'evento sia ancora nello stato CREATED (non ancora aperto per la vendita)
    require(eventToUpdate.state == EventState.CREATED, "L'evento deve essere nello stato CREATED");

    // Verifica che la nuova data sia nel futuro
    require(_date > block.timestamp, "La data dell'evento deve essere nel futuro");
}
```


EventRegistry.sol

Registra e organizza gli eventi, permettendo di elencarli e filtrarli per creatore, garantendo un accesso strutturato e decentralizzato ai dati.

registerEvent(string _name, string _location, uint256 _date): Aggiunge un nuovo evento al registro.

listEvents(): Restituisce l'elenco di tutti gli eventi registrati.

findEventsByCreator(address _creator): Filtra gli eventi creati da un utente specifico.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

// Importazione di contratti OpenZeppelin per sicurezza e gestione degli accessi
You, settimana scorsa | 1 author (You)
import "@openzeppelin/contracts/security/Pausable.sol"; // Permette di sospendere il contratto
You, settimana scorsa | 1 author (You)
import "@openzeppelin/contracts/access/Ownable.sol"; // Definisce un proprietario del contratto

/**
 * @title EventRegistry
 * @dev Contratto che gestisce un registro decentralizzato di eventi.
 * @dev Supporta la registrazione, visualizzazione, ricerca e cancellazione di eventi.
 */
You, settimana scorsa | 1 author (You)
contract EventRegistry is Pausable, Ownable {

    /// @notice Struttura dati per memorizzare le informazioni di un evento
    You, settimana scorsa | 1 author (You)
    struct Event {
        string name; // Nome dell'evento
        string location; // Luogo dell'evento
        uint256 date; // Data dell'evento in formato timestamp UNIX
        address creator; // Indirizzo del creatore dell'evento
    }

    /// @notice Array dinamico che memorizza tutti gli eventi registrati
    Event[] private events;

    /// @notice Evento emesso quando un nuovo evento viene registrato
    /// @param eventId Identificativo univoco dell'evento
    /// @param name Nome dell'evento
    /// @param creator Indirizzo del creatore dell'evento
    event EventRegistered(uint256 indexed eventId, string name, address indexed creator);

    /// @notice Evento emesso quando un evento viene aggiornato
    /// @param eventId Identificativo univoco dell'evento
    /// @param name Nuovo nome dell'evento
    event EventUpdated(uint256 indexed eventId, string name);

    /// @notice Evento emesso quando un evento viene eliminato
    /// @param eventId Identificativo univoco dell'evento eliminato
    event EventDeleted(uint256 indexed eventId);
```

```
/**
 * @notice Registra un nuovo evento nella blockchain.
 * @dev La funzione è protetta dal modifier 'whenNotPaused', quindi non può essere eseguita se il contratto è in pausa.
 * @param _name Nome dell'evento.
 * @param _location Luogo dell'evento.
 * @param _date Data dell'evento in formato timestamp UNIX (deve essere futura).
 */
function registerEvent(
    string memory _name,
    string memory _location,
    uint256 _date
) external whenNotPaused {
    // Verifica che la data dell'evento sia nel futuro
    require(_date > block.timestamp, "La data dell'evento deve essere nel futuro");

    // Controlla che il nome dell'evento non sia vuoto
    require(bytes(_name).length > 0, "Il nome dell'evento non può essere vuoto");

    // Controlla che la posizione dell'evento non sia vuota
    require(bytes(_location).length > 0, "La posizione dell'evento non può essere vuota");

    // Aggiunge il nuovo evento all'array degli eventi registrati
    events.push(Event({
        name: _name,
        location: _location,
        date: _date,
        creator: msg.sender // L'utente che chiama la funzione diventa il creatore dell'evento
    }));

    // Calcola l'ID dell'evento come l'indice dell'array
    uint256 eventId = events.length - 1;

    // Emette un evento sulla blockchain per segnalare la registrazione dell'evento
    emit EventRegistered(eventId, _name, msg.sender);
}

/**
 * @notice Restituisce l'elenco di tutti gli eventi registrati.
 * @dev La funzione è 'view', quindi non modifica lo stato della blockchain.
 * @return Array di eventi registrati.
 */
function listEvents() external view returns (Event[] memory) {
    return events;
}

/**
 * @notice Trova tutti gli eventi creati da un particolare utente.
 * @dev La funzione effettua due cicli per raccogliere e restituire gli eventi creati dall'utente specificato.
 * @param _creator Indirizzo del creatore degli eventi da cercare.
 * @return Array di eventi creati dall'utente specificato.
 */
function findEventsByCreator(address _creator) external view returns (Event[] memory) {
    uint256 count = 0;

    // Primo ciclo: conta quanti eventi sono stati creati dall'utente
    for (uint256 i = 0; i < events.length; i++) {
        if (events[i].creator == _creator) {
            count++;
        }
    }

    // Crea un nuovo array della dimensione corretta per contenere gli eventi dell'utente
    Event[] memory creatorEvents = new Event[](count);
    uint256 index = 0;

    // Secondo ciclo: popola l'array con gli eventi dell'utente
    for (uint256 i = 0; i < events.length; i++) {
        if (events[i].creator == _creator) {
            creatorEvents[index] = events[i];
            index++;
        }
    }

    return creatorEvents;
}
```


