

20/01/2022



# ticketTwo

Relazione completa sulla  
realizzazione della web app per il  
progetto relativo al corso di  
Software Cybersecurity

Cucchieri Giacomo, Cuicchi Manila,  
De Bartolomeo Gabriele, Francalancia Simone,  
Pierigè Giacomo

UNIVERSITÀ POLITECNICA DELLE MARCHE  
LM INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE  
AA 2020-2021

## Sommario

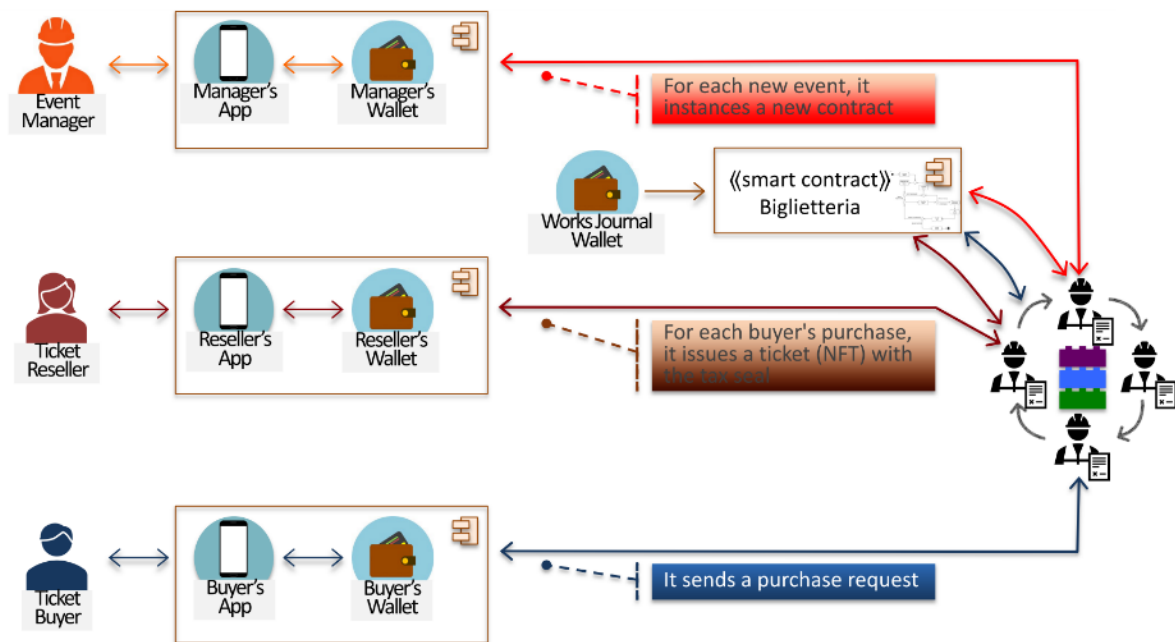
INTRODUZIONE .....	4
INGEGNERIA DEI REQUISITI .....	5
Early Requirements Analysis .....	5
Strategic Dependency Model .....	5
Strategic Rationale Model .....	6
Late Requirements Analysis .....	7
Strategic Dependency Model .....	7
Strategic Rationale Model .....	8
Individuazione degli asset .....	9
Asset table e policy di sicurezza .....	11
Casi d'uso .....	12
Casi di misuse .....	18
Casi d'abuso .....	21
Alberi d'attacco .....	26
Applicazione del metodo STRIDE .....	29
DESIGN .....	31
Architettura software .....	31
Scelte tecnologiche .....	32
Scelta della blockchain .....	32
Scelta del linguaggio di programmazione .....	32
Scelta del sistema operativo .....	33
Scelta del database .....	33
Scelta del sistema di pagamento esterno .....	34
Comunicazione sicura tra client e server .....	34
Scansione QR code .....	34
Sigillo fiscale .....	35
Autenticazione .....	35
Accountability .....	36
Gestione dei segreti .....	36
Protezione .....	37
Protezione a livello piattaforma .....	37
Protezione a livello applicazione .....	37
Protezione a livello record .....	37
Conformità alle linee guida di progettazione sicura .....	38
IMPLEMENTAZIONE DEL FRONTEND DELL'APPLICAZIONE WEB .....	42

Il Frontend.....	42
Struttura del codice .....	44
HTML elements .....	45
widget.js.....	45
HTMLpage.js.....	46
Elementi grafici .....	46
Form personalizzati .....	49
Scripts.....	53
Web pages .....	55
areaRiservata.js.....	55
IMPLEMENTAZIONE DEL BACKEND DELL'APPLICAZIONE WEB.....	56
Introduzione al backend e struttura .....	56
Librerie utilizzate e package.json .....	57
app.js.....	57
Database .....	59
MODELLI .....	61
Descrizione del modello <b>User</b> .....	61
Descrizione del modello <b>Event</b> .....	63
Descrizione del modello <b>Ticket</b> .....	66
Descrizione del modello <b>Receipt</b> .....	68
Descrizione del modello <b>OTP</b> .....	70
Descrizione del modello <b>Access</b> .....	71
Analisi dettagliata di uno schema (Access).....	72
CONTROLLERS.....	75
Descrizione del controller <b>UserController</b> .....	76
Descrizione del controller <b>AuthController</b> .....	79
Descrizione del controller <b>EventController</b> .....	82
Descrizione del controller <b>TicketController</b> .....	85
Descrizione del controller <b>ReceiptController</b> .....	88
MIDDLEWARES.....	89
Descrizione del middleware <b>verifyToken</b> .....	89
Descrizione del middleware <b>checkLogin</b> .....	89
Descrizione del middleware <b>checkPrivileges</b> .....	90
Descrizione del middleware <b>verifyOTP</b> .....	90
SMART CONTRACT.....	91
Lo Smart Contract.....	91

Descrizione degli attributi del contratto.....	92
Descrizione del costruttore .....	93
Functions .....	95
generateRandomPassword.js.....	95
checkPassword.js .....	95
wallet.js.....	95
contract.js.....	96
query.js.....	97
ticket.js.....	100
paypal.js .....	100
getToken.js.....	101
mailer.js.....	101
timeFunctions.js.....	102

## INTRODUZIONE

Nella presente relazione vengono mostrate le caratteristiche tecniche di progettazione e di implementazione di un'applicazione web con funzione di biglietteria aperta alla vendita online, denominata **ticketTwo**, che prevede l'interazione con una blockchain dedicata, secondo lo schema della figura seguente.



*Figura 1 Schema della biglietteria online basata su blockchain*

In particolare, viene fornita una descrizione delle scelte effettuate dal punto di vista della sicurezza informatica nello sviluppo e nell'utilizzo del software, seguendo le linee guida indicate nel corso di "Software Cybersecurity" per l'A.A. 2020-2021.

# INGEGNERIA DEI REQUISITI

In questa prima fase della realizzazione della nostra web app basata su blockchain, si è cominciato con l'individuare quali sono i possibili requisiti funzionali e non funzionali che il software dovrà rispettare. Per requisiti funzionali, si intendono tutte quelle specifiche volte a definire le azioni possibili e i vincoli che possono essere posti al nostro software, mentre, per requisiti non funzionali, si intendono tutte quelle caratteristiche che non vengono richieste esplicitamente da un possibile cliente, ma che sono comunque necessarie ed influenzano il lavoro degli sviluppatori, permettendo di descrivere **come** il sistema svolge certi compiti/azioni.

Come primo passo siamo andati ad utilizzare un linguaggio complementare a quello UML, chiamato **i\* (i-star)**, che pone l'attenzione nell'analizzare in particolare i requisiti di sistema, lasciando successivamente al diagramma dei casi d'uso il compito di rappresentare il **design** vero e proprio del sistema.

## Early Requirements Analysis

In questa **fase di analisi iniziale**, detta Early Requirements Analysis, abbiamo cercato di individuare quali sono i processi del nostro sistema software che vanno automatizzati, senza supporre che ci sia un sistema software.

In altre parole, si tratta di modellare il problema da affrontare senza fare ipotesi su quale sarà lo strumento impiegato per implementarlo. È un'analisi che pone l'attenzione sull'ambiente, cioè sul dominio applicativo nel quale gli attori più rilevanti per il nostro sistema e le loro dipendenze si trovano ad operare.

Durante l'attività di individuazione degli attori principali, dei loro obiettivi e della ricerca delle rispettive dipendenze che influenzano il sistema, abbiamo impiegato un **approccio agile**; quindi, alcuni cambiamenti fondamentali sono stati individuati nel corso della costruzione dei diagrammi (che rappresentano i vari scenari possibili), e ciò ha comportato delle modifiche progressive che sono state apportate nella realizzazione dei diagrammi seguenti. Questo a giustificare le diversità che possono emergere tra i vari diagrammi realizzati.

## Strategic Dependency Model

Nella Figura 2 si notano quali sono gli attori rilevanti del nostro sistema e le loro dipendenze che verranno poi automatizzate.

Gli attori principali sono:

- l'Event Manager;
- il Cliente;
- la Biglietteria;
- l'Annullatore dei biglietti;
- il Sistema di pagamento (considerato come attore esterno al sistema).

Le varie dipendenze individuate, che si colgono leggendo il diagramma, sono:

- la Biglietteria, per emettere i biglietti al Cliente che li acquista, deve ottenere le informazioni necessarie dall'organizzatore dell'evento, soprattutto per quanto riguarda i posti totali disponibili;
- l'acquisto dipende dal servizio di pagamento (esterno al sistema), cioè dal suo corretto funzionamento e disponibilità. Di conseguenza, anche la ricezione di avvenuto pagamento dipende da quest'ultimo, che è fondamentale per la Biglietteria per confermare il pagamento, emettere il biglietto e poterlo, così, inviare al Cliente;
- il giorno dell'evento, l'Annullatore deve avere a disposizione l'elenco di tutti i biglietti emessi dalla Biglietteria;
- l'esperienza di acquisto del biglietto, valutata dal Cliente, dipende esclusivamente dalla gestione delle vendite da parte dello staff della Biglietteria;
- la soddisfazione complessiva del Cliente dipende dalla qualità dell'evento, quindi è responsabilità dell'Event Manager curare l'evento stesso nei dettagli.

