RELAZIONE

***-introduzione spiegando cosa ci è stato chiesto di fare***

L’obiettivo di questo progetto è creare un sistema che si occupi della vendita di biglietti online per degli eventi pubblici (concerti, spettacoli teatrali, incontri sportivi ecc.). Nel progetto bisogna includere, quindi, tutto ciò che riguarda la gestione degli eventi (creazione, modifica, cancellazione), la gestione dei biglietti (emettere il biglietto, apporre il sigillo fiscale, vendere il biglietto) ed infine implementare la parte relativa all’acquisto del biglietto. In particolare, uno dei requisiti del sistema è di memorizzare una parte degli asset in una blockchain di tipo Ethereum.

Sono stati individuati tre principali attori che dovranno interagire con il sistema: il gestore degli eventi (event manager), il rivenditore (ticket reseller) e l’acquirente (ticket buyer). Gli asset che andranno memorizzati nella blockchain sono: gli eventi, i biglietti venduti comprensivi di sigillo fiscale e le richieste di acquisto da parte degli acquirenti. I biglietti venduti saranno rappresentati nella blockchain come NFT (Non-Fungible Token).

Il progetto è stato sviluppato in diverse fasi: in prima analisi (**Early Analysis**) sono stati individuati i compiti dei tre attori, l’elenco dei principali asset e le interazioni tra loro. Successivamente è stata aggiunto come ulteriore attore il sistema stesso (**Late Analysis**), e infine si è proceduto con una valutazione dei rischi. Per la rappresentazione delle varie analisi sono stati usati i diagrammi i\* ed altre tabelle standard, in particolare lo Stride Model.

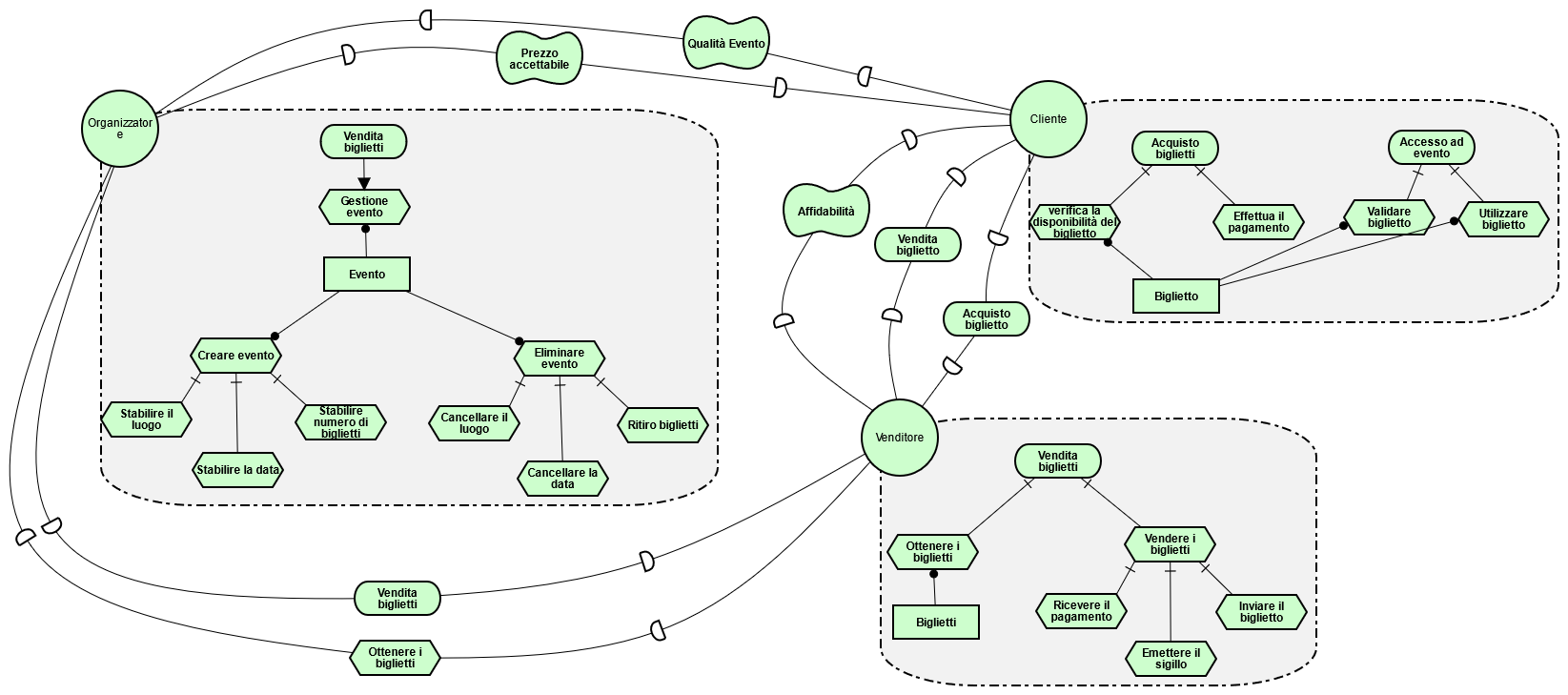
Dopo queste prime analisi si è passati a decidere quali parti del sistema dovessero essere implementate all’interno della blockchain e quali no e quali tecnologie adottare per l’implementazione del sistema software.

*-****diagrammi i star***

**Early Analysis**

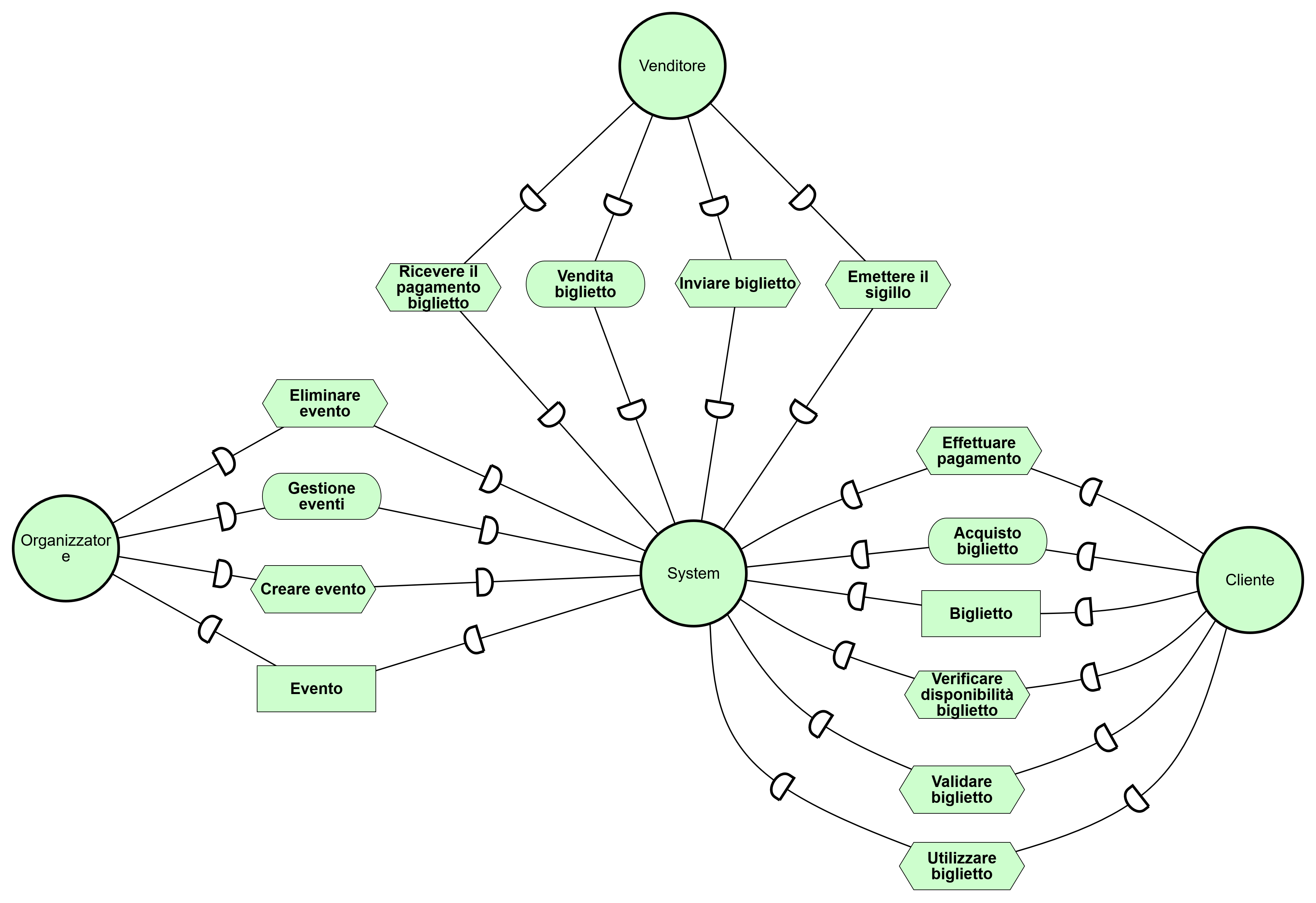
Per ogni soggetto sono stati individuati i corrispondenti goal scomposti a loro volta in risorse e compiti.