PaymentManager.sol

Amministra i pagamenti e rimborsi in ETH, assicurando che i fondi siano bloccati fino alla conclusione dell'evento e gestendo rimborsi manuali in caso di annullamento.

depositFunds() → Permette agli utenti di depositare fondi per acquistare biglietti.

processRefund(address _user, uint256 _amount) → Esegue un rimborso manuale a un utente in caso di evento annullato.

releaseFundsToCreator(address _eventCreator, uint256 _amount) → Trasferisce i fondi accumulati all'organizzatore dopo la chiusura dell'evento.

```
function processRefund(address _user, uint256 _amount) external onlyOwner whenNotPaused {
    uint256 contractBalance = address(this).balance;

    // Registra un log di debug per il rimborso
    emit DebugLog("Tentativo di rimborso", contractBalance, _amount);

    // Verifica che il contratto abbia abbastanza fondi per il rimborso
    require(contractBalance >= _amount, "Fondi insufficienti per il rimborso");

    // Esegue il trasferimento dei fondi all'utente
    (bool success, ) = payable(_user).call{value: _amount}("");

    // Se il trasferimento fallisce, attiva l'Emergency Stop
    if (!success) {
        _pause();
        emit EmergencyStopActivated("Emergency Stop attivato! Fondi insufficienti.");
    }

    // Emette un evento per segnalare il rimborso
    emit RefundProcessed(_user, _amount);
}

/**
 * @notice Rilascia i fondi accumulati al creatore di un evento.
 * @dev Può essere chiamata per pagare il creatore di un evento dopo la vendita dei biglietti.
 * @param _eventCreator Indirizzo del creatore dell'evento.
 * @param _amount Importo da trasferire in wei.
 */
function releaseFundsToCreator(address _eventCreator, uint256 _amount) external whenNotPaused {
    // Verifica che l'indirizzo del creatore dell'evento sia valido
    require(_eventCreator != address(0), "Indirizzo del creatore non valido");

    // Controlla che il contratto abbia abbastanza fondi per il pagamento
    require(address(this).balance >= _amount, "Fondi insufficienti");

    // Esegue il trasferimento al creatore dell'evento
    (bool success, ) = payable(_eventCreator).call{value: _amount}("");

    // Verifica che il trasferimento sia avvenuto con successo
    require(success, "Transfer fallito");
}
```

```
/**
 * @title PaymentManager
 * @dev Contratto per la gestione dei pagamenti e dei rimborsi per gli eventi.
 * @dev Supporta depositi, prelievi, rimborsi e il rilascio di fondi ai creatori di eventi.
 */
You, settimana scorsa | 1 author (You)
contract PaymentManager is Pausable, Ownable {

    /// @notice Mappatura degli indirizzi degli utenti ai loro saldi in token ETH
    mapping(address => uint256) private balances;

    /// @notice Evento emesso quando un utente deposita fondi
    /// @param user Indirizzo dell'utente che deposita i fondi
    /// @param amount Importo depositato in wei
    event FundsDeposited(address indexed user, uint256 amount);

    /// @notice Evento emesso quando un utente preleva fondi
    /// @param user Indirizzo dell'utente che preleva i fondi
    /// @param amount Importo prelevato in wei
    event FundsWithdrawn(address indexed user, uint256 amount);

    /// @notice Evento emesso quando viene processato un rimborso a un utente
    /// @param user Indirizzo dell'utente rimborsato
    /// @param amount Importo rimborsato in wei
    event RefundProcessed(address indexed user, uint256 amount);

    /// @notice Evento emesso quando i fondi vengono rilasciati al creatore di un evento
    /// @param eventCreator Indirizzo del creatore dell'evento
    /// @param amount Importo trasferito in wei
    event FundsReleased(address indexed eventCreator, uint256 amount);

    /// @notice Evento di debug per registrare informazioni su rimborsi e bilanci
    /// @param message Messaggio di debug
    /// @param contractBalance Saldo del contratto in wei
    /// @param refundAmount Importo del rimborso in wei
    event DebugLog(string message, uint256 contractBalance, uint256 refundAmount);

    /// @notice Evento emesso quando viene attivato l'Emergency Stop
    /// @param message Messaggio di emergenza
    event EmergencyStopActivated(string message);

    /**
     * @notice Permette a un utente di depositare fondi nel contratto.
     * @dev La funzione è protetta dal modifier 'whenNotPaused', quindi non può essere eseguita se il contratto è in pausa.
     */
    function depositFunds() external payable whenNotPaused {
        // Aggiunge i fondi al saldo dell'utente
        balances[msg.sender] += msg.value;

        // Emette un evento per registrare il deposito
        emit FundsDeposited(msg.sender, msg.value);
    }
}
```