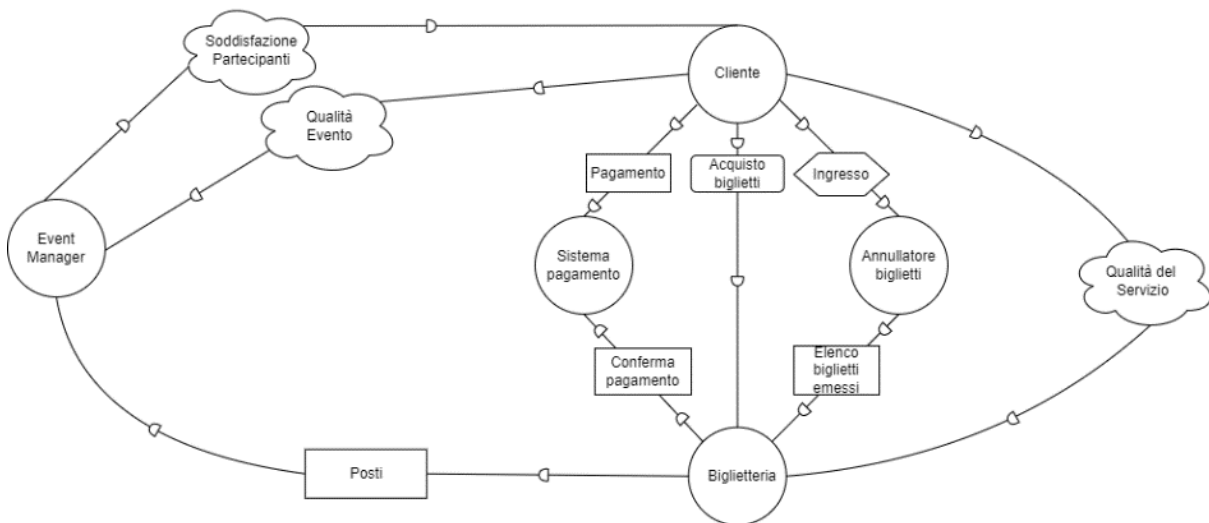


Figura 2 Early Requirements Analysis, Strategic Dependency Model

## Strategic Rationale Model

Dopo aver individuato lo scenario nel quale gli attori interagiscono, e indicato le relazioni di dipendenza, possiamo soffermarci sugli obiettivi e sotto-obiettivi di ogni attore del sistema.

Questo passaggio viene svolto attraverso l'**AND-OR DECOMPOSITION** (cioè una decomposizione congiuntiva o disgiuntiva), che ci permette di scomporre l'obiettivo principale in sotto-obiettivi, detti task o attività, per rendere più chiare le attitudini, ovvero le azioni possibili e i relativi vincoli entro cui si possono svolgere.

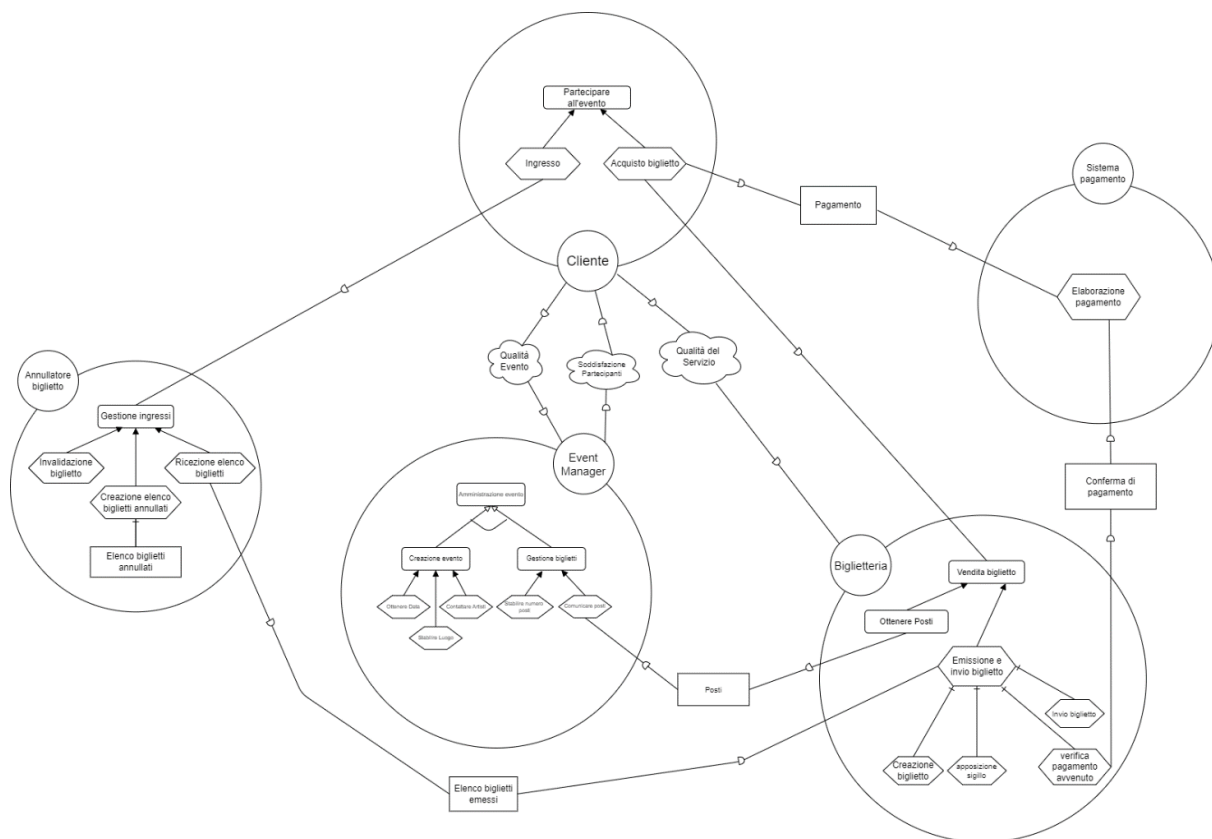


Figura 3 Early Requirements Analysis, Strategic Rationale Model

## Late Requirements Analysis

In questa fase si è cominciato a ragionare sul software, cioè a delineare un modello più chiaro di come il sistema deve agire nella realtà. Si è trattato principalmente di riconoscere il vero supporto che il sistema software può dare nel dominio, ossia nello scenario che abbiamo finora descritto.

Fino a questo momento si è sicuramente compreso che il software è una piattaforma in cui tutti gli attori interagiscono e che raccoglie le informazioni necessarie che permettono le loro interazioni.

## Strategic Dependency Model

Nel diagramma di Figura 4, ponendo al centro dell'attenzione il sistema, sono state evidenziate maggiormente quali sono le dipendenze tra gli attori e il sistema stesso.

Partendo dagli attori principali, abbiamo definito quali sono le azioni da loro compiute con il software che verrà realmente implementato. Ciò ha permesso successivamente di avere una visione immediata dei moduli che comporranno il nostro sistema.



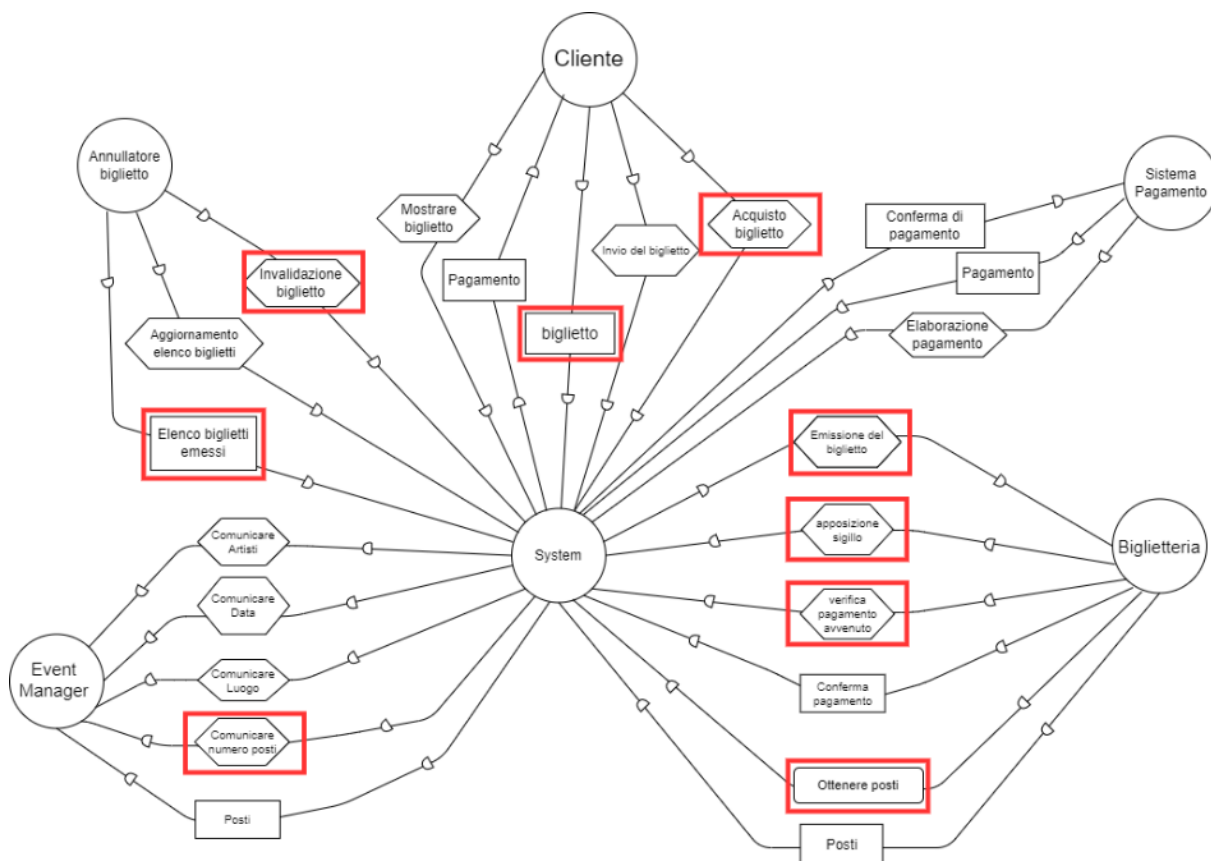


Figura 4 Late Requirements Analysis, Strategic Dependency Model

## Strategic Rationale Model

In questo diagramma, definito Strategic Rationale Model, emergono non solo le attitudini di ogni attore che dipendono dal sistema, ma anche le attitudini che il software dovrà possedere, cioè le attività e le risorse che esso dovrà **gestire** realmente.

Nel nostro caso notiamo come l'amministrazione generale dell'evento si articola in sotto-obiettivi principali, quali:

- gestione dell'evento;
- gestione del biglietto;
- gestione dell'ingresso.

Ognuno dei sotto-obiettivi elencati qui sopra si suddivide in ulteriori task e risorse, alle quali fa riferimento per svolgere al meglio le attività che deve eseguire.

Un'attenzione particolare va posta alla gestione del biglietto, che si divide ulteriormente in altri due sotto-obiettivi: la gestione della vendita del biglietto e la gestione dell'emissione del biglietto; il fine ultimo è rendere più chiara la visione di come sono composti questi ultimi, ma soprattutto di poterli implementare al

meglio senza tralasciare aspetti importanti che, se non implementati correttamente nella fase dedicata, potrebbero generare problemi e malfunzionamenti dell'intero sistema.