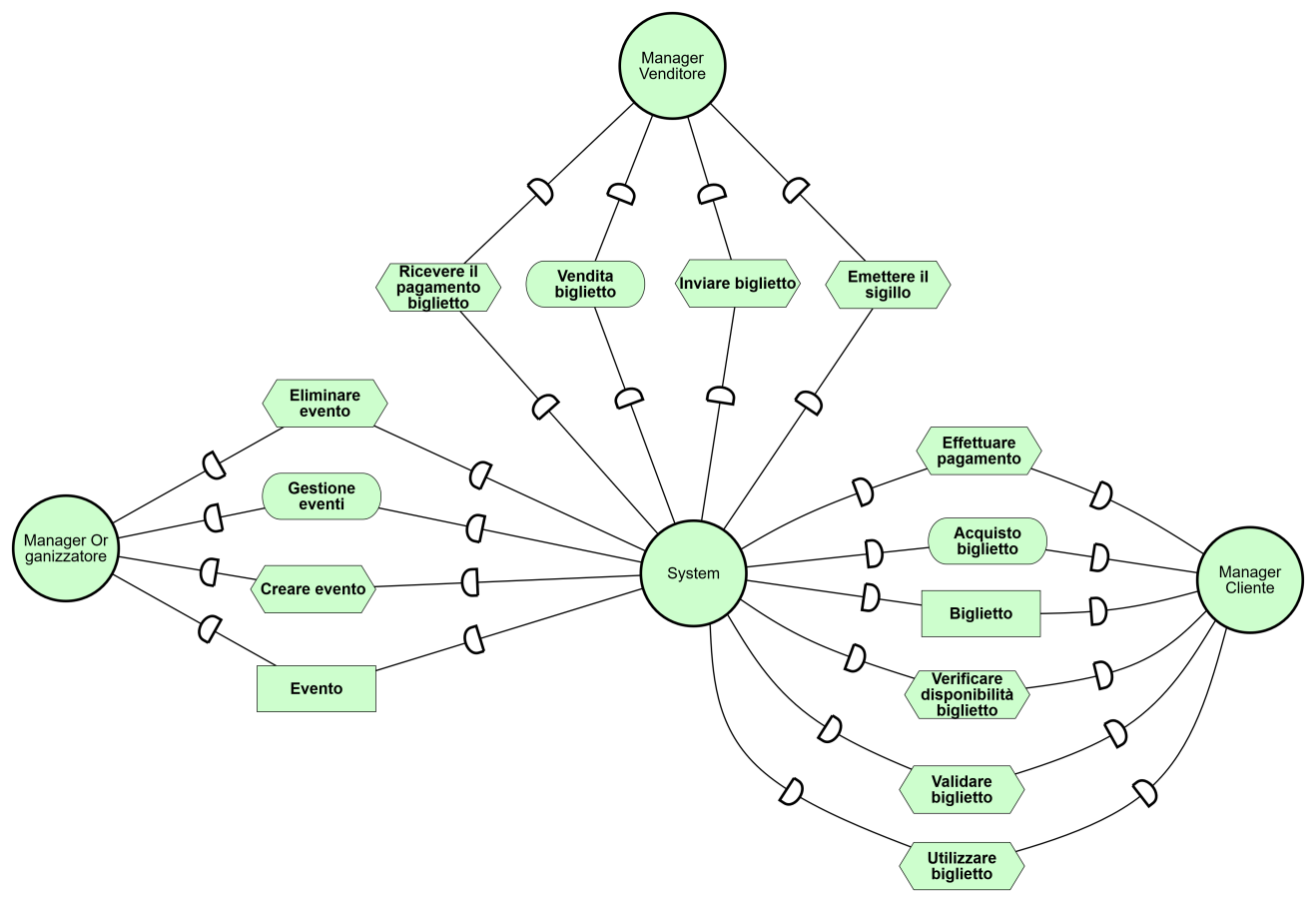
L’organizzatore dell’evento è colui che si occupa della gestione degli eventi quindi della loro creazione, modifica e cancellazione. Il venditore dei biglietti è colui che si occupa della vendita all’utente dei biglietti. L’utente si registra al sistema e richiede biglietti degli eventi a cui è interessato. Abbiamo individuato anche le dipendenze tra i soggetti come si può vedere nel seguente diagramma:



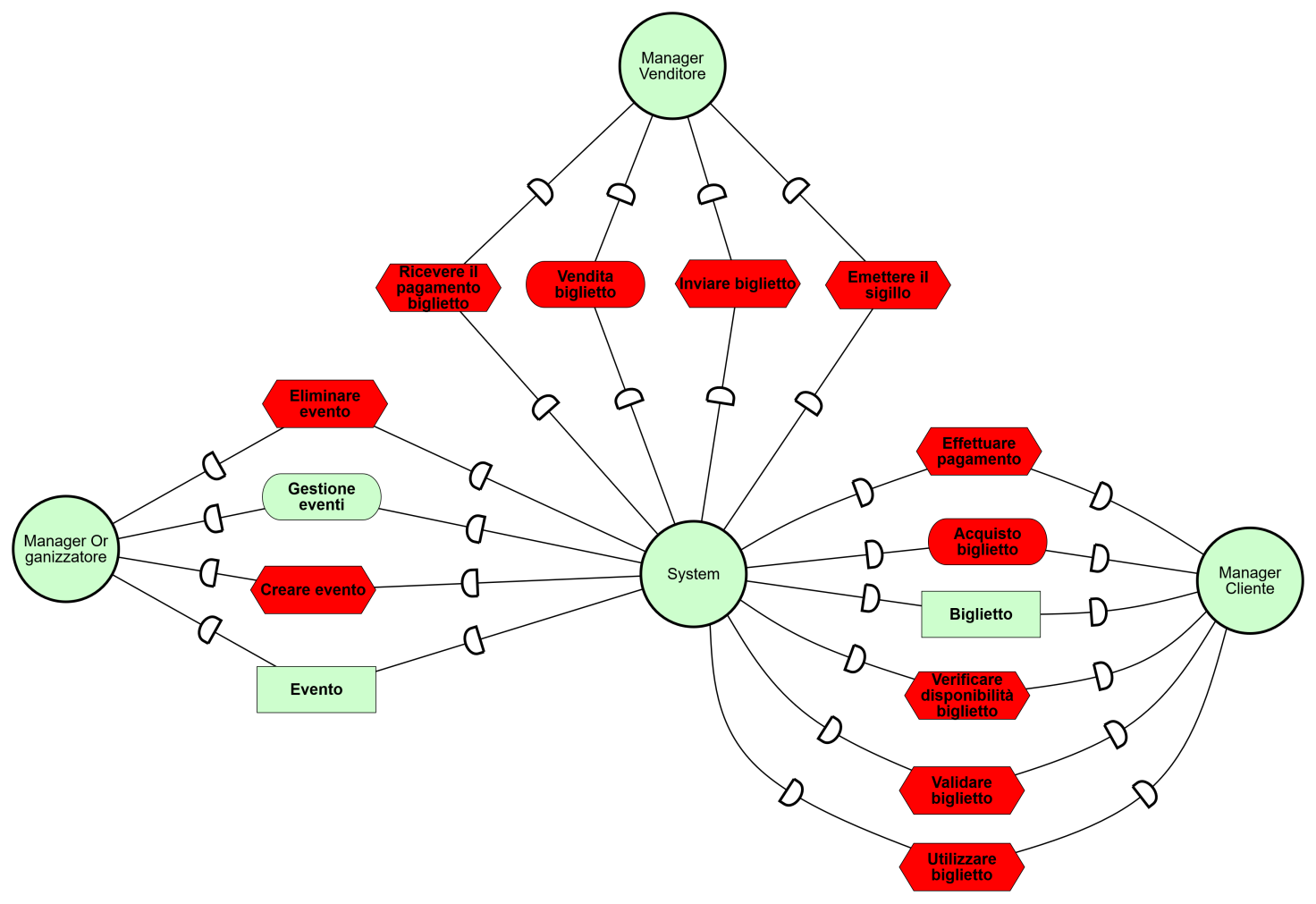
**Late Analysis & Architectural Analysis**

Nel primo diagramma vengono evidenziate le dipendenze tra i soggetti prima indicati e il sistema, mentre nel secondo il sistema viene separato nelle sue varie componenti che andranno ad interagire direttamente con gli altri attori.





**Risk Assessment**

Nel seguente diagramma sono stati colorati in rosso gli asset ritenuti vulnerabili

Ad ogni asset è stata data un’importanza usando una scala che va da 1 a 5 e il relativo impatto che avrebbero nel caso in cui venissero attaccati. I valori sono riportati nella seguente tabella:

|  |  |  |
| --- | --- | --- |
| Asset | Value | Exposure (Impact) |
| Creare evento | 5. | 5. |
| Eliminare evento | eventi potenzialmente critici per la sicurezza | si avrebbero perdite finanziarie se gli eventi venissero cancellati da persone non autorizzate. Costi di ripristino del sistema. Danni ai clienti se persone non autorizzate creano eventi falsi |
| Ricevere il pagamento | 5. | 5. |
| Emettere il sigillo | eventi potenzialmente critici per la sicurezza | perdita finanziaria se i pagamenti sono falsi. Evasione fiscale se i sigili sono falsi. Danni a carico del cliente se i biglietti non vengono recapitati |
| Inviare il biglietto |
| Effettuare il pagamento | 4. | 4. |
| Validare il biglietto |  | Dati sensibili posso essere intercettati. Cliente non può entrare all'evento se il biglietto è stato validato da altri. Perdita della reputazione |
| verificare disponibilità del biglietto | 3. | 5. |
|  |  | perdita economica se il sistema viene corrotto e gli eventi risultano esauriti |