TicketManager.sol

Si occupa della creazione, validazione e rimborso dei biglietti, trasformandoli in NFT ERC-721 per garantire autenticità e tracciabilità.

- mintTicket(address _to, string _uri, uint256 _eventId): Crea un NFT ERC-721 come biglietto collegato a un evento.
- markTicketAsVerified(uint256 _ticketId): Valida il biglietto al momento dell'ingresso all'evento.
- isTicketVerified(uint256 _ticketId): Verifica se un biglietto è stato già validato.

- 1 Cosa sono gli NFT? **Token unici** su blockchain che certificano la **proprietà** di asset digitali, **non duplicabili** e **non intercambiabili**.
- 2 Perché vengono usati in TickeChain? Garantiscono **biglietti autentici e tracciabili**, eliminando il rischio di falsificazione e rendendo gli utenti proprietari effettivi.
- 3 Come funzionano in TicketManager.sol? Ogni biglietto è un **NFT ERC-721**, assegnato a un evento e un utente, validabile tramite QR code e rimborsabile tramite bruciatura (_burn()).

Design Patter Implementati

Permette di definire stati ben precisi e regolare le transizioni tra essi, evitando stati incoerenti.

State Machine

Evita attacchi di reentrancy nei rimborsi separando il prelievo dal pagamento.

Pull Payment Pattern

Protegge il contratto da attacchi di reentrancy, aggiornando prima lo stato e poi eseguendo interazioni esterne.

Checks-Effects-Interactions

Controlla rigorosamente i parametri prima di eseguire operazioni per prevenire errori e vulnerabilità.

Guard Check

Registry Pattern

Organizza e centralizza i dati degli eventi, evitando duplicazioni e rendendo le operazioni più efficienti.

Circuit Breaker (Emergency Stop)

Protegge il sistema da malfunzionamenti critici, bloccando temporaneamente operazioni rischiose.