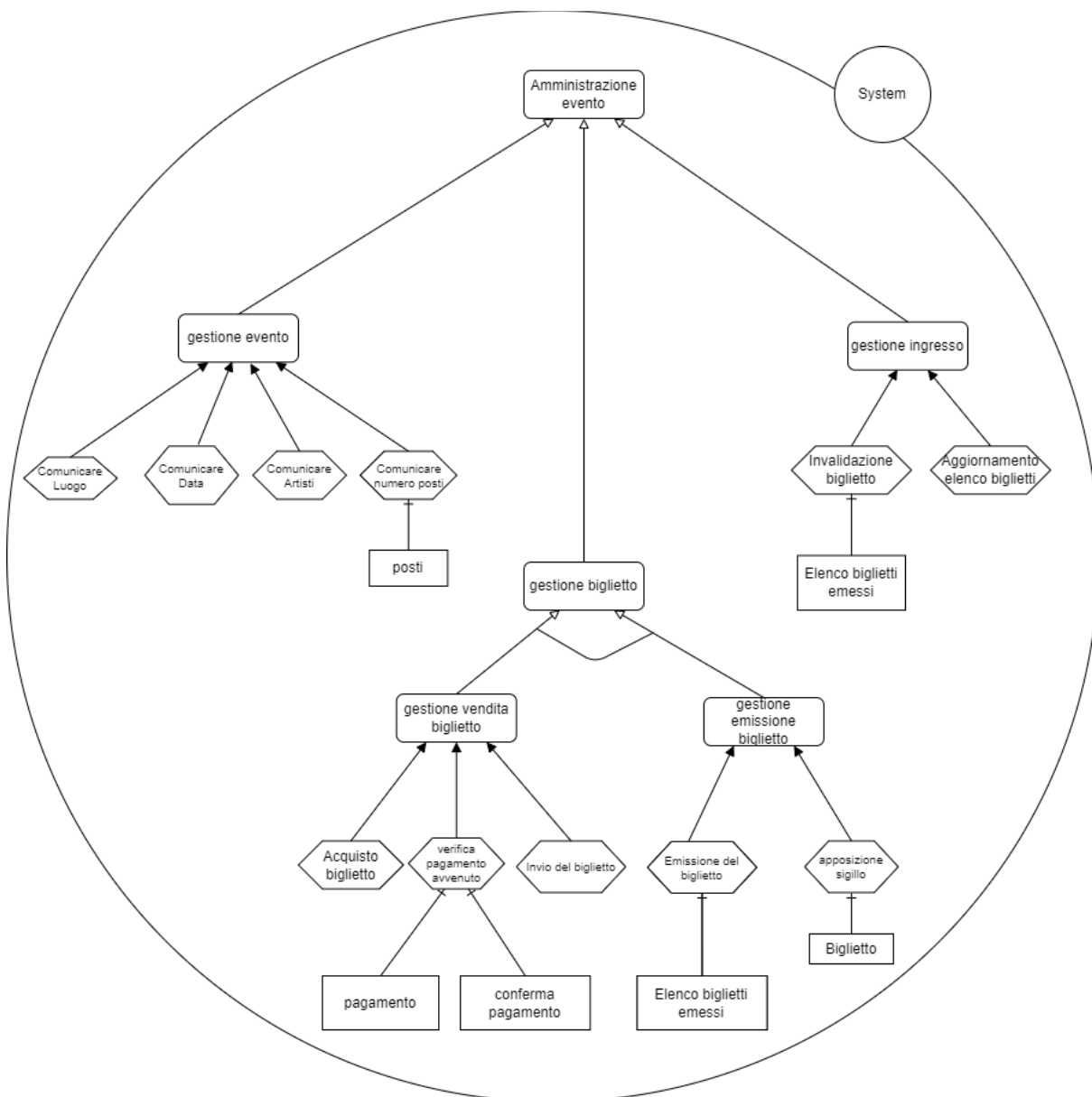


Figura 5 Late Requirements Analysis, Strategic Rationale Model

## Individuazione degli asset

In questa fase sono posti in evidenza i moduli del sistema, con i rispettivi obiettivi e loro suddivisione, andando ad evidenziare (con il rettangolo in rosso) gli asset critici del nostro software.

Per **asset** intendiamo quelle componenti sulla cui gestione viene posta grande attenzione, cioè le componenti principali del sistema stesso. Una amministrazione errata di tale asset potrebbe generare un funzionamento non corretto dei servizi offerti dal prodotto (software) finale.

Dal diagramma si evince che gli asset critici sono:

- il biglietto;
- invalidazione biglietto;
- lettura biglietto;
- elenco biglietti emessi;
- acquisto biglietto;
- verifica pagamento avvenuto;
- invio del biglietto.

Il biglietto è l'asset maggiormente critico, in quanto è l'oggetto principale dell'intero sistema che stiamo realizzando ed è fondamentale che rispetti in modo preciso le specifiche date; in particolare deve essere facilmente verificabile e non falsificabile.

Allo stesso modo, è importante che anche l'invalidazione avvenga in maniera corretta e questo è responsabilità dell'attore Annullatore biglietto, rispettando precisamente le attitudini legate alla sua mansione.

Gli asset restanti, ovvero lettura biglietto, elenco biglietti emessi, acquisto biglietto, verifica pagamento avvenuto e invio del biglietto, sono altrettanto critici e dipendono in parte dal corretto funzionamento dell'attore esterno al sistema, ovvero il sistema di pagamento.

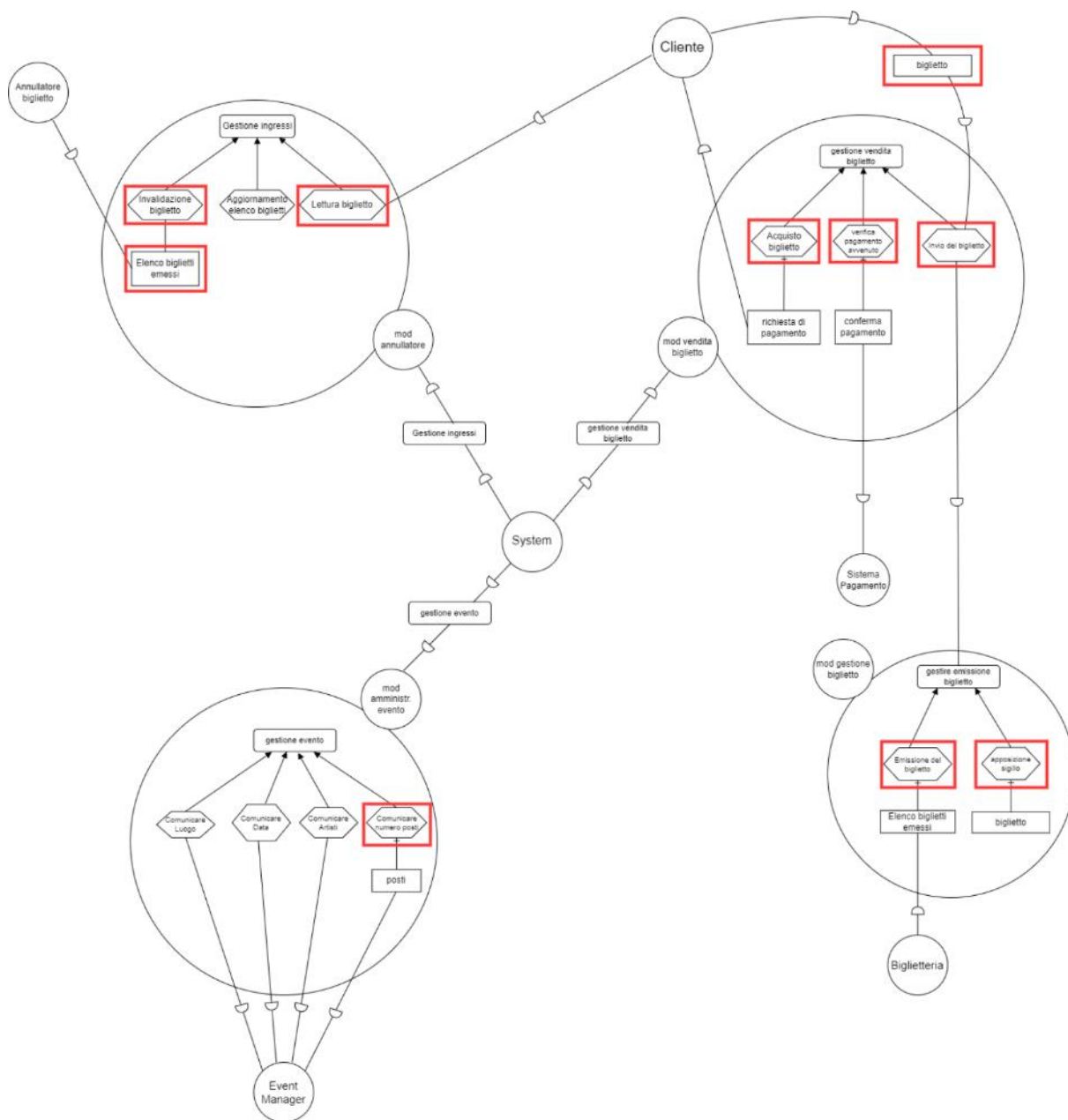


Figura 6 Individuazione degli asset critici

## Asset table e policy di sicurezza

Per ciascuno degli asset critici individuati si è effettuata un'analisi volta a individuare: le policy di sicurezza messe a rischio in caso di attacco, le policy organizzative che il sistema deve garantire, i valori di importanza dell'asset, nonché la gravità di un suo eventuale malfunzionamento e l'impatto dello stesso sulla corretta funzionalità dell'intero sistema, secondo quanto illustrato in Tabella 1 (si veda il file Asset.xlsx per una migliore visualizzazione).