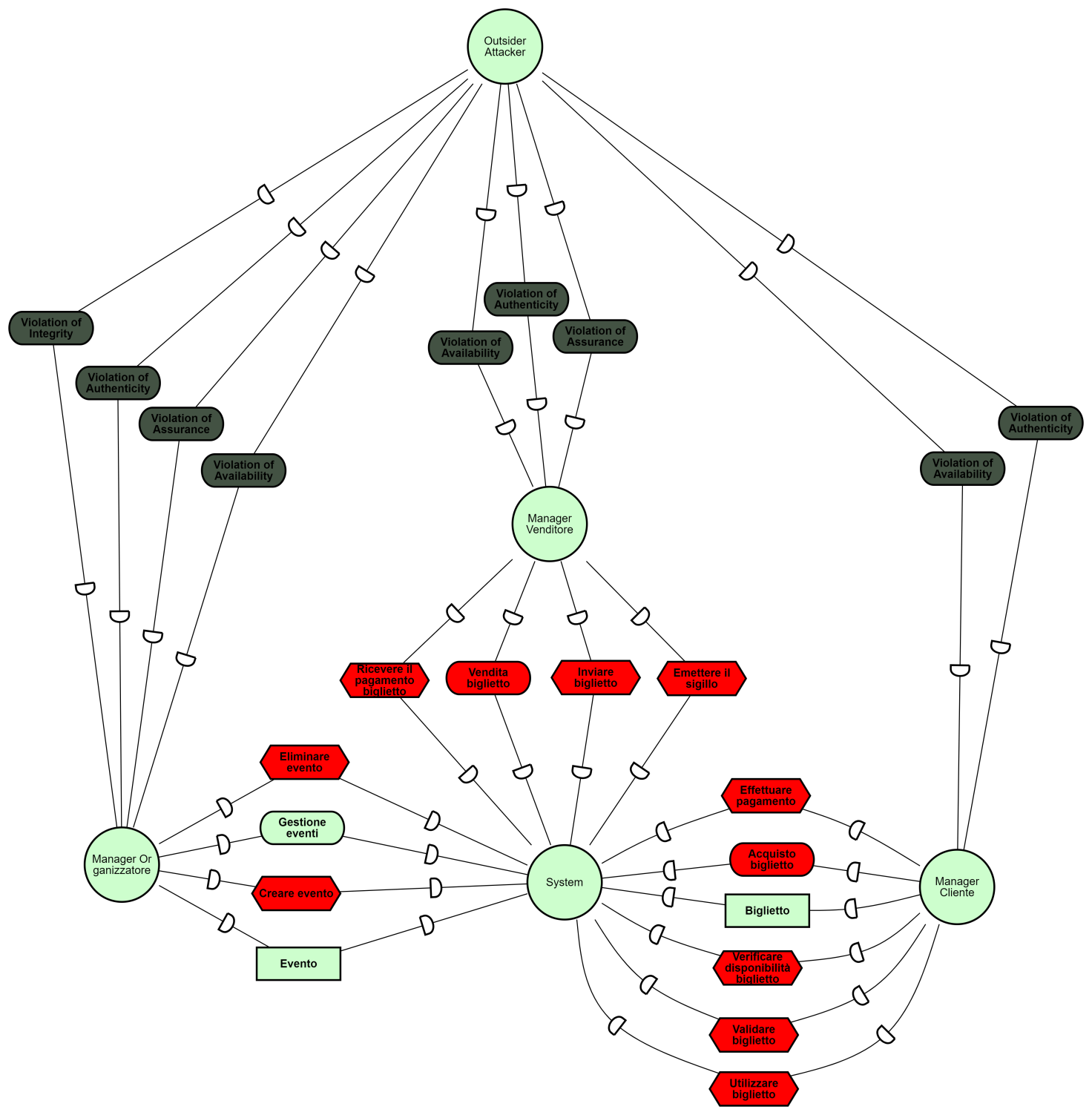
Analizzando gli asset vulnerabili si è cercato di capire quali potrebbero essere i possibili attacchi, i quali sono stati divisi in due tipi: attacchi intenzionali(abuse case) e accidentali (misuse case).

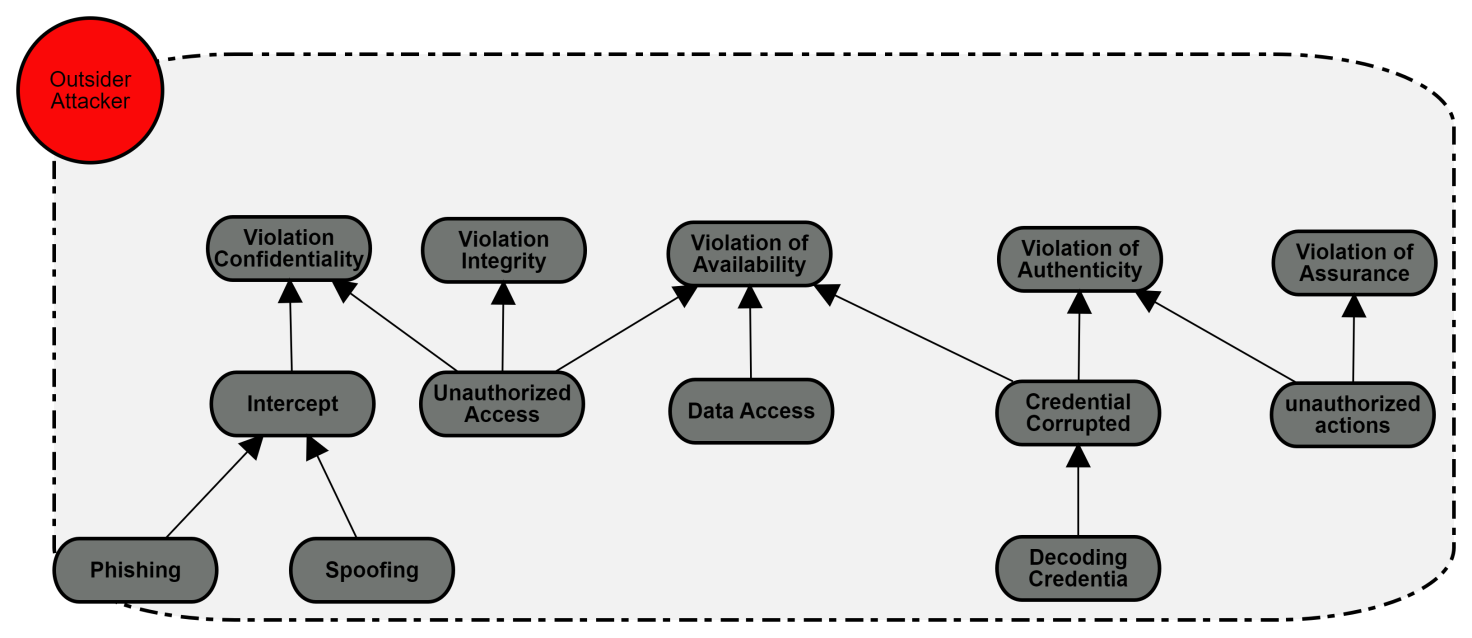
Abuse Case:

* Creazione/cancellazione eventi non previsti: un soggetto esterno malevolo dopo aver rubato all’account all’event manager, a causa di password semplici o in chiaro o con tecniche di phishing, riesce a creare e/o eliminare degli eventi.
* Intercettare il pagamento: il soggetto malevolo intercetta i dati di pagamento ingannando il cliente con server di spoofing.
* Intercettare il biglietto: L'attaccante avendo accesso all'email del cliente riesce ad ottenere il biglietto. Si potrebbe fare in modo che il biglietto sia stampabile solo entrando nel sito di acquisto e nella mail viene inviato solo il link al biglietto.
* Corrompere l’account del venditore: l’attaccante riesce a corrompere l'account del venditore e richiedere sigilli a suo nome. Questo potrebbe avvenire se l’attaccante riuscisse a rubare l’account del venditore con tecniche di phishing o intercettare le sue credenziali perché sono troppo facili
* Attacco del 51%: Un attaccante con risorse di calcolo maggiori al resto della rete potrebbe dominare la verifica e l'approvazione delle transazioni, controllandone il contenuto.
* Furto d’identità: Dal momento che la blockchain preserva l'anonimità e la privacy, la sicurezza degli asset dipende dalla difesa della chiave privata. Se un attaccante acquisisse la chiave privata di un account nessuno potrebbe recuperarlo.
* Double spending: Un attaccante potrebbe usare lo stesso token digitale per poter effettuare più di un pagamento.

Tutti gli abuse case individuati sono stati descritti in modo più dettagliato nelle seguenti tabelle, per ognuno di essi si è andati ad indicare: i soggetti che subiscono l’attacco, i dati sul quale agiscono, le precondizioni affinché avvenga l’attacco, l’insieme di azioni che devono essere svolte dal soggetto malevolo, le conseguenze dell’attacco, quali azioni possono essere intraprese per cercare di prevenire l’attacco ed in fine i requisiti non funzionali che vengono violati.