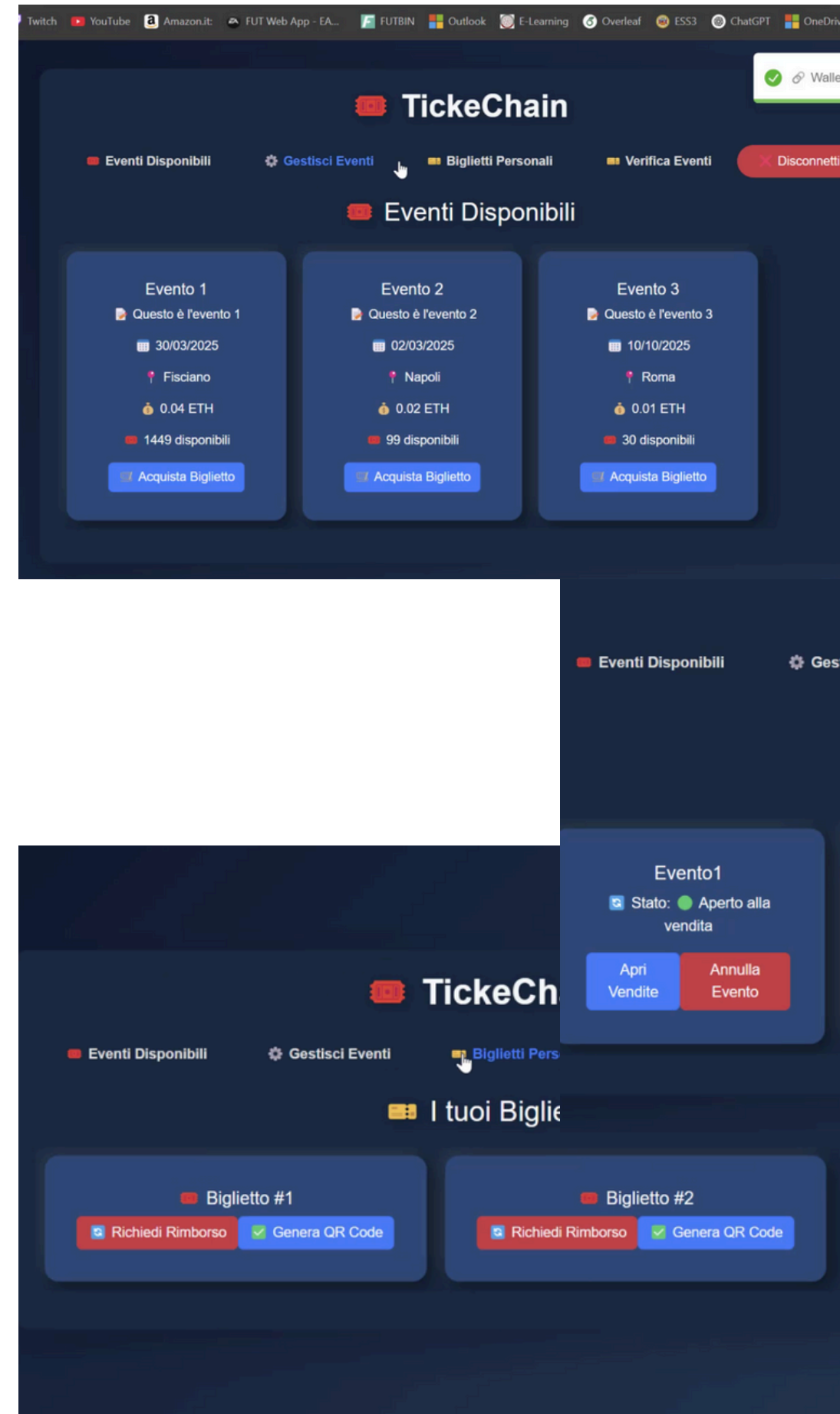
Secure Ether Transfer

Garantisce che i trasferimenti di ETH siano eseguiti in modo sicuro e senza vulnerabilità.

Front-End

Il frontend di TickeChain, sviluppato con **React + Vite**, offre un'interfaccia veloce e intuitiva per gestire eventi, biglietti e rimborsi. L'integrazione con **MetaMask** ed **Ethers.js** permette agli utenti di connettere il proprio wallet ed eseguire transazioni direttamente su blockchain, senza intermediari.

La dashboard interattiva mostra eventi attivi, biglietti posseduti e fondi disponibili, aggiornando i dati in tempo reale. Il design, basato su **Bootstrap**, garantisce un'esperienza responsiva e accessibile. Inoltre, il sistema supporta **QR code scanner**, consentendo la verifica sicura dei biglietti direttamente dall'app.



Testing e Debugging

TickeChain è stato testato con **Hardhat**, **Chai** e **Mocha**, garantendo la correttezza e la sicurezza degli smart contract. I test coprono creazione eventi, acquisto e rimborso biglietti, gestione fondi e protezione dagli attacchi.

Per il debugging, vengono utilizzati eventi Solidity per tracciare le operazioni e identificare eventuali anomalie. Le simulazioni su rete locale con **Hardhat** e **Ganache** permettono di replicare scenari reali e verificare il comportamento del sistema prima del deploy sulla blockchain.

```
C:\Users\anton\TickeChain>npx hardhat test
```

EventFactory

- ✓ Dovrebbe creare un evento
- ✓ Dovrebbe aggiornare un evento esistente
- ✓ Dovrebbe eliminare un evento esistente
- ✓ Dovrebbe cambiare lo stato dell'evento
- ✓ Dovrebbe ridurre il numero di biglietti disponibili (45ms)
- ✓ Dovrebbe annullare un evento e attivare Emergency Stop dopo 3 cancellazioni

EventRegistry

- ✓ Dovrebbe registrare un nuovo evento
- ✓ Dovrebbe restituire l'elenco di tutti gli eventi
- ✓ Dovrebbe trovare gli eventi creati da un utente specifico
- ✓ Dovrebbe eliminare un evento esistente

PaymentManager

- ✓ Dovrebbe permettere il deposito di fondi (simulato)
- ✓ Dovrebbe eseguire un rimborso (simulato) (40ms)
- ✓ Dovrebbe rilasciare i fondi al creatore dell'evento (simulato)
- ✓ Dovrebbe restituire il saldo di un utente (simulato)
- ✓ Dovrebbe attivare e disattivare Emergency Stop

TicketManager

- ✓ Dovrebbe creare un nuovo biglietto (43ms)
- ✓ Dovrebbe rimborsare e bruciare un biglietto
- ✓ Dovrebbe verificare un biglietto
- ✓ Dovrebbe controllare se un biglietto è attivo

```
19 passing (2s)
```



Grazie per l'attenzione!