RISULT	POLICY DI SICUREZZA	POLICY ORGANIZZATIVE	VALORE	IMPATTO
Comunicare numero posti	1) Integrità 2) Autenticità 3) Garanzia	1) Numero dei posti illustrato nella trasmissione 2) Numero dei posti proveniente da una fonte sicura 3) Autenticazione a comunicare il numero dei posti	1. Valore basso, necessario per stabilire un limite ai biglietti disponibili.	2. Impatto medio-basso in quanto il problema è facilmente risolvibile. Comporta il rischio di stampare un numero maggiore o minore di biglietti rispetto ai posti effettivamente disponibili.
Acquisto biglietto	1) Responsabilità 2) Affidabilità 3) Resilienza (evidenziata non implementata) 4) Disponibilità	1) La biglietteria e il cliente non possono negare di aver eseguito la transazione (non ripudio). 2) L'utente deve essere certo di non perdere gli euro del biglietto durante la transazione. 3) Il nostro sistema protegge le eccezioni: l'acquisto del biglietto funziona nonostante un eventuale attacco, oppure il sistema annulla la richiesta se un passaggio durante il processo di acquisto.	5. Valore alto, asset indispensabile per i clienti e la biglietteria. Permette al cliente di ottenere il biglietto.	4. Impatto medio-alto in quanto potrebbe causare danni economici ai clienti in caso di perdita di denaro e reputazione in caso di rallentamento del sistema di pagamento esterno.
Verifica pagamento avvenuto	1) Disponibilità 2) Autenticità 3) Integrità	1) Il sistema deve essere sempre disponibile per rilasciare la ricevuta di avvenuto pagamento. 2) Autenticità della ricevuta. 3) Ricevuta di avvenuto pagamento non compromessa.	4. Valore medio-alto, senza ricevuta di pagamento il cliente non è in grado di ricevere/acquistare il biglietto.	2. Impatto medio-basso, comporta problemi di reputazione. Non comporta problemi economici in quanto la responsabilità del danno appartiene al sistema di pagamento esterno.
Emissione del biglietto	1) Disponibilità 2) Responsabilità 3) Affidabilità 4) Resilienza 5) Garanzia	1) Appena concluso l'acquisto, il sistema deve rendersi subito disponibile per erogare il biglietto. 2) La biglietteria non può negare di aver inviato il biglietto. 3-5) Il sistema garantisce la corretta erogazione del biglietto. 4) Se accade una interruzione del servizio, il sistema garantisce che il biglietto venga emesso.	3. Valore medio, il biglietto senza sigillo non ha valore.	4. Impatto medio-alto, l'interruzione del servizio comporta l'impossibilità di procedere con l'apposizione del sigillo e quindi della vendita del biglietto.
Apposizione sigillo	1) Integrità 2) Disponibilità 3) Autenticità	1) Apposizione del sigillo associata correttamente al biglietto. 2) Appena conclusa l'emissione, la biglietteria sarà subito disponibile per l'apposizione sigillo. 3) Autenticità del sigillo verificata tramite firma digitale.	5. Valore alto, senza sigillo i clienti non hanno valore e non sono vendibili.	5. Impatto elevato, senza sigillo il biglietto non è autentico e potrebbe causare danni economici, tecnici e di reputazione.
Digietto	1) Disponibilità 2) Autenticità 3) Integrità	1) Il cliente sarà in grado di accedere in un qualunque momento alla ricerca tramite l'apposita piattaforma. 2-3) Il sigillo serve per garantire l'integrità e l'autenticità del biglietto stesso.	5. Valore elevato, è l'asset principale del sistema.	5. Impatto altissimo: elevato, senza biglietto il cliente non sarà in grado di accedere e partecipare all'evento.
Inizio del biglietto	1) Disponibilità 2) Integrità	1) Il sistema deve essere sempre disponibile per iniziare il biglietto. 2) Durante l'inizio del biglietto, la ricerca è protetta da modifiche.	2. Valore medio-basso, se si presenta un guasto, appena il sistema tornerà disponibile inizierà il biglietto all'acquisto.	2. Impatto medio, riguarda principalmente la reputazione dell'organizzazione e la soddisfazione del cliente per la tempestività d'inizio.
Lettura biglietto	1) Disponibilità 2) Garanzia	1) La disponibilità del servizio viene garantita anche tramite ridondanza. 2) Solo chi è autorizzato può accedere ai dati del biglietto in fase di lettura.	2. Valore medio, la lettura del biglietto può avvenire tramite modi alternativi per far fronte ad eventuali guasti.	4. Impatto medio-alto, l'eventuale mal funzionamento potrebbe causare eventuale inaccessibilità del cliente in termini di tempistiche di attesa.
Invalutazione biglietto	1) Disponibilità 2) Garanzia	1) La disponibilità del servizio viene garantita anche tramite ridondanza. 2) Solo chi è autorizzato può accedere ai dati del biglietto in fase di invalidazione.	5. Valore alto, azione necessaria per l'entrata e per i controlli stas.	5. Impatto elevato, senza invalidazione si potrebbe utilizzare il biglietto per ulteriori ingressi e/o vendite illegali.
Elenco biglietti emessi/annullati	1) Disponibilità 2) Integrità 3) Garanzia	1) L'elenco dei biglietti sarà disponibile in un qualsiasi momento. 2-3) L'elenco sarà disponibile in sola scrittura lettura per l'attore biglietteria e l'attore annullatore, mentre sarà disponibile in sola lettura per tutti gli attori.	4. Valore medio-alto, la consultazione è necessaria per la fase di invalidazione del biglietto e quindi di ingresso all'evento.	4. Impatto medio-alto, senza l'elenco non si può tenere traccia degli ingressi validi.

Tabella 1 (si veda il file Asset.xlsx per una migliore visualizzazione)

Successivamente si è passati allo studio dei possibili casi di interazione tra l'utente e il sistema, o tra varie parti del sistema, distinguendo in particolare quelli che sono i Casi d'uso, i Casi di Misuso e i Casi di Abuso, di cui si fornisce una descrizione puntuale nei seguenti paragrafi.

## Casi d'uso

Descrivono i possibili scenari previsti per il "normale utilizzo" del sistema (ossia seguendo un comportamento non malevolo o accidentale). Sono rappresentati secondo la sequenza ordinata delle azioni di cui si compongono, le quali definiscono l'interazione tra l'utente e il sistema o tra varie componenti del sistema.

Use case: Comunicare numero posti	
Actors	Event manager, Biglietteria
Description	L'organizzatore dell'evento comunica appena possibile alla biglietteria il numero di posti disponibili totali per il luogo in cui sarà ospitato l'evento
Data	Posti: numero dei posti (intero)
Stimulus and preconditions	L'event manager ha trovato luogo, data ed artisti per l'evento
Basic Flow	1.L'event manager dopo aver creato un evento stabilisce il numero dei posti 2.L'event manager comunica il numero dei posti alla biglietteria tramite il modulo apposito
Alternate Flow	---
Exception Flow	1.L'event manager invia il numero dei posti alla biglietteria 2.La biglietteria non riceve l'informazione
Response and postconditions	- Conferma avvenuta ricezione dati, - Dati ricevuti correttamente dalla biglietteria.
Non Functional requirements	Integrità, autenticità, garanzia
Comments	L'event manager deve essere autorizzato ad inviare il dato, la biglietteria deve avere i necessari permessi per la lettura

Tabella 2

Use case: Acquisto biglietto	
Actors	Biglietteria, Cliente, Sistema pagamento
Description	Interazione tra un cliente e la biglietteria per l'acquisto di un biglietto
Data	Pagamento, Conferma di pagamento
Stimulus and preconditions	Il cliente vuole partecipare all'evento e deve avere un wallet con i soldi necessari al pagamento. Sistema di pagamento disponibile
Basic Flow	1. Il cliente avvia la transazione per l'acquisto del biglietto di un evento dall'apposito sito. 2. La biglietteria interroga il sistema di pagamento in seguito alla richiesta di acquisto da parte del cliente. 3. Il sistema di pagamento emetterà una ricevuta necessaria per procedere con l'acquisto.
Alternate Flow	---
Exception Flow	1. Il cliente avvia la transazione per l'acquisto del biglietto di un evento dall'apposito sito Eccezione A: Il cliente non dispone della somma necessaria per procedere all'acquisto Eccezione B: Il sistema di pagamento non è disponibile 2. In entrambi i casi il sistema rifiuta la transazione e viene visualizzato un messaggio di errore
Response and postconditions	Ricevuta la conferma del pagamento la biglietteria procede all'emissione e invio del biglietto
Non Functional requirements	Responsabilità, affidabilità, resilienza, safety, disponibilità
Comments	Il sistema di pagamento è esterno al sistema implementato

Tabella 3



Use case: Verifica pagamento avvenuto	
Actors	Biglietteria, Sistema di pagamento
Description	Il sistema di pagamento dopo l'elaborazione del pagamento invia una conferma di pagamento avvenuto alla biglietteria
Data	Conferma di pagamento
Stimulus and preconditions	Il cliente ha versato la quota necessaria per l'acquisto Sistema di pagamento disponibile
Basic Flow	<ol style="list-style-type: none"> <li>1. Il sistema di pagamento riceve la quota</li> <li>2. Il sistema di pagamento elabora il pagamento</li> <li>3. Il sistema di pagamento emette ed inoltra la ricevuta alla biglietteria</li> </ol>
Alternate Flow	---
Exception Flow	<ol style="list-style-type: none"> <li>1. La biglietteria non riceve la conferma di pagamento.</li> <li>2. Viene visualizzato un messaggio di errore</li> </ol>
Response and postconditions	La biglietteria riceve la conferma di pagamento
Non Functional requirements	Disponibilità, Autenticità, Integrità
Comments	---

Tabella 4

Use case: Emissione del biglietto	
Actors	Biglietteria
Description	La biglietteria genera il biglietto senza il sigillo fiscale
Data	Biglietto senza sigillo, Elenco biglietti emessi
Stimulus and preconditions	La biglietteria ha ricevuto la conferma di pagamento dal sistema di pagamento
Basic Flow	<ol style="list-style-type: none"> <li>1. Generazione biglietto</li> <li>2. Inserimento dati relativi all'evento sul biglietto (luogo, data, artista)</li> </ol>
Alternate Flow	---
Exception Flow	---
Response and postconditions	Aggiornato il biglietto con i dati necessari verrà apposto il sigillo
Non Functional requirements	Disponibilità, Responsabilità, Affidabilità, Resilienza, Garanzia
Comments	---

Tabella 5

Use case: Apposizione sigillo	
Actors	Biglietteria
Description	La biglietteria appone il sigillo fiscale sul biglietto
Data	Sigillo, biglietto
Stimulus and preconditions	Biglietto senza sigillo correttamente generato
Basic Flow	<ol style="list-style-type: none"> <li>1. Ricezione biglietto dal modulo</li> <li>2. Calcolo hash biglietto</li> <li>3. Apposizione sigillo fiscale</li> </ol>
Alternate Flow	---
Exception Flow	---
Response and postconditions	Il biglietto diventa valido per l'invio al cliente
Non Functional requirements	Integrità, Disponibilità, Autenticità
Comments	---

Tabella 6

Use case: Biglietto	
Actors	Biglietteria, Cliente
Description	Il biglietto, con i dati necessari, permette al cliente di accedere all'evento
Data	Biglietto con sigillo fiscale
Stimulus and preconditions	Il biglietto deve essere valido (con sigillo)
Basic Flow	<ol style="list-style-type: none"> <li>1. Richiesta di acquisto e pagamento</li> <li>2. Generazione ed apposizione del sigillo fiscale sul biglietto</li> <li>3. Invio del biglietto al cliente</li> </ol>
Alternate Flow	---
Exception Flow	Vedere exception flow, specifiche degli asset "acquisto biglietto", "verifica pagamento", "emissione biglietto"
Response and postconditions	Il cliente ottiene un biglietto valido per accedere all'evento
Non Functional requirements	Disponibilità, Autenticità, Integrità
Comments	---