|  |  |
| --- | --- |
| Use case ID: | AT-01-01 |
| Use Case Name: | Creazione eventi non previsti |
| Actor | Attaccante |
| Description | Creare eventi senza averne autorizzazione dopo aver rubato account |
| Data (asset) | Evento nel db |
| Stimulus and Pre. | Password debole o in chiaro(2) |
| Attack 1 Flow | Account organizzatore rubato tramite tecniche di phishing |
| Attack 2 Flow | Account organizzatore rubato per password troppo semplice o in chiaro |
| Response and Post. |  |
| Mitigations | Autenticazione cifrata, ACLs, ruoli con privilegi minimi per i vari attori. Usare HTTPS, Controllo accessi tramite Firewall |
| Non Functional Requirements |  |
|  |  |
| Use case ID: | AT-01-02 |
| Use Case Name: | Cancellazione eventi non prevista |
| Actor | Attaccante |
| Description | Cancellare eventi senza averne autorizzazione dopo aver rubato account |
| Data (asset) | Evento nel db |
| Stimulus and Pre. | Password debole o in chiaro(2) |
| Attack 1 Flow | Account organizzatore rubato tramite tecniche di phishing |
| Attack 2 Flow | Account organizzatore rubato per password troppo semplice o in chiaro |
| Response and Post. |  |
| Mitigations | Autenticazione cifrata, ACLs, ruoli con privilegi minimi per i vari attori. Usare HTTPS, Controllo accessi tramite Firewall |
| Non Functional Requirements |  |
|  |  |
| Use case ID: | AT-01-06 |
| Use Case Name: | Effettuare pagamento: Intercettare dati di pagamento |
| Actor | Attaccante, Cliente |
| Description | Intercettare dati di pagamento ingannando il cliente con server di spoofing |
| Data (asset) | Dati di pagamento |
| Stimulus and Pre. | Invio pagamento e cliente connesso al server di spoofing |
| Attack 1 Flow | Un network monitor viene inserito nel sistema in modo da intercettare i pacchetti |
| Attack 2 Flow | Server web di spoofing simile a un sito per l'acquisto biglietto |
| Response and Post. |  |
| Mitigations | Autenticazione cifrata, Certificati. Usare HTTPS con certificati validi |
| Non Functional Requirements |  |
|  |  |
|  |  |
| Use case ID: | AT-01-03 |
| Use Case Name: | Intercettare il biglietto |
| Actor | Attaccante, Cliente, Venditore |
| Description | L'attaccante avendo accesso all'email del cliente riesce ad ottenere il biglietto |
| Data (asset) | Biglietto |
| Stimulus and Pre. | Invio del biglietto al cliente sull'email. L'attaccante ha accesso alla mail del cliente |
| Attack 1 Flow | Password del'account email troppo semplice |
| Attack 2 Flow | Rubare l'account attraverso tecniche di phishing |
| Response and Post. |  |
| Mitigations | Biglietto stampabile solo entrando nel sito di acquisto, nella mail viene inviato solo il link al biglietto. Il biglietto può essere visualizzato ed eventualmente stampato solo dal sito con autenticazione e autorizzazione dell'utente. Autenticazione cifrata, Certificati. Usare HTTPS con certificati validi |
| Non Functional Requirements |  |
|  |  |
|  |  |
| Use case ID: | AT-01-08 |
| Use Case Name: | Corrompere validazione biglietto |
| Actor | Attaccante, Cliente |
| Description | Convalidare il biglietto al posto del cliente che lo ha acquistato |
| Data (asset) | Dati del biglietto |
| Stimulus and Pre. | Intercettare i dati del biglietto ingannando il cliente |
| Attack 1 Flow | Rubare account cliente con tecniche di phishing |
| Attack 2 Flow | Rubare credenziali cliente |
| Response and Post. |  |
| Mitigations | Autenticazione cifrata, Usare HTTPS |
| Non Functional Requirements |  |
|  |  |
| Use case ID: | AT-01-05 |
| Use Case Name: | Corrompere account Venditore |
| Actor | Attaccante, Venditore |
| Description | corrompere l'account del veniditore e richiedere sigilli a suo nome |
| Data (asset) | Evento nel db |
| Stimulus and Pre. | Intercettare le credenziali dell'account del venditore |
| Attack 1 Flow | Rubare account venditore con tecniche di phishing |
| Attack 2 Flow | Intercerttare credenziali venditore perchè troppo facili |
| Response and Post. |  |
| Mitigations | Autenticazione cifrata, ACLs, ruoli con privilegi minimi per i vari attori. Usare HTTPS, Controllo accessi tramite Firewall |
| Non Functional Requirements |  |
|  |  |
| Use case ID: | AT-01-09 |
| Use Case Name: | Attacco del 51% |
| Actor | Attaccante |
| Description | Un attaccante con risorse di calcolo maggiori al resto della rete potrebbe dominare la verifica e l'approvazione delle transazioni, controllandone il contenuto |
| Data (asset) | Dati biglietto, dati account, dati transazioni, wallet dei clienti |
| Stimulus and Pre. | Rete distribuita troppo piccola |
| Attack 1 Flow | il nodo dominante(attaccante) può superare tutti gli altri nodi e si impropria della rete |
| Attack 2 Flow | L'attaccante propaga sulla rete blocchi falsi, contenenti operazioni fraudolente |
| Response and Post. |  |
| Mitigations | Utilizzare tecniche di Proof of Stake, evitare reti troppo piccole |
| Non Functional Requirements |  |
|  |  |
| Use case ID: | AT-01-10 |
| Use Case Name: | Furto di identità |
| Actor | Attaccante |
| Description | Dal momento che la blockchain preserva l'anonimità e la privacy, la sicurezza degli asset dipende dalla difesa della chiave privata. Se un attaccante acquisisse la chiave privata di un account nessuno potrebbe recuperarlo. |
| Data (asset) | dati account, wallet dei clienti (tutti gli asset posseduti dalla vittima) |
| Stimulus and Pre. | Algoritmi crittografici poco robusti |
| Attack 1 Flow | L'attaccante sfrutta la debolezza degli alogirtmi crittografici e riesce a craccare la chiave privata |
| Attack 2 Flow |  |
| Response and Post. |  |
| Mitigations | Utilizzare standard crittografici |
| Non Functional Requirements |  |
|  |  |
| Use case ID: | AT-01-11 |
| Use Case Name: | Double spending |
| Actor | Attaccante, Cliente |
| Description | Un attaccante potrebbe usare lo stesso token digitale per poter effettuare più di un pagamento |
| Data (asset) | Eventi |
| Stimulus and Pre. | Codici sorgenti non aggiornati, utilizzo di tecnologie obsolete |
| Attack 1 Flow | L'attaccante/cliente, sfruttando l'eventuale facilità di replicare le informazioni, consuma la stessa valuta digitale per poter effettuare pagamenti diversi |
| Attack 2 Flow |  |
| Response and Post. |  |
| Mitigations | Manutenzione periodica del sistema, revisione codici sorgenti, aggiornamento delle tecnologie utlizzate |
| Non Functional Requirements |  |

Gli abuse case sono stati riportato sottoforma di diagramma i\*

È stato inserito uno schema per evidenziare quali proprietà di sicurezza possono essere violate da un eventuale attaccante esterno 

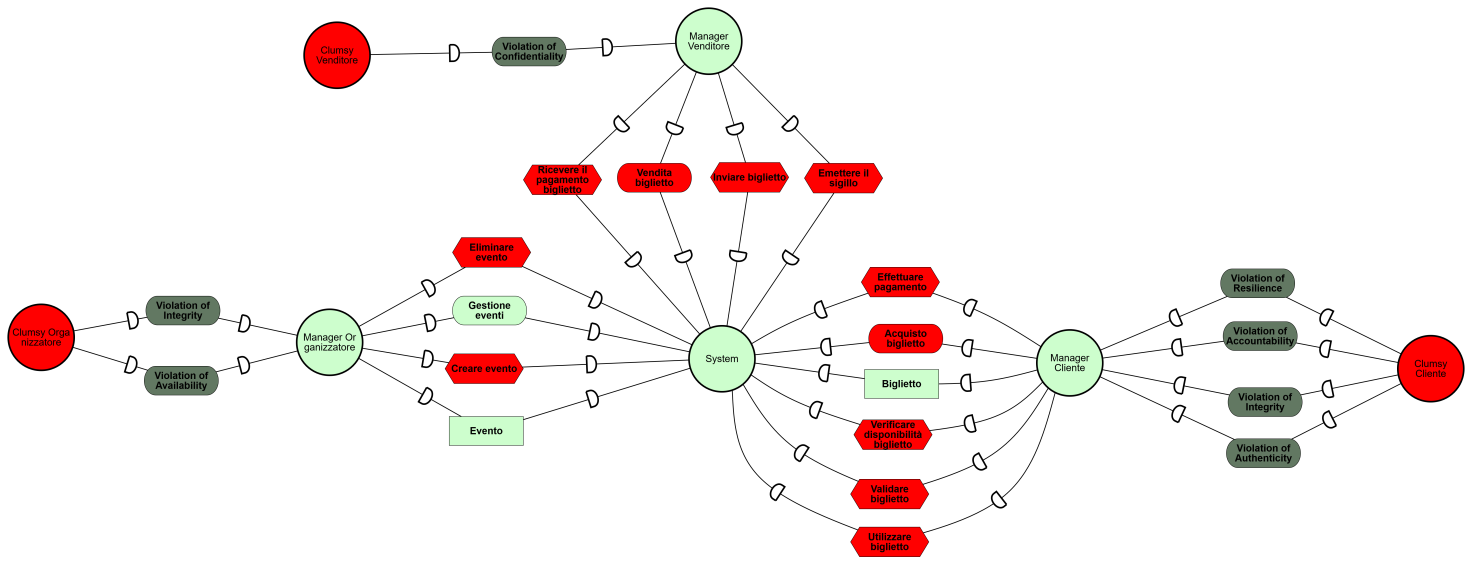
Si è proceduto con l’analisi di quelli che potrebbero essere gli attacchi causati da soggetti interni al sistema in modo accidentale redigendo i misuse case ed individuando i seguenti tipi di attacco:

* Intercettare i dati del pagamento o del sigillo: questo potrebbe avvenire se il venditore salva i dati del pagamento del cliente il locale o per il sigillo se il cliente salva quest’ultimo in locale. Per quanto riguarda il pagamento bisognerebbe evitare che il venditore abbia accesso ai dati di pagamento, oppure prevedere solo un accesso parziale. Per il sigillo dovrebbe essere non replicabile e nominativo.
* Eliminare un evento: ossia l’eliminazione accidentale di un evento, basterebbe chiedere la conferma prima che l’evento venga eliminato.
* Creazione di eventi fasi: Un organizzatore può creare un evento che in realtà non esiste ad esempio con una data obsoleta, luogo non corretto. Per evitare questo problema bisognerebbe introdurre dei controlli di sistema prima che gli eventi vengano pubblicati.
* Creazione di account falsi: Un cliente può avere la possibilità di creare un numero elevato di account falsi. Creando un numero notevole di nodi falsi nella rete, un cliente può causare ai nodi reali rallentamenti e interruzioni. Anche per mitigare questo problema si potrebbero introdurre controlli di sistema per la verifica dei nuovi account.