Tabella 7



Use case: Invio del biglietto	
Actors	Biglietteria, Cliente
Description	La biglietteria invia una email con annesso biglietto
Data	Biglietto
Stimulus and preconditions	Il biglietto deve avere il sigillo fiscale
Basic Flow	La biglietteria invia il biglietto al cliente
Alternate Flow	---
Exception Flow	1. Il cliente non riceve/elimina la mail con il biglietto 2. Il cliente può scaricarlo comunque dall'area riservata
Response and postconditions	Il cliente ottiene un biglietto valido per accedere all'evento
Non Functional requirements	Disponibilità, Integrità
Comments	---

Tabella 8

Use case: Lettura biglietto	
Actors	Annullatore biglietto, Cliente
Description	Il cliente mostra il biglietto che sarà letto dall'annullatore all'ingresso
Data	Biglietto con sigillo, elenco biglietti emessi/annullati
Stimulus and preconditions	Il cliente dispone di un biglietto valido
Basic Flow	1. Il cliente mostra il biglietto all'annullatore all'ingresso 2. L'annullatore legge il biglietto
Alternate Flow	Verifica manuale della validità del biglietto
Exception Flow	1. Possibile malfunzionamento del lettore di QR code 2. L'annullatore sostituisce il lettore difettoso con uno funzionante
Response and postconditions	Biglietto invalidato
Non Functional requirements	Disponibilità, Garanzia
Comments	L'annullatore procede alla lettura tramite un lettore di QR code connesso al sistema

Tabella 9

Use case: Invalidazione biglietto	
Actors	Annullatore biglietto, Cliente
Description	L'annullatore invalida il biglietto appena letto
Data	Biglietto con sigillo, elenco biglietti emessi/annullati
Stimulus and preconditions	Biglietto letto con successo
Basic Flow	1. L'annullatore invalida il biglietto 2. Aggiornamento dell'elenco dei biglietti emessi/annullati
Alternate Flow	---
Exception Flow	---
Response and postconditions	il biglietto viene annullato, l'elenco biglietti/annullati viene aggiornato, quindi il cliente può entrare all'evento
Non Functional requirements	Disponibilità, Garanzia
Comments	

Tabella 10

Use case: Elenco biglietti emessi/annullati	
Actors	Event manager, Biglietteria, Annullatore biglietto
Description	L'elenco tiene traccia dei biglietti emessi e annullati mediante un database. Ad ogni biglietto emesso corrisponde un campo della tabella che mostra se il biglietto è valido o meno.
Data	Biglietto con sigillo, elenco biglietti emessi/annullati
Stimulus and preconditions	Creazione tabella dei biglietti emessi/annullati nel database
Basic Flow	1 Emissione del biglietto 2 Creazione record associato al biglietto nel database 3.a Lettura ed invalidazione del biglietto 3.b Aggiornamento campo booleano di validità del biglietto nel database
Alternate Flow	---
Exception Flow	1. Biglietto non trovato nell'elenco 2. Il sistema restituisce un messaggio di errore
Response and postconditions	Elenco biglietti emessi/annullati aggiornato
Non Functional requirements	Disponibilità, Integrità, Garanzia
Comments	Durante l'operazione di invalidazione del biglietto il campo booleano relativo alla validità del biglietto passerà da true a false (il campo rappresenta lo stato della validazione del biglietto)

Tabella 11

## Casi di misuso

Descrivono i possibili scenari previsti quando si verifica un utilizzo del sistema che non è corretto o consentito, ma è tuttavia privo di intenzioni malevole. Sono rappresentati secondo la sequenza ordinata delle azioni di cui si compongono, le quali definiscono l'interazione tra l'utente e il sistema o tra varie componenti del sistema.

Misuse case: Comunicare numero posti	
Actors	Event manager maldestro, Biglietteria
Description	L'organizzatore dell'evento maldestro sbaglia a comunicare o cancellare il numero dei posti e/o dettagli dell'evento.
Data (asset)	Posti: numero dei posti (intero)
Stimulus and Preconditions	I dati sono già stati inseriti e l'event manager sta effettuando altre operazioni manuali.
Attack 1 Flow	Modifica accidentale del numero dei posti e/o dettagli dell'evento
Attack 2 Flow	Cancellazione accidentale del numero dei posti e/o dettagli dell'evento
Response and postconditions	Si è comunicato un numero errato di posti disponibili.
Mitigations	
Non Functional requirements	

Tabella 12

Misuse case: Acquisto biglietto	
Actors	Biglietteria, Utente Maldestro, Sistema di pagamento, Attaccante
Description	L'utente maldestro diventa vittima di furto di dati.
Data (asset)	Pagamento, Conferma di pagamento, Credenziali del sistema di pagamento
Stimulus and preconditions	L'utente maldestro gestisce in modo non corretto le proprie credenziali.
Attack 1 Flow	Il cliente inserisce i dati in un sito di pagamento non autentico.
Response and postconditions	Il cliente rischia di incorrere nel furto dei propri dati/credenziali per effettuare il pagamento.
Mitigations	
Non Functional requirements	

Tabella 13

Misuse case: Emissione biglietto	
Actors	Staff della biglietteria maldestro
Description	Un membro dello staff altera per sbaglio l'elenco dei biglietti emessi o permette a terzi di accedere nella propria area riservata
Data (asset)	Elenco biglietti emessi
Stimulus and preconditions	
Attack 1 Flow	Un membro dello staff dimentica di fare il logout
Attack 2 Flow	Un membro dello staff espone le credenziali a terzi
Attack 3 Flow	Un membro dello staff per errore modifica o cancella una voce dell'elenco dei biglietti emessi
Response and postconditions	1-2. Utente non autorizzato ottiene le credenziali ed accede all'area riservata 3. Elenco dei biglietti alterato
Mitigations	
Non Functional requirements	

Tabella 14

Misuse case: lettura biglietto	
Actors	Annullatore del biglietto maldestro, Cliente
Description	L'annullatore gestisce in maniera errata la modifica dell'elenco biglietti annullati
Data (asset)	Biglietto, elenco biglietti emessi/annullati
Stimulus and preconditions	Il Cliente vuole entrare all'evento ma c'è una rottura del lettore di QR Code
Attack 1 Flow	Il lettore QR Code si danneggia per cause accidentali e non è più funzionante, l'annullatore maldestro effettua una modifica errata del database
Response and postconditions	Elenco dei biglietti emessi/annullati non aggiornato o erroneamente aggiornato
Mitigations	
Non Functional requirements	

Tabella 15

Misuse case: Invalidazione biglietto	
Actors	Annullatore del biglietto maldestro, Cliente
Description	L'annullatore gestisce in maniera errata la modifica dell'elenco biglietti annullati
Data (asset)	Biglietto, elenco biglietti emessi/annullati
Stimulus and preconditions	Il Cliente vuole entrare all'evento ma c'è una rottura del lettore di QR Code
Attack 1 Flow	Il lettore QR Code si danneggia per cause accidentali e non è più funzionante, l'annullatore maldestro effettua una modifica errata del database
Response and postconditions	Elenco dei biglietti emessi/annullati non aggiornato o erroneamente aggiornato
Mitigations	
Non Functional requirements	

Tabella 16

Misuse case: Elenco biglietti emessi/annullati	
Actors	Annullatore del biglietto maldestro, Operatore della biglietteria maldestro
Description	L'annullatore gestisce in maniera errata la modifica dell'elenco biglietti annullati
Data (asset)	Biglietto, elenco biglietti emessi/annullati
Stimulus and preconditions	Creazione tabella database con l'elenco dei biglietti emessi/annullati
Attack 1 Flow	L'annullatore maldestro effettua una modifica errata del database.
Response and postconditions	Elenco dei biglietti emessi/annullati non aggiornato o erroneamente aggiornato
Mitigations	
Non Functional requirements	

Tabella 17



## Casi d'abuso

Descrivono i possibili scenari previsti per un utilizzo intenzionalmente malevolo del sistema, rappresentati secondo la sequenza ordinata delle azioni di cui si compongono, che definiscono l'interazione tra l'utente e il sistema o tra le varie componenti del sistema.

Abuse case: Comunicare numero posti	
Actors	Event Manager, Attaccante
Description	L'attaccante altera il numero effettivo di posti disponibili per l'evento.
Data (asset)	Posti
Stimulus and Preconditions	1. L'attaccante sottrae le credenziali all'event manager. 2.
Attack 1 Flow	Accesso non autorizzato all'account dell'event manager per aumentare il numero dei posti tramite SQL injection
Attack 2 Flow	L'attaccante intercetta ed altera i dati effettuando un attacco di tipo MITM
Response and postcondition	L'attaccante ha modificato il numero di posti disponibili per l'evento e quindi l'attaccante ha guadagnato un extra profitto illecito e i clienti non trovano posto il giorno dell'evento.
Mitigations	
Non Functional requirements	

Tabella 18

Abuse case: Acquisto biglietto	
Actors	Biglietteria, Utente, Sistema di pagamento, Attaccante
Description	L'attaccante cerca di sottrarre denaro al cliente
Data (asset)	Pagamento, Conferma di pagamento
Stimulus and preconditions	<ol style="list-style-type: none"> <li>1. Il cliente inserisce le proprie credenziali nel sito di pagamento non autentico.</li> <li>2. L'attaccante entra in possesso delle credenziali di un membro dello staff della biglietteria.</li> <li>3. L'attaccante entra in possesso delle credenziali del cliente relative al sito dei biglietti.</li> <li>4. Nessuna precondizione.</li> </ol>
Attack 1 Flow	L'attaccante utilizza le credenziali inserite dell'utente nel sito di phishing (sito relativo al pagamento) per sottrarre denaro.
Attack 2 Flow	L'attaccante si introduce nella biglietteria e richiede più volte il pagamento.
Attack 3 Flow	L'attaccante si finge la biglietteria e richiede più volte il pagamento.
Attack 4 Flow	L'attaccante impedisce al cliente di acquistare il biglietto.
Response and postconditions	<p>L'attaccante ottiene un ricavo o blocca il servizio di pagamento.</p> <p>Il cliente perde denaro e non riceve il biglietto per accedere all'evento.</p>
Mitigations	
Non Functional requirements	