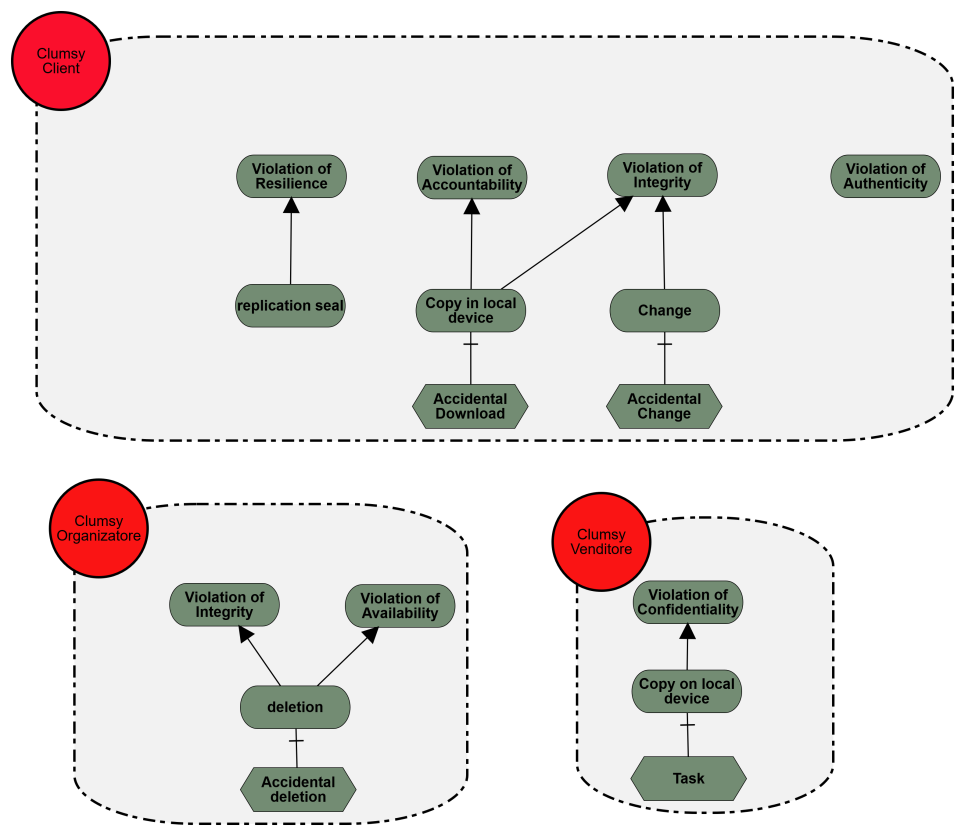
Anche i misuse case sono stati riportati sottoforma tabellare, le tabelle riportano le stesse informazioni di quelle mostrate in precedenza per gli abuse case.

|  |  |
| --- | --- |
| Use case ID: | AT-01-03 |
| Use Case Name: | Intercettare dati di pagamento |
| Actor | Venditore |
| Description | Venditore salva i dati di pagamento del cliente in locale |
| Data (asset) | Dati di pagamento |
| Stimulus and Pre. | Dati di pagamento in chiaro accedibili da organizzatore |
| Attack 1 Flow | Venditore salva una copia dei dati di pagamento del cliente in locale |
| Attack 2 Flow | Un attaccante accede alla macchina del venditore e ottiene i dati di pagamento del cliente |
| Response and Post. | Una copia dei dati di pagamento sono salvati in locale nella macchina del venditore |
| Mitigations | Evitare che il venditore abbia accesso ai dati di pagamento, oppure prevedere solo un accesso parziale |
| Non Functional Requirements | Confidentiality |
|  |  |
| Use case ID: | AT-01-05 |
| Use Case Name: | Intercettare sigillo |
| Actor | Cliente |
| Description | Il cliente salva il sigillo in locale |
| Data (asset) | Sigillo |
| Stimulus and Pre. | Sigillo non nominativo e facilmente replicabile |
| Attack 1 Flow | Cliente salva una copia del sigillo in locale |
| Attack 2 Flow | Un attaccante accede alla macchina del cliente e ottiene il sigillo |
| Response and Post. | Il sigillo è salvato in locale nella macchina del cliente |
| Mitigations | Rendere sigillo non replicabile e nominativo |
| Non Functional Requirements | Integrity, Authenticity, Accountability, Resilience |
|  |  |
|  |  |
| Use case ID: | AT-01-02 |
| Use Case Name: | Eliminazione Evento |
| Actor | Organizzatore |
| Description | l'organizzatore elimina accidentalmente un evento |
| Data (asset) | Evento |
| Stimulus and Pre. | l'evento viene eliminato senza la richiesta di una conferma |
| Attack 1 Flow | l'organizzatore effettua accidentalemte la chiamata alla funzione per eliminare l'vento |
| Attack 2 Flow |  |
| Response and Post. | l'evento viene eliminato |
| Mitigations | richichiere la conferma prima dell'eliminazione completa dell'evento |
| Non Functional Requirements | Integrity, Availability |
|  |  |
| Use case ID: | AT-01-11 |
| Use Case Name: | Creazione account falsi |
| Actor | Cliente |
| Description | Un cliente può avere la possibiltà di creare un numero elevato di account falsi.Creando un numero notevole di nodi falsi nella rete, un cliente/attaccante può causare ai nodi reali rallentamenti e/o interruzioni. |
| Data (asset) | Evento |
| Stimulus and Pre. | Facilità nel creare account falsi |
| Attack 1 Flow | L'introduzione di tanti nuovi nodi nella rete rende necessaria una lunga sincronizzazione di questi utlimi con i nodi reali. Il servizio potrebbe rallentare. |
| Attack 2 Flow |  |
| Response and Post. | Rallentamenti e/o interruzioni del servizio |
| Mitigations | Introdurre controlli di sistema per la verifica dei nuovi account |
| Non Functional Requirements | Integrity, Availability, Authenticity |
|  |  |
| Use case ID: | AT-01-12 |
| Use Case Name: | Creazione eventi falsi |
| Actor | Organizzatore |
| Description | Un organizzatore può creare un evento che in realtà non esiste (data dell'evento obsoleta, luogo non corretto, finzione dell'esistenza di un evento,...) |
| Data (asset) | Evento |
| Stimulus and Pre. | Mancanza di controlli del sistema |
| Attack 1 Flow | L'organizzatore crea un evento non esistente e lo diffonde nella rete. |
| Attack 2 Flow |  |
| Response and Post. | L'evento falso viene pubblicato |
| Mitigations | Introdurre controlli di sistema prima della pubblicazione di nuovi eventi |
| Non Functional Requirements | Integrity, Availability |

Di seguito viene riportato il diagramma i\* dei misuse case:



Per rendere le cose più chiare e schematiche sono stati creati degli schemi con le proprietà che ogni attore del sistema potrebbe violare e le relative cause per cui avviene tale violazione



Abbiamo terminato la valutazione del rischio con le contromisure da adottare per mitigare ed eventualmente eliminare tutti i rischi individuati. Tali considerazioni sono state rappresentante nella seguente tabella, dove sono stati riportati tutti gli asset individuati in precedenza e per ognuno si va ad indicare: quali tipi di attacchi possono esserci, con quale probabilità possono essere attaccati, come si potrebbe prevenire l’attacco ed il relativo costo in caso di attacco.

mettere la tabella risk assessment and mitigation table aggiornata

***-cosa abbiamo messo nella blockchain***

Un altro passo del progetto è stato quello di definire cosa dovesse essere implementato all’interno della blockchain. I dati inseriti nella blockchain saranno comunque ridotti, per motivi di efficienza, rispetto ai dati completi che invece saranno memorizzati in un database relazionale. Di conseguenza la fonte dati più completa e puntuale sarà questo database.

La decisione di usare un database al di fuori della blockchain comporta però un rischio aggiuntivo che è quello di mantenere consistenza tra blockchain e database, in particolare, c’è il rischio che gli attori utilizzino i contract senza passare dal software. Ai fini del progetto si decide di accettare questo rischio cercando di mitigarlo con opportuni controlli implementati nei contract.

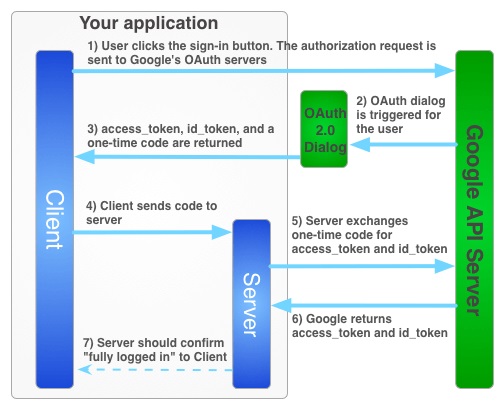
Di seguito per ogni asset si è deciso come procedere:

* **Pagamento del biglietto:** per motivi di efficienza e dato che nel progetto verrà usata una blockchain di test si è deciso di non includere il pagamento nella blockchain, verrà implementata una funzione che simuli un pagamento fittizio che può fallire o avere successo.
* **Emissione del sigillo fiscale:** ai fini del progetto si è deciso di automatizzare la generazione del sigillo fiscale al momento del pagamento sottoforma di hash, poi in fase di emissione del biglietto verrà registrato in blockchain insieme ad altri dati del biglietto.
* **Gestione degli eventi:** gli eventi verranno registrati in blockchain ma con un numero ridotto di informazioni.
* **Verifica della disponibilità del biglietto**: per motivi di efficienza è stata implementata offchain.
* **Creazione del biglietto:** Il gestore degli eventi si occupa di generare i biglietti relativi ad un evento, solo dopo la vendita il biglietto verrà generato anche nella blockchain sottoforma di NFT registrando anche vari dati, tra cui il sigillo fiscale.

**-*soluzioni tecnologiche adottate***

Abbiamo deciso di realizzare il sistema creando un backend, realizzato in Java e framework Spring Boot, che fornisca un numero limitato di servizi Rest con dati Json. Gli accessi ai servizi vengono regolati con autenticazione tramite **Google account**. L’applicazione fatta in questo modo permette indistintamente la realizzazione di un **frontend web o mobile**. Inoltre, l’autenticazione con account Google garantisce sicurezza e affidabilità, considerato anche il suo l’ampio utilizzo nel panorama delle applicazioni mobile e web. Un altro considerevole vantaggio di questa scelta è la **non memorizzazione** della password dell’utente nel nostro database, evitando quindi tutte le implicazioni di sicurezza, in senso lato, che avrebbe richiesto tale opzione.

Per l’autenticazione e l’autorizzazione all’utilizzo dei servizi Rest del backend è stata usata una leggera variante del protocollo **OAuth2** sfruttando il **server di Google**. Il seguente schema mostra il funzionamento del protocollo OAuth2 tramite Google server “ordinario”:



1. Il frontend invia una richiesta di autorizzazione al server di Google, che restituisce la pagina di login;
2. Dopo il corretto login il server restituisce al frontend il token id e l’access token;
3. Il frontend invia al backend il token id e l’access token;
4. Il backend verifica il token id e l’access token con il server di Google;
5. Se la verifica ha successo il flusso è completo.

La **variante** che utilizzeremo differisce dal precedente protocollo in questo modo:

1. Il frontend, dopo il login, riceve dal server di Google un codice di autorizzazione che invia al backend;
2. Il backend invia il codice autorizzazione, client id e client secret (definiti nella console di google) al server di Google per la verifica;
3. Se la verifica ha successo, le informazioni ricevute dal server di Google vengono “assemblate” in un **token JWT** che viene inviato al frontend;
4. Il frontend usa il token JWT nell’header delle richieste da mandare al backend.

Si è scelto di utilizzare la variante per sfruttare il flusso di autenticazione OAuth già nativo di Spring Security, evitando quindi di dover scrivere nuove classi ad hoc per l’esigenza.

Sono stati definiti cinque ruoli che definiscono l’**autorizzazione** all’utilizzo di specifici servizi:

* **ROLE\_USER** (Livello utente generico – Ruolo minimo posseduto da tutti gli utenti autenticati)
* **ROLE\_TICKET\_BUYER** (Acquirente)
* **ROLE\_TICKET\_RESELLER** (Rivenditore)
* **ROLE\_EVENT\_MANAGER** (Gestore eventi)
* **ROLE\_ADMIN** (Amministratore di sistema)

Per l'implementazione degli smart contract abbiamo utilizzato il linguaggio **Solidity**, i contract sono stati poi compilati tramite tool **solc** e tradotti in classi Java utilizzabili dal backend tramite l’apposito tool di **web3j.**

Per la realizzazione del frontend abbiamo scelto di usare la libreria javascript **React**.

Si è deciso di cifrare le comunicazioni tra frontend e backend tramite HTTPS e protocollo di comunicazione TLS 1.3, essendo un progetto realizzato a scopo didattico sono stati creati dei certificati Self-Signed.

Per effettuare i test abbiamo usato un DBMS **MySQL** e **Ganache** per la blockchain Ethereum.

Nel backend abbiamo usato **AspectJ** per loggare le chiamate ai servizi e i relativi risultati (questa parte va scritta diversamente e forse va spostata in un altro punto della relazione)

-Prerequisiti e ipotesi di lavoro

Event Manager e Ticket Reseller vengono definiti a priori o inseriti dagli amministratori di sistema, ogni nuovo utente viene inserito con ruoli: **ROLE\_USER** e **ROLE\_TICKET\_BUYER**.

La parte amministrativa non è stata implementata perché non richiesta nei requisiti del progetto.

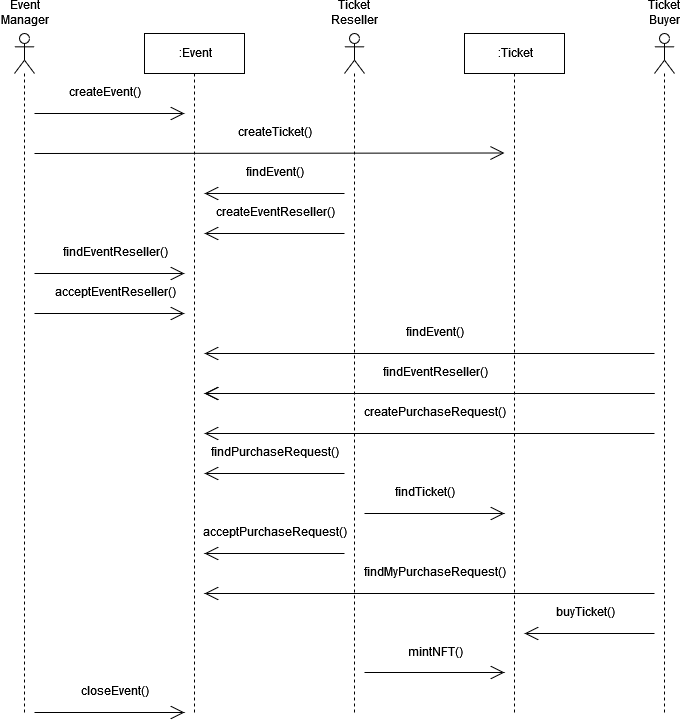
Il prezzo del biglietto viene deciso dall’event manager, e si suppone che abbia già un accordo con i rivenditori del proprio evento.

Anche se non specificato nei requisiti di progetto abbiamo deciso di dare la possibilità ad un utente con ROLE\_USER di inserire/modificare/cancellare dei propri commenti ad un evento, leggibili da tutti agli altri user autenticati.

***-inserire descrizione funzionamento sistema***

Prima di passare alla trattazione delle singole classi si cercherà di mettere in evidenza, in maniera sintetica, il workflow ordinario del software, indicando anche la sequenza di chiamate ai servizi da parte dei vari attori.

Di seguito viene riportato un diagramma selle sequenze UML rappresentante il workflow:



* Il gestore dell’evento dopo essersi loggato nel sistema, può creare un evento (***createEvent***). L’evento verrà inserito sia nel DB che in blockchain;
* Dopo aver creato l'evento può creare i relativi biglietti (***createTicket***). In questa fase i biglietti verranno registrati solo nel DB;
* A questo punto il rivenditore può ricercare gli eventi ancora aperti (**findEvent**), e deve fare una richiesta nel sistema per proporsi come rivenditore dell’evento (***createEventReseller***). Gli EventReseller saranno registrati sia nel DB che nella blockchain;
* Il gestore dell’evento, che si suppone abbia già fatto accordi con i rivenditori, trova (***findEventReseller***) e accetta (***acceptEventReseller***) le richieste dei rivenditori. Quest’accettazione viene inserita sia nel DB che in blockchain;
* Dal momento che un evento ha almeno un rivenditore registrato e accettato, l’acquirente può cominciare a richiedere i biglietti di un certo evento ancora aperto (**findEvent**) a un rivenditore (***findEventReseller***). Verrà creata una PurchaseRequest (***createPurchaseRequest***) sia nel DB che nella blockchain;
* Il rivenditore cerca le PurchaseRequest per le quali è stato indicato come rivenditore (***findPurchaseRequest***), poi verifica la disponibilità di biglietti (***findTicket***), ed evade in rigoroso ordine di arrivo (è vincolato) le PurchaseRequest. Può accettare una richiesta assegnando un biglietto a tale richiesta (***acceptPurchaseRequest***) o rifiutarla. L’eventuale rifiuto verrà registrato anche in blockchain;
* L’acquirente troverà un biglietto abbinato alla sua richiesta con il relativo prezzo(**findMyPurchaseRequest**) e a questo punto può acquistare il biglietto (***buyTicket***). In questo punto viene creato il sigillo fiscale solo nel DB;
* Il rivenditore vedrà se il biglietto della PurchaseRequest è stata pagato oppure no, completerà la richiesta in maniera positiva (il biglietto viene emesso in blockchain ***mintNFT***) o negativa (il biglietto assegnato viene liberato). In entrambi i casi il risultato viene scritto anche in blockchain;
* Infine, il gestore dell’evento potrà chiudere l’evento (***closeEvent***), a questo punto gli acquirenti non potranno più fare nuove richieste. La chiusura è registrata anche in blockchain.

***-inserire descrizione classi e controlli***

Il codice sorgente completo è consultabile e scaricabile su GitHub all’indirizzo:

Nella stesura del codice sorgente si è cercato di seguire le linee guida **SEI CERT Oracle Coding Standard for Java**. Inoltre, abbiamo integrato nell’IDE Eclipse l’estensione SonarLint (prodotto da SonarSource) che identifica on the fly i possibili problemi nel codice.

Si è cercato di sanificare tutti gli input ed eseguire vari tipi di controlli. In particolare, l’utilizzo standard di Jpa (Java Persistence Api) garantisce che non possano essere sfruttate vulnerabilità di tipo Sql Injection.

Tra le varie classi, alcune delle quali andremo ad esplicare in seguito, abbiamo:

* Usato annotazioni **@NotNull** e **@Nullable** per distinguere campi e metodi che possono essere o non essere nulli;
* Dichiarato **final** tutti i campi che non devono essere modificati dopo la prima assegnazione;
* Dichiarato **final** tutte le classi e i metodi che non devono essere ereditati;
* Definito tramite annotazione **@Size,** un limite di caratteri minimo e massimo per i campi stringa dei **model**;
* Utilizzato il pacchetto **jackson-dataformat-xml** per la trasformazione degli oggetti in json response e quindi la gestione dei Rest controller;
* Utilizzato vari pacchetti di **Spring Boot**, tra i quali il pacchetto **JPA** per la gestione semi-automatica dei **repository**;
* Usato il pacchetto **web3j** per l’utilizzo delle classi **contract** e quindi l’interfacciamento con la blockchain.

La struttura delle cartelle che abbiamo utilizzato è la seguente:

* **aspect**: classi che l’utilizzano **AspectJ**;
* **auth**: classi per autenticazione e autorizzazione tramite **OAuth2** e **JWT**;
* **config**: classi di configurazione (con annotazione **@Configuration**);
* **contract**: classi autogenerate, rappresentanti i contract, tramite tool web3j da file solidity compilati;
* **controller**: classi contenenti i servizi Rest accessibili previa autenticazione/autorizzazione dall’esterno, contengono controlli specifici relativi al singolo servizio;
* **model**: classi entità (annotazione **@Entity**) che generano le relative tabelle nel database, la maggior parte dei controlli sono inseriti a questo livello in fase di istanziazione delle classi;
* **repository**: interfacce repository che ereditano da **JpaRepository** e che si occupano delle query verso il database;
* **service**: servizi interni del sistema, non utilizzabili esternamente.
* **view:** interfacce per la definizione di diversi profili di Json response (le informazioni vengono limitate nella response a seconda del servizio chiamato e del chiamante).

Per evitare una trattazione troppo lunga e prolissa ci limiteremo a trattare le entità, i contract implementati e i vari servizi REST dei controller.