Tabella 19

Abuse case: verifica pagamento avvenuto	
Actors	Biglietteria, Sistema di pagamento, Attaccante
Description	Il sistema di pagamento invia la ricevuta di pagamento avvenuto alla biglietteria e quest'ultimo viene intercettato dall'attaccante.
Data (asset)	Conferma di pagamento.
Stimulus and preconditions	Il sistema di pagamento ha inviato la ricevuta di pagamento alla biglietteria.
Attack 1 Flow	L'attaccante invia una ricevuta di pagamento al cliente non autentica e si appropria di quella autentica.
Attack 2 Flow	L'attaccante blocca la ricezione della ricevuta di pagamento.
Response and postconditions	<ol style="list-style-type: none"> <li>1. L'attaccante riceve la ricevuta autentica e ottiene il biglietto.</li> <li>2. L'attaccante blocca la conferma di pagamento.</li> </ol>
Mitigations	
Non Functional requirements	

Tabella 20

Abuse case: emissione biglietto	
Actors	Staff della biglietteria, Attaccante
Description	L'attaccante accede all'area riservata della biglietteria per emettere il biglietto gratuitamente, ottenere un ricavo o per impedire il servizio di emissione del biglietto
Data (asset)	Elenco biglietti emessi
Stimulus and preconditions	L'attaccante ottiene l'accesso all'area riservata dello staff della biglietteria
Attack 1 Flow	Una volta ottenuto l'accesso all'area riservata, l'attaccante si finge l'ente di certificazione per apporre il sigillo
Attack 2 Flow	Attacco di tipo DoS per bloccare il sistema ed impedire l'emissione del biglietto
Response and postconditions	1. L'attaccante ottiene biglietti validi senza pagare o ottenere un ricavo vendendoli 2. L'attaccante interrompe il servizio di emissione del biglietto
Mitigations	
Non Functional requirements	

Tabella 21

Abuse case: apposizione sigillo	
Actors	Biglietteria, Attaccante
Description	L'attaccante appone un sigillo fasullo o cerca di interrompere il servizio.
Data (asset)	Sigillo, Biglietto.
Stimulus and preconditions	Biglietto senza sigillo correttamente generato
Attack 1 Flow	L'attaccante appone un sigillo non autentico sul biglietto generato.
Attack 2 Flow	L'attaccante cerca di interrompere il servizio.
Response and postconditions	1. Biglietto fasullo generato, 2. Servizio interrotto
Mitigations	
Non Functional requirements	

Tabella 22



Abuse case: Invio biglietto	
Actors	Cliente, Attaccante
Description	L'attaccante sottrae le credenziali al cliente o blocca il download del biglietto.
Data (asset)	Biglietto.
Stimulus and preconditions	L'utente ha acquistato un biglietto valido.
Attack 1 Flow	L'attaccante sottrae le credenziali di login all'utente tramite accesso non autorizzato all'area riservata del cliente (phishing, social engineering, SQL injection sul database contenente le credenziali dei clienti)
Attack 2 Flow	L'attaccante cerca di interrompere il servizio per impedire il download all'utente
Response and postconditions	1. Credenziali/biglietto rubato 2. Servizio di invio biglietto interrotto
Mitigations	
Non Functional requirements	

Tabella 23

Abuse case: lettura biglietto	
Actors	Annullatore biglietto, Cliente, Attaccante
Description	L'attaccante cancella l'elenco dei biglietti emessi oppure non permette la consultazione del database dei biglietti venduti
Data (asset)	Biglietto, elenco biglietti emessi/annullati
Stimulus and preconditions	Il Cliente vuole accedere all'evento
Attack 1 Flow	Attacco SQL Injection per cancellare elenco biglietti emessi per impedire la lettura del biglietto
Attack 2 Flow	Attacco di DoS per impedire l'invalidazione del biglietto
Response and postconditions	Viene impedito l'ingresso all'evento
Mitigations	
Non Functional requirements	

Tabella 24

Abuse case: invalidazione biglietto	
Actors	Annullatore biglietto, Cliente, Attaccante
Description	L'attaccante cancella l'elenco dei biglietti emessi oppure non permette la consultazione del database dei biglietti venduti
Data (asset)	Biglietto, elenco biglietti emessi/annullati
Stimulus and preconditions	Il Cliente vuole accedere all'evento
Attack 1 Flow	Attacco SQL injection per cancellare elenco biglietti emessi per impedire la lettura del biglietto
Attack 2 Flow	Attacco di DoS per impedire l'invalidazione del biglietto
Response and postconditions	Viene impedito l'ingresso all'evento
Mitigations	
Non Functional requirements	

Tabella 25

Abuse case: Elenco biglietti emessi/annullati	
Actors	Elenco biglietti emessi/annullati, Attaccante
Description	L'attaccante tenta di alterare l'elenco dei biglietti emessi/annullati per il riutilizzo dei biglietti
Data (asset)	Elenco biglietti emessi/annullati, Attaccante
Stimulus and preconditions	Tabella database con l'elenco dei biglietti emessi/annullati esistente
Attack 1 Flow	Attacco SQL injection per alterare l'elenco dei biglietti emessi/annullati
Response and postconditions	Elenco alterato Biglietti riutilizzabili
Mitigations	
Non Functional requirements	

Tabella 26

## Alberi d'attacco

In questa fase ci occupiamo di individuare quali sono le superfici di attacco e di impatto al fine di definire i nostri vettori di attacco. Infatti, gli **alberi di attacco** ci permettono di individuare tali vettori.

Ora vediamo nello specifico che cosa si intende per i termini che abbiamo menzionato precedentemente: la **superficie di attacco** sta ad indicare tutte quelle interfacce di input o output che costituiscono il nostro sistema, con il quale un utente esterno può interagire; queste interfacce sono un canale tramite cui un utente malevolo può attaccare il nostro sistema. Per **superficie di impatto** intendiamo quella superficie che viene colpita da un possibile attaccante, il quale sfrutta una certa vulnerabilità del sistema compromettendolo. Il **vettore di attacco** lo possiamo vedere come un vettore che collega le due superfici, quella di attacco e quella di impatto. In altre parole, per vettore di attacco intendiamo descrivere esattamente come, sfruttando una certa interfaccia, si è in grado di arrivare a colpire un certo punto nella superficie d'impatto.

Utilizzando il linguaggio i\* (i-star), complementare ad UML, l'albero d'attacco viene realizzato sfruttando la rappresentazione **AND-OR-DECOMPOSITION**, e successivamente vedremo che per ogni vettore di attacco definito è possibile associare, in maniera qualitativa, il valore della probabilità che possa verificarsi e che impatto possa generare.

Nel primo diagramma di Figura 7, il vettore di attacco individuato prevede l'**ottenere un ricavo** che si scompone in due possibili sotto-attività, quali: **sottrarre soldi al cliente** e **sottrarre i biglietti alla biglietteria**.

Ovviamente per ognuno di questi due scenari, in cascata, sono stati individuati dei sotto-task che possono essere compiuti da un eventuale attaccante che decide di attaccare il sistema per trarne profitto, fino ad arrivare a mostrare quali sono i possibili "strumenti", o meglio interfacce, che può sfruttare per mettere in atto l'aggressione.

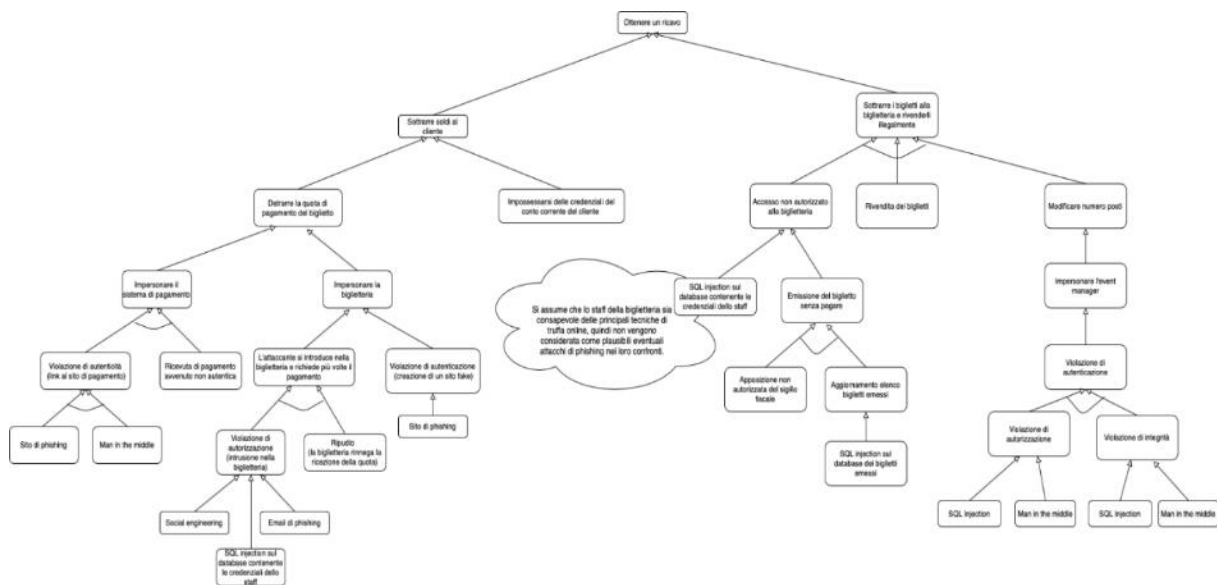


Figura 7

Di seguito riportiamo gli altri scenari, o meglio alberi di attacco, individuati per evitare possibili attacchi che possano sfruttare le vulnerabilità del nostro sistema, ovvero:

- **Entrare senza pagare il biglietto;**
- **Impedire lo svolgimento dell'evento.**

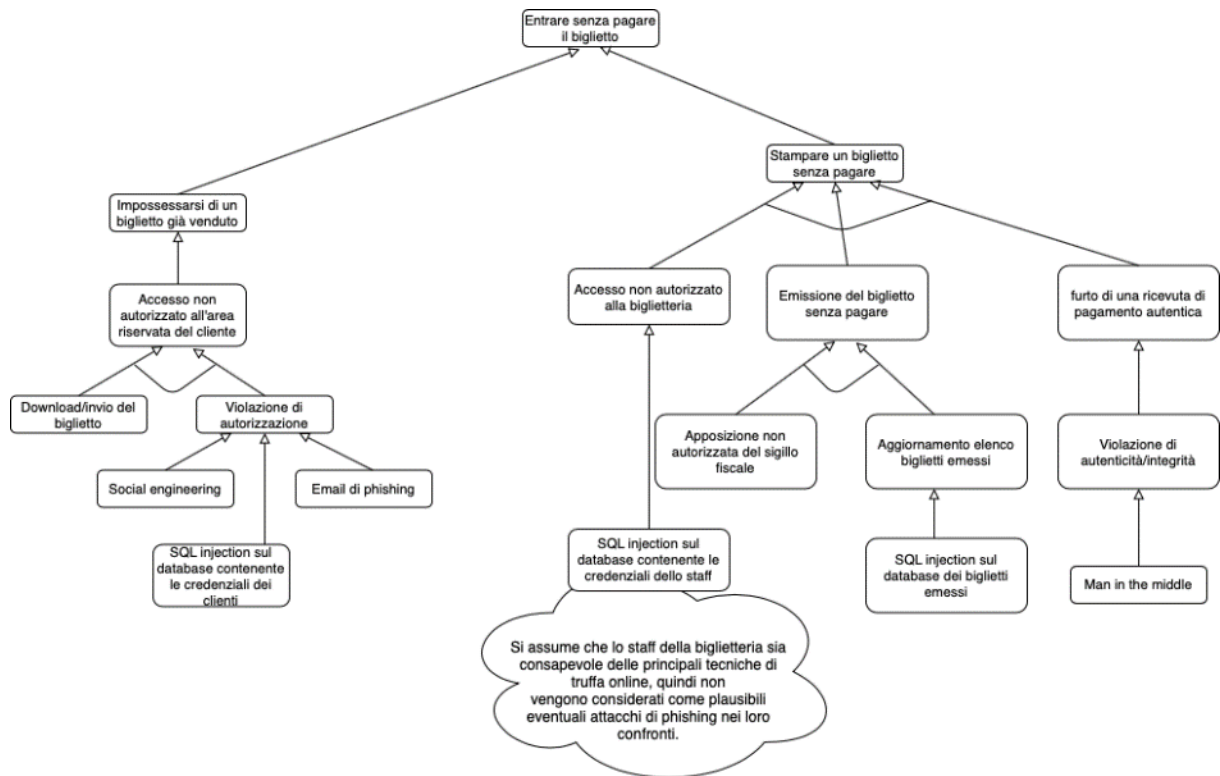


Figura 8

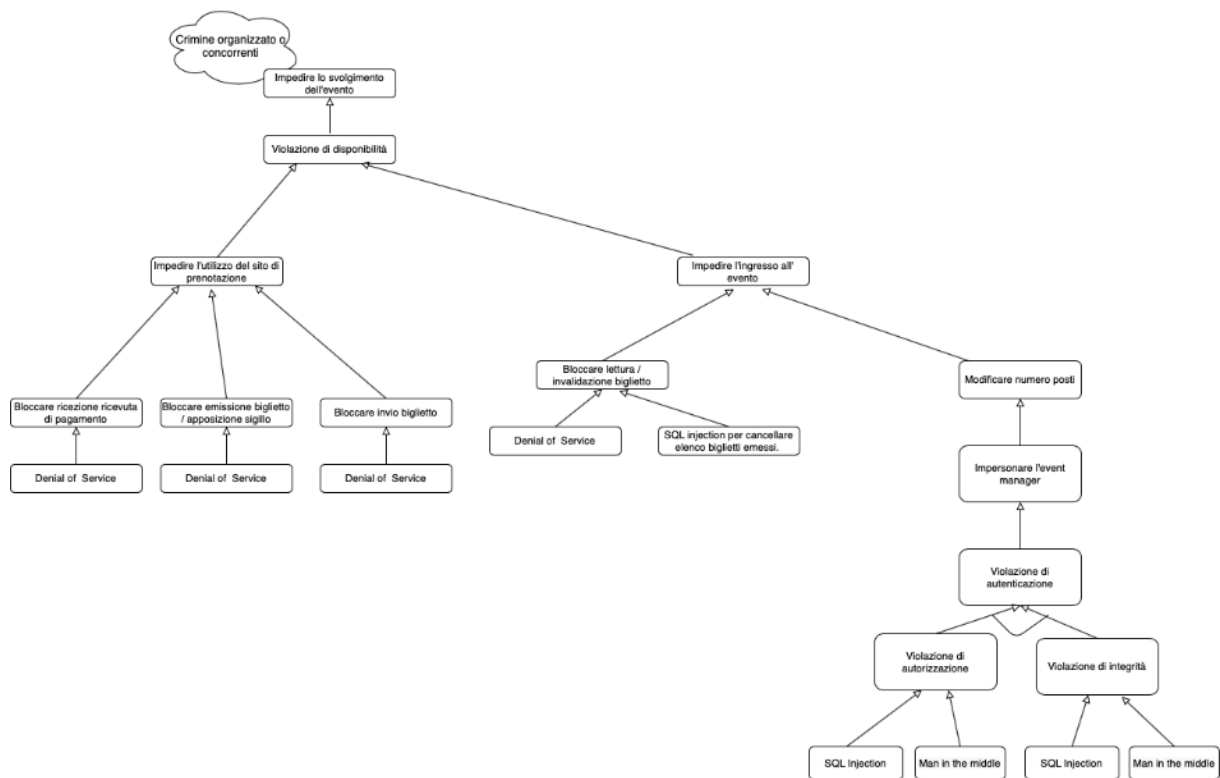


Figura 9

In quest'ultima immagine invece abbiamo mostrato quali possono essere le azioni svolte in maniera accidentale da parte degli utenti (attori principali) che, utilizzando in modo diretto il nostro sistema, possono causare un malfunzionamento dello stesso, o addirittura comprometterne il funzionamento.

Il tutto ribadendo che le azioni in questo caso non vengono compiute in maniera malevola, ma con modalità del tutto inconsapevoli, cioè se l'utente ha un comportamento maldestro. Le cause possono essere dovute ad una disinformazione sul funzionamento corretto delle interfacce che formano il sistema.

Gli attori interessati sono: **Event Manager Maldestro**, **Staff della biglietteria Maldestro** e **Cliente Maldestro**.

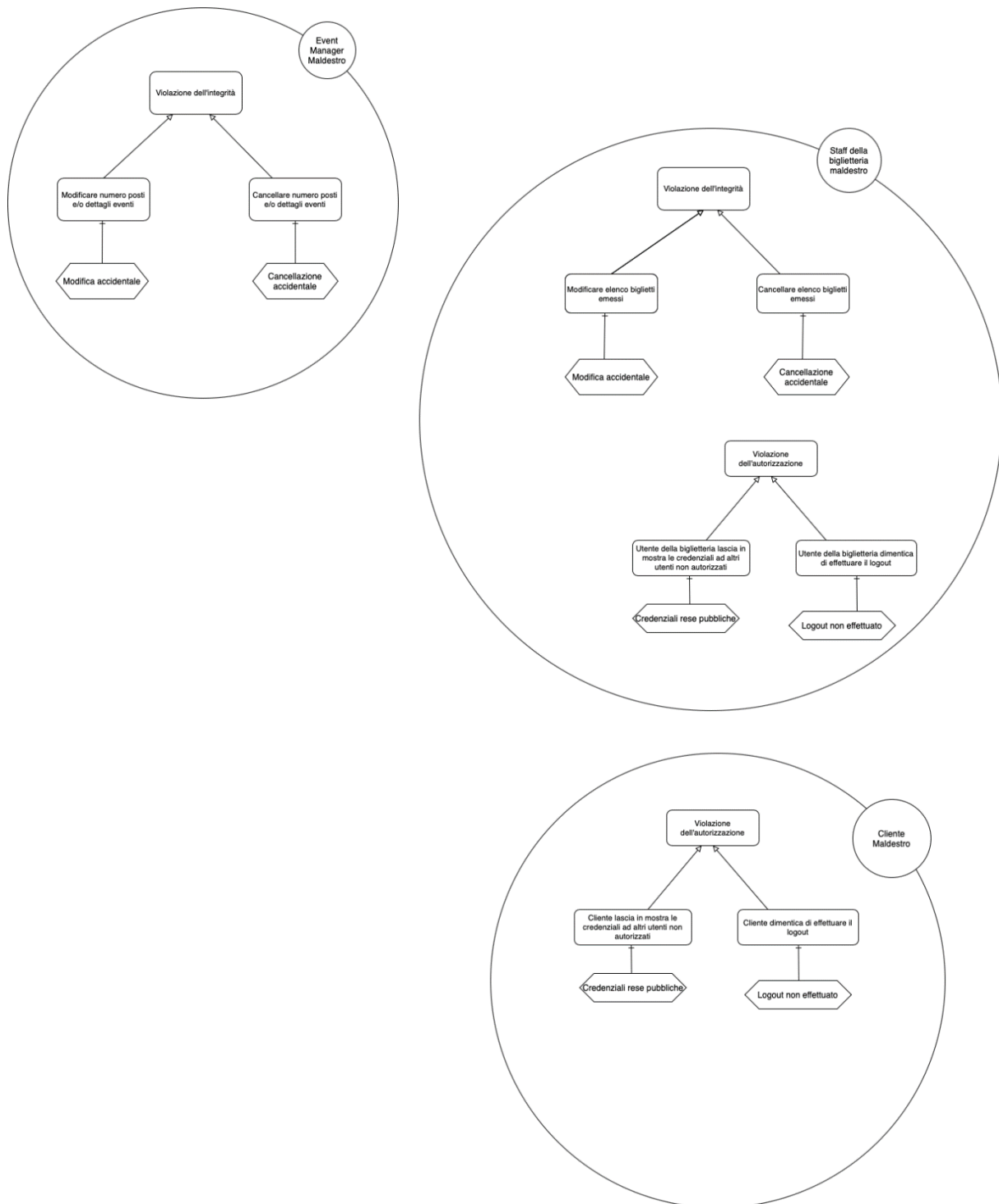


Figura 10

## Applicazione del metodo STRIDE

In questa fase abbiamo cominciato a costruire la tabella che rappresenta il **MODELLO STRIDE**, il cui schema già ci suggerisce delle prime forme di violazione che possono interessare gli asset individuati per il nostro sistema.

Nella tabella di riferimento (che troviamo nel file excel con il nome “STRIDE table”), abbiamo riportato i nomi dei nostri asset e per ognuno siamo andati a porre una “X” in relazione alla categoria di violazione che interessava quel certo asset.

In questa tabella sono anche riportati i valori, espressi in forma qualitativa, della probabilità con cui un attacco relativo a un asset si verifica, ed anche quello relativo all’impatto che può generare un certo attacco. Moltiplicando questi due valori otteniamo il **rischio inerente**, ovvero il rischio che si incorre nel caso in cui si verifichi quella certa violazione prima di aver provveduto ad implementare una misura di mitigazione per quel possibile attacco.

Successivamente si vanno ad individuare tali misure per ogni possibile minaccia relativa ad ogni asset che costituisce il sistema, e in seguito si vanno a riproporre i valori qualitativi di probabilità e impatto, il cui prodotto, a questo punto, genera il **rischio residuo**, ovvero il rischio in cui si incorre qualora si verifichi un certo attacco, che però può essere mitigato da una data misura di sicurezza implementata ad-hoc.

Ciò ovviamente è legato anche alla **fattibilità** di implementare una certa misura di sicurezza e il relativo costo. Infatti, abbiamo riportato una colonna specifica nella quale viene mostrato il costo richiesto per ogni misura di mitigazione rilevata.