**Entità Java**

I sorgenti delle entità in linguaggio Java sono nel percorso: **backend/src/main/java/it/univpm/eventchain/model**. Ci limiteremo a descrivere i campi e la funzionalità di ogni classe entità.

**Comment.java**

La classe modella il commento

* **id**: campo identificatore;
* **event**: evento a cui il commento fa riferimento;
* **body**: il corpo del commento;
* **sender**: l’autore del commento;
* **timestamp**: istante in cui il commento è stato creato/modificato.

**Ticket.java**

La classe modella il biglietto

* **id**: campo identificatore;
* **event**: evento a cui il biglietto fa riferimento;
* **price**: il prezzo in centesimi di euro del biglietto;
* **state**: lo stato del biglietto che comprende i valori (BUYABLE, BOOKED, PAYED, SELLED, VALIDATED);
* **ticketBuyer**: acquirente del biglietto;
* **ticketReseller**: venditore del biglietto;
* **type**: tipo descrittivo del biglietto;
* **taxSeal:** sigillo fiscale del biglietto;
* **tokenAddress:** indirizzo in blockchain dove è deployato lo smart contract **NFTicket**.

**Ticket.java**

La classe modella il biglietto

* **id**: campo identificatore;
* **event**: evento a cui il biglietto fa riferimento;
* **price**: il prezzo in centesimi di euro del biglietto;
* **state**: lo stato del biglietto che comprende i valori (BUYABLE, BOOKED, PAYED, SELLED, VALIDATED);
* **ticketBuyer**: acquirente del biglietto;
* **ticketReseller**: venditore del biglietto;
* **type**: tipo descrittivo del biglietto;
* **taxSeal:** sigillo fiscale del biglietto;
* **tokenAddress:** indirizzo in blockchain dove è deployato lo smart contract **NFTicket**.

**PurchaseRequest.java**

La classe modella la richiesta di acquisto

* **id**: campo identificatore;
* **event**: evento a cui la richiesta fa riferimento;
* **description**: aggiunta descrittiva della richiesta d’acquisto;
* **status**: lo stato della richiesta d’acquisto che comprende i valori (INIT, ACCEPTED, REFUSED, FAILED, SUCCEED);
* **ticketBuyer**: acquirente della richiesta di acquisto;
* **ticketReseller**: venditore della richiesta di acquisto;
* **prority**: numero che indica l’ordine di arrivo della richiesta (corrisponde al requestID dello smart contract **PurchaseRequestContract**);
* **paymentTime:** timestamp della riuscita del pagamento;
* **timestamp:** timestamp della creazione della richiesta di acquisto;
* **responseText:** risposta descrittiva alla richiesta da parte del rivenditore, utile in caso di rifiuto o fallimento;
* **failedPayment:** contatore dei tentativi di pagamento falliti.

**Member.java**

La classe modella l’utente

* **id**: campo identificatore;
* **username**: e-mail corrispondente all’account Google;
* **registrationId**: metodo di registrazione (nel nostro caso abbiamo previsto solo la registrazione tramite account Google);
* **name**: nome completo;
* **givenName**: nome;
* **familyName**: cognome;
* **picture**: link con immagine del profilo;
* **publicName:** nome con cui l’utente viene identificato dagli altri utenti (questa è l’unica informazione dell’utente visibile dagli altri utenti);
* **timestamp:** timestamp della creazione della richiesta di acquisto;
* **responseText:** risposta descrittiva alla richiesta da parte del rivenditore, utile in caso di rifiuto o fallimento;
* **authorities:** elenco dei ruoli che appartengono all’utente. I possibili ruoli sono USER, ADMIN, EVENT\_MANAGER, TICKET\_RESELLER, TICKET\_BUYER.

**Event.java**

La classe modella l’evento

* **id**: campo identificatore;
* **title**: titolo dell’evento;
* **description**: descrizione dell’evento;
* **start**: timestamp dell’inizio dell’evento;
* **end**: timestamp della fine dell’evento;
* **location**: campo descrittivo del luogo dell’evento;
* **eventManager**: gestore e creatore dell’evento;
* **opened:** booleano che indica se l’evento è ancora aperto.

**EventReseller.java**

La classe modella la relazione molti a molti tra evento e rivenditore

* **id**: campo identificatore;
* **event**: evento;
* **ticketReseller**: rivenditore;
* **accepted**: booleano che indica se il rivenditore è stato accettato per l’evento in oggetto.

**Smart contract in Solidity**

I sorgenti dei contract in linguaggio solidity sono stati inseriti nel percorso:

**backend/src/main/resources/contracts**

**BaseContract.sol**

Questo contract funge come una sorta di “classe base” che gli altri contract ereditano, contiene una funzione per la conversione di un bytes32 in string e tre modificatori per dei controlli generici su interi e stringhe.

**EventResellerContract.sol**

Questo contract rappresenta la relazione (analogo della classe model **EventReseller**) tra un rivenditore e un certo evento, nell’ipotesi che ci possano essere più rivenditori per ogni evento. Contiene i seguenti campi address:

* **targetEvent**: contiene l’indirizzo dove è deployato l’evento;
* **reseller**: contiene l’indirizzo pubblico dell’account Ethereum del reseller.

**NFTicket.sol**

Questo contract rappresenta il biglietto (analogo della classe model **Ticket**). Contiene i seguenti campi:

* **purchaseRequest**: contiene l’indirizzo dove è deployata la purchase request;
* **price:** il prezzo del biglietto;
* **taxSeal:** sigillo fiscale;
* **ticketType:** tipo di biglietto;
* **uri**: contiene un URI dove vengono visualizzate le info del biglietto.
* La funzione **addTicket** permette di aggiungere il biglietto all’evento definito nella **purchaseRequest** e richiama la funzione **successRequest** della **purchaseRequest**.
* La funzione **validCaller** controlla che il chiamante corrisponda al reseller della **purchaseRequest.**

**PurchaseRequestContract.sol**

Questo contract rappresenta la richiesta di acquisto (analogo della classe model **PurchaseRequest**). Contiene i seguenti campi:

* **ticketBuyer:** contiene l’indirizzo dell’account Ethereum dell’acquirente;
* **ticketEvent:** contiene l’indirizzo dove è deployato l'evento;
* **ticketReseller:** contiene l’indirizzo dell’account Ethereum del rivenditore;
* **description:** descrizione testuale della richiesta;
* **response:** risposta del rivenditore alla richiesta (utile in caso di rifiuto);
* **requestId**: Id che definisce la sequenza di arrivo delle richieste. Viene assegnato quando la richiesta viene aggiunta all’evento;
* **tokenId:** Id del token che viene generato se la richiesta viene completata;
* **status:** stato della richiesta che assume valori: INIT, FAILED, SUCCESS
* La funzione **addRequest** permette di aggiungere la richiesta all’evento.
* La funzione **successRequest** aggiorna lo stato della richiesta a SUCCESS e richiama le funzioni **addTicket** e **responseRequest** del **ticketEvent**, vienerichiamata dal **NFTicket**;
* La funzione **failedRequest** aggiorna lo stato della richiesta a FAILED e richiama la funzione **responseRequest** di **EventContract.**

**EventContract.sol**

In questo contract vengono importati altri due contract esterni che sono:

1. Il primo è un’implementazione dello standard ERC-721 per la creazione di NFT su Ethereum:

<https://github.com/0xcert/ethereum-erc721/src/contracts/tokens/nf-token-metadata.sol>

1. Il secondo fornisce un modificatore onlyOwner che permette solo al deployer del contratto di richiamare le specifiche funzioni definite con questo modificatore:

<https://github.com/0xcert/ethereum-erc721/src/contracts/ownership/ownable.sol>

L’EventContract rappresenta l'evento (analogo della classe model **Event**). Contiene i seguenti campi:

* **title:** titolo dell’evento;
* **start:** timestamp corrispondente all’inizio dell’evento;
* **end:** timestamp corrispondente alla fine dell’evento;
* **location:** descrizione testuale del luogo dell’evento;
* **status:** stato dell’evento che assume valori: OPENED,CLOSED,DELETED;
* **ticketCount:** conteggio dei ticket aggiunti all’evento;
* **requestCount:** conteggio delle purchaseRequest aggiunte all’evento;
* **nextResellerReqIds:** mappa all’indirizzo del rivenditore l’indice della prossima request ID da servire;
* **request:** mappa all’indirizzo dell’acquirente un booleano che indica se l’acquirente ha una richiesta pendente;
* **requestAddress:** mappa gli ID delle request agli indirizzi del PurchaseRequest;
* **resellerRequests**: mappa per ogni rivenditore la lista delle request ID in ordine di arrivo;
* **tickets**: mappa i token ID al relativo indirizzo del **NFTicket**;
* **ticketResellers**: mappa ad ogni indirizzo un booleano che indica se è un rivenditore per questo evento;
* La funzione **addTicket** viene richiamata dalla **PurchaseRequest**, controlla che il **NFTicket** non sia stato già inserito, scrive nei vari mapping dei ticket e richiama le apposite funzioni per la creazione del token NFT restituendo il tokenID;
* La funzione **addTicketReseller** viene richiamata dalla **EventReseller**, fa dei controlli di correttezza della richiesta e setta il mapping dei ticket reseller dell’evento**;**
* La funzione **addPurchaseRequest** viene richiamata dalla **PurchaseRequest**, fa dei controlli di correttezza della richiesta e setta i vari mapping della request, tra cui il mapping con l’array delle richieste del reseller;
* La funzione **responseRequest** viene richiamata dalla **PurchaseRequest**, fai dei controlli di correttezza della richiesta, sia che la richiesta sia andata a buon fine sia che la richiesta sia fallita o rigettata, fa avanzare l’indice alla prossima richiesta del reseller;
* Il modificatore **validResponseRequest** verifica che la request che il reseller vuole evadere sia quella dell’indice corrente**;**
* Si omette il commento delle altre funzioni perché già facilmente comprensibili dal codice sorgente.

**Servizi Rest dei controller Java**

I sorgenti dei controller in linguaggio Java sono nel percorso: **backend/src/main/java/it/univpm/eventchain/controller**

La classe GlobalDefaultExceptionHandler intercetta tutte le eccezioni (ad esempio di validazione) nei vari servizi restituendo una response HTTP ERROR Bad Request (400) con relativo messaggio di errore. Le altre possibili response eccezionali sono la HTTP ERROR Unauthorized (401) nel caso di utente non autenticato, la HTTP ERROR Forbidden (403) nel caso l’utente sia autenticato ma non ha il ruolo necessario per l’utilizzo della Rest. In caso di successo, i vari servizi restituiscono una response HTTP 200.

Si è pensato di inviare nelle response delle Rest solo lo stretto indispensabile per cercare di evitare che soggetti indesiderati e/o con cattive intenzioni vedano dati sensibili, per questo motivo sono state usate le annotazioni **@JsonIgnore** e **@JsonView**.

@JsonIgnore è usato per nascondere una certa proprietà (campo) di un oggetto Java, che quindi non verrà serializzata nella response del servizio Rest.

@JsonView serve per filtrare i campi in base al contesto della serializzazione. Abbiamo definito due tipi di profili JsonView: Public e Private, definiti nell’interfaccia View (**backend/src/main/java/it/univpm/eventchain/view**). Il profilo Public è il più restrittivo e mostra solo le proprietà definite Public, quasi tutti i servizi Rest usano questo profilo. Il profilo Private mostra sia le proprietà definite Public sia quelle definite Private.

Si è anche cercato di mettere in piedi delle procedure di recovery in caso di fallimento in punti critici nel codice. In particolare, abbiamo usato l’annotazione **@Transactional** per i metodi che hanno operazioni non divisibili, effettuando quindi il Rollback in caso di fallimento di una delle operazioni.

Abbiamo anche utilizzato l’annotazione **@Retryable** per il salvataggio nel database dopo scrittura in blockchain, cercando di mitigare il rischio in caso di temporanea mancanza di connessione a

Di seguito la trattazione dei vari controller:

* **CommentController**

**Servizi:** *update (HTTP PUT)/ delete (HTTP DELETE)/ find (HTTP GET)*

**Funzionalità:** aggiornamento/ cancellazione / ricerca di un commento

**Ruolo:** ROLE\_USER

**Controlli specifici*:*** solo il proprietario del commento può aggiornare / cancellare il commento

* **EventResellerController**

**Servizio:** *acceptEventReseller (HTTP PUT)*

**Funzionalità:**Accettazione da parte del gestore dell’evento di un nuovo rivenditore, scrive in blockchain utilizzando l’analoga funzione definita nell’**EventResellerContract**

**Ruolo:** ROLE**\_***EVENT\_MANAGER*

**Controlli specifici*:*** solo il proprietario dell’evento può accettare nuovi rivenditori e solo se l’evento è ancora aperto.

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** *createPurchaseRequest (HTTP POST)*

**Funzionalità:**Crea la richiesta di acquisto. La richiesta di acquisto viene anche deployata in blockchain (vedi **PurchaseRequestContract**).

**Ruolo:**  ROLE\_TICKET\_BUYER

**Controlli specifici*:*** L’eventReseller deve essere stato accettato. Se l’evento è chiuso o se l’acquirente ha già inserito una richiesta pendente per l’evento non permette di inserirne una nuova.

* **MemberController**

**Servizio:** user*(HTTP GET)*

**Funzionalità:**Restituisce le info personali (anche quelle Private) dell’utente loggato. È l’unico servizio Rest che restituisce anche i campi privati, ovvero con annotazione@JsonView**(View.Private.class)**

**Ruolo:**  ROLE\_*USER*

**Controlli specifici*:*** Ogni utente vede solo le proprie info.

* **TicketController**

**Servizio:** *find (HTTP GET)*

**Funzionalità:**Restituisce le info di uno specifico biglietto.

**Ruolo:**  ROLE\_*USER*

**Controlli specifici*:*** Se l’evento è chiuso solo il gestore dell’evento del biglietto, o l’acquirente del biglietto oppure il rivenditore del biglietto possono vedere le info.

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** findMyTicket *(HTTP GET)*

**Funzionalità:**Restituisce l’elenco dei ticket dell’acquirente.

**Ruolo:**  ROLE\_TICKET\_BUYER

**Controlli specifici*:*** Nessuno

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** resellerTickets *(HTTP GET)*

**Funzionalità:**Restituisce l’elenco dei ticket venduti dal rivenditore.

**Ruolo:**  ROLE\_TICKET\_RESELLER

**Controlli specifici*:*** Nessuno

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** *delete (HTTP DELETE)*

**Funzionalità:**Cancella uno specifico biglietto.

**Ruolo:**  ROLE\_EVENT\_MANAGER

**Controlli specifici*:*** Se il biglietto non è libero (non è stato prenotato o acquistato) può essere cancellato.

* **PurchaseRequestController**

**Servizio:** findMyPurchaseRequest *(HTTP GET)*

**Funzionalità:**Restituisce l’elenco delle richieste d’acquisto dell’acquirente.

**Ruolo:**  ROLE\_TICKET\_BUYER

**Controlli specifici*:*** Nessuno

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** resellerPurchaseRequests *(HTTP GET)*

**Funzionalità:**Restituisce l’elenco delle richieste d’acquisto assegnate al rivenditore.

**Ruolo:**  ROLE\_TICKET\_RESELLER

**Controlli specifici*:*** Nessuno

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** *acceptPurchaseRequest (HTTP PUT)*

**Funzionalità:**Il rivenditore accetta la richiesta di acquisto assegnandole un biglietto, il biglietto diventa prenotato.

**Ruolo:**  ROLE\_TICKET\_RESELLER

**Controlli specifici*:*** Viene controllato se in ordine di arrivo per lo specifico rivenditore è la richiesta da gestire (il rivenditore è vincolato a servire in ordine di arrivo le richieste)

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** refusePurchaseRequest *(HTTP PUT)*

**Funzionalità:**Il rivenditore rifiuta la richiesta, inserendo anche la spiegazione del rifiuto. Viene richiamata la funzione **failedRequest** del **PurchaseRequestContract**

**Ruolo:**  ROLE\_TICKET\_RESELLER

**Controlli specifici*:*** Come sopra

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** buyPurchaseRequest *(HTTP PUT)*

**Funzionalità:**L’acquirente acquista il biglietto effettuando il pagamento, se il pagamento fallisce viene registrato il fallimento in un contatore.

**Ruolo:**  ROLE\_TICKET\_BUYER

**Controlli specifici*:*** Viene controllato che l’acquirente sia lo stesso che ha fatto la richiesta

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** completePurchaseRequest *(HTTP PUT)*

**Funzionalità:**Se il pagamento è avvenuto correttamente viene richiamato il metodo **successRequest** del **PurchaseRequestContract** e quindi viene deployato il **NFTicket**, altrimenti il biglietto viene liberato e viene chiamato il **failedRequest** del **PurchaseRequestContract**

**Ruolo:**  ROLE\_TICKET\_RESELLER

**Controlli specifici*:*** Viene controllato se in ordine di arrivo per lo specifico rivenditore è la richiesta da gestire (il rivenditore è vincolato a servire in ordine di arrivo le richieste)

* **EventController**

**Servizi:** *create (HTTP POST) / update (HTTP PUT) / delete (HTTP PUT) / close (HTTP PUT)*

**Funzionalità:** *Creazione / aggiornamento / cancellazione/ chiusura di un evento. La creazione deploya il contract, gli altri servizi richiamano le analoghe funzioni dell’****EventContract***

**Ruolo:**  ROLE\_EVENT\_MANAGER

**Controlli specifici*:*** Solo il creatore dell’evento può aggiornare/chiudere/cancellare l’evento, l’evento può essere cancellato solo se non ci sono richieste di acquisto per l’evento.

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** *findMyEvents (HTTP GET)*

**Funzionalità:** *Ricerca dei propri eventi creati*

**Ruolo:**  ROLE\_EVENT\_MANAGER

**Controlli specifici*:*** Nessuno.

---------------------------------------------------------------------------------------------------------------------------

**Servizio:** *createTickets (HTTP POST) / findNewEventReseller (HTTP GET)*

**Funzionalità:** *Creazione dei biglietti / Ricerca dei nuovi potenziali rivenditori*

**Ruolo:**  ROLE\_EVENT\_MANAGER

**Controlli specifici*:*** Solo il creatore dell’evento può creare i biglietti per il proprio evento, solo lui può ricercare i nuovi potenziali rivenditori.

---------------------------------------------------------------------------------------------------------------------------

**Servizi:** *findActiveEvents (HTTP GET) / findInactiveEvents (HTTP GET) / / show (HTTP GET) / findTicketByEvents (HTTP GET) / createComment (HTTP GET) / showComments (HTTP GET)*

**Funzionalità:**Ricerca degli eventi attivi / Ricerca degli eventi non attivi / Ricerca di un evento specifico / ricerca biglietti di un evento / creazione di un commento a un evento / Ricerca dei commenti di un evento

**Ruolo:**  ROLE\_USER

**Controlli specifici*:*** Nessuno.

---------------------------------------------------------------------------------------------------------------------------

**Servizi:** createEventReseller *(HTTP POST) / resellerFindPurchaseRequest (HTTP GET)*

**Funzionalità:**Richiesta di diventare rivenditore di un evento, viene deployato l’EventResellerContract / Il rivenditore ricerca per un certo evento le richieste di acquisto a lui assegnate.

**Ruolo:**  ROLE\_TICKET\_RESELLER

**Controlli specifici*:*** Il rivenditore non può creare richieste multiple per un certo evento / Il rivenditore può vedere solo le richieste di acquisto assegnate a lui

---------------------------------------------------------------------------------------------------------------------------

**Servizi:** *findEventReseller (HTTP GET) / findMyPurchaseRequest (HTTP GET)*

**Funzionalità:**Ricerca di rivenditori approvati per un certo evento / Ricerca delle proprie richieste di acquisto

**Ruolo:**  ROLE\_TICKET\_BUYER

**Controlli specifici*:*** Nessuno.