# DESIGN

## Architettura software

Durante la stesura dei diagrammi i\*, in particolare nello **Strategic Rationale Model**, abbiamo individuato una serie di moduli software in cui scomporre la nostra applicazione web. Al centro del diagramma troviamo il modulo **System**, che si occupa di elaborare le richieste che arrivano dagli attori e di controllare e gestire gli accessi alle risorse della web app. Gli altri moduli fungono da interfaccia tra gli attori della nostra biglietteria automatica e le risorse di sistema.

A partire dai risultati ottenuti dopo l'analisi dei requisiti, abbiamo deciso di strutturare la nostra applicazione web utilizzando un'**architettura client-server** basata sul pattern **Model View Controller (MVC)**.

Operazioni come l'emissione e l'invalidazione dei biglietti sono molto critiche e delicate, devono quindi essere caratterizzate da alta disponibilità. Inoltre, è necessario garantire l'autenticità e l'integrità dei biglietti. L'utilizzo di una **Distributed Ledger Technology (DLT)** risulta essere la soluzione più adatta per garantire i requisiti di sicurezza e dependability associati a questi servizi.

Ad ogni evento disponibile nel catalogo dell'applicazione web viene associata un'istanza di uno **smart contract**, che ha il compito di gestire l'emissione e l'invalidazione dei biglietti per quell'evento.

In particolare, la scelta è ricaduta su una blockchain privata permissioned, che permette di definire policy più stringenti:

- possibilità di eseguire transazioni private;
- selezione dei nodi validatori;
- selezione degli utenti della rete.

In una blockchain, i dati relativi alle transazioni e lo stato degli smart contract sono ridondati e distribuiti. Ogni nodo della blockchain possiede una **copia del registro distribuito** e memorizzarci tutti i dati necessari al funzionamento della web app sarebbe molto oneroso. Pertanto, abbiamo scelto di memorizzare nella blockchain solo i dati critici, mettendo tutto il resto all'interno di un database; il suo utilizzo semplifica anche la gestione delle procedure di autenticazione e autorizzazione, necessarie per proteggere l'accesso alle risorse.

Il **modulo System** esegue i seguenti compiti:

- elabora le richieste dei client, verificando la loro validità e sanificandole;
- interagisce con la blockchain per soddisfare le richieste di emissione ed invalidazione dei biglietti;
- interagisce con il database per recuperare i dati relativi alle risorse del sito web (utenti, eventi, biglietti...).



## Scelte tecnologiche

### Scelta della blockchain

Le prime scelte tecnologiche effettuate sono state quelle relative alla blockchain. Come richiesto dalle specifiche che ci sono state fornite, abbiamo scelto di implementare una nostra blockchain basata su **Quorum**. La configurazione e la distribuzione della blockchain sono state svolte usando il tool a riga di comando **Quorum-wizard**. Questo tool ci ha permesso di istanziare una blockchain di prova in locale con soli tre nodi validatori.

Per la compilazione degli smart contract abbiamo deciso di impiegare la libreria **solc**, integrabile con il nostro server Node.js. Lo smart contract viene ricompilato ogni volta prima di fare il deploy di un'istanza del contratto.

In generale, l'utilizzo della blockchain garantisce ridondanza, diversità e distribuzione (quando i nodi sono dislocati geograficamente).

### Scelta del linguaggio di programmazione

La scelta della blockchain ha influenzato anche la scelta del linguaggio di programmazione utilizzato per sviluppare il server. L'interazione tra una web app ed una blockchain Quorum o Ethereum è possibile tramite l'utilizzo delle **librerie Web3**. Queste librerie sono disponibili per Javascript, Python e Java.

La scelta è ricaduta su **Javascript**, poiché permette di implementare un potente web server in maniera semplice e veloce. Javascript presenta diversi vantaggi:

- permette di usare lato server lo stesso linguaggio di programmazione usato dal client;
- consente di effettuare in modo semplice chiamate di funzioni asincrone, molto utili nell'interazione con una blockchain o nei servizi web in generale;
- fornisce una vasta gamma di librerie per implementare le misure di controllo individuate in fase di progettazione;
- permette di utilizzare il gestore di pacchetti **npm** per integrare librerie e gestire le dipendenze.

Per utilizzare il linguaggio Javascript lato server è necessario installare il framework **Node.js**.

Il gestore di pacchetti npm, tra le varie cose, consente di eseguire una scansione delle vulnerabilità note presenti all'interno dei pacchetti importati. Per eseguire la scansione è sufficiente lanciare il comando **npm audit**. Nel caso in cui venga rilevata una vulnerabilità, ne viene calcolato l'impatto e vengono fornite delle azioni correttive da applicare per rimuoverla o mitigarla.

Quando si esegue l'installazione di tutti i pacchetti necessari al funzionamento dell'applicazione web, si può notare che tra questi non è presente alcuna vulnerabilità conosciuta.

```
added 566 packages, and audited 567 packages in 8s

75 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Figura 11 I pacchetti software utilizzati da ticketTwo non contengono vulnerabilità conosciute

## Scelta del sistema operativo

Per quanto riguarda il sistema operativo la scelta è ricaduta su **Linux**, poiché **Quorum-wizard** gira ottimamente su questo sistema operativo, senza la necessità di installare una macchina virtuale. Inoltre, è open source e solitamente molto utilizzato lato server.

## Scelta del database

La scelta del database per la nostra applicazione web è ricaduta su **MongoDB**, un database di tipo non relazionale, gratuito, che si integra facilmente con progetti Node.js tramite l'uso della libreria **Mongoose**.

L'utilizzo di un database ci permette di archiviare i dati personali degli utenti inseriti al momento dell'iscrizione al sito, quelli necessari per l'autenticazione, quelli relativi agli eventi disponibili sul sito e quelli relativi alle ricevute di pagamento dei biglietti. Utilizziamo il database anche per archiviare, temporaneamente, i codici OTP inviati agli utenti tramite e-mail per la procedura di autenticazione a due fattori.

Il database che abbiamo scelto per collegare il nostro server risiede su un **cluster AWS** che si trova nei pressi di Francoforte. In caso di guasti, incidenti o malfunzionamenti al servizio, i nostri dati sono comunque al sicuro. Infatti, quando si inizializza un database su MongoDB, di default vengono assegnati tre diversi server, uno primario e due di scorta, per la memorizzazione dei dati.

MongoDB fornisce, quindi, un servizio affidabile garantendo **ridondanza**. I server però sono situati tutti nello stesso data center, perciò non è garantita la distribuzione: in caso di calamità, si corre il rischio concreto di perdere i dati.

L'utilizzo di un database esterno per memorizzare i dati della web app richiede di definire delle **policy stringenti** per l'accesso e la modifica degli elementi archiviati su di esso. Il database è stato configurato in maniera tale che solo l'account utilizzato dal nostro server acceda con privilegi di scrittura. Tutte le modifiche al database devono, quindi, passare per il server, che esegue una serie di controlli sulle richieste.

Ci è sembrato opportuno aggiungere ulteriori misure di sicurezza per garantire confidenzialità, integrità e autenticità dei dati archiviati. La libreria **Mongoose Encryption** ci permette di cifrare e autenticare un record prima di memorizzarlo sul database, in modo tale da rendere impossibili letture o alterazioni malevole direttamente dall'interfaccia grafica fornita da MongoDB. La libreria utilizza due chiavi segrete, una per la **cifratura** e una per la **firma digitale**.

## Scelta del sistema di pagamento esterno

Il pagamento dei biglietti è sicuramente una delle operazioni più delicate che avvengono su ticketTwo. Implementare un sistema di pagamento da zero è un'impresa molto ardua e comporta altissimi rischi. Perciò abbiamo deciso di optare per un sistema di pagamento esterno. Così, oltre a poter fare affidamento su un servizio già ampiamente collaudato, abbiamo il vantaggio di condividere (e quindi ridurre) i rischi relativi alle attività di pagamento con il provider del servizio.

La scelta è ricaduta su **PayPal**, uno dei più noti servizi di pagamento digitale. L'azienda statunitense fornisce delle API che permettono di integrare in maniera molto semplice il loro servizio in progetti Node.js. Inoltre, mette a disposizione una **sandbox** che permette di testare i pagamenti effettuati con la nostra applicazione web prima di renderla effettivamente operativa, in maniera tale da scovare bug e vulnerabilità.

## Comunicazione sicura tra client e server

Durante la fase di progettazione abbiamo individuato una serie di minacce agli asset della nostra web app, derivanti dall'utilizzo di una connessione non sicura tra client e server. Per renderla sicura è necessario garantire mutua autenticazione tra i due (o eventualmente autenticare solo il server) e cifrare i dati scambiati. Queste operazioni vengono effettuate in maniera automatica quando si stabilisce una connessione ad un server che utilizza il protocollo **HTTPS (HyperText Transfer Protocol over Secure Socket Layer)**.

In Node.js è possibile implementare un server di tipo HTTPS semplicemente importando la **libreria https** e invocando la **funzione https.createServer()**. Il protocollo per poter funzionare richiede una chiave privata (nel nostro caso una **chiave RSA** a 2048 bit) e un certificato di chiave pubblica, rilasciato da un'autorità terza o auto-firmato.

Nella fase di testing dell'applicazione abbiamo deciso di optare per la seconda opzione, generando un certificato con il tool a riga di comando **OpenSSL**. Naturalmente un certificato auto-firmato risulta poco attendibile, quindi nel momento in cui metteremo **ticketTwo** online dovremo provvedere a farne rilasciare uno da un'autorità garante. Ciò comporterà costi aggiuntivi, ma garantisce maggiore sicurezza per la nostra applicazione.

L'utilizzo di HTTPS ci permette di eliminare i rischi di attacchi **man in the middle**, **phishing** e **eavesdropping**.

## Scansione QR code

Per semplificare l'operazione di invalidazione dei biglietti, durante la fase di progettazione abbiamo deciso di impiegare un **lettore di codici QR**. In questo modo, per accedere ad un evento il cliente non deve fare altro che mostrare il codice che ha ricevuto dopo l'acquisto del biglietto.

Il codice QR relativo ad un biglietto viene generato a partire dai dati del biglietto che sono memorizzati nel database della web app in formato JSON. Generare un codice QR a partire da un JSON in Node.js è molto semplice. Per prima cosa abbiamo trasformato l'oggetto JSON in una stringa, poi abbiamo importato la libreria **qrcode** e invocato la funzione **qrcode.toDataURL()**, che restituisce una stringa **base64** contenente una rappresentazione binaria dell'immagine del QR code.

L'utente può scaricare e stampare un **PDF del biglietto** direttamente dalla propria area riservata. La generazione del PDF avviene utilizzando la **libreria html-pdf** lato server. Nel momento in cui il cliente si

recherà all'evento, dovrà mostrare all'annullatore il codice presente sul biglietto cartaceo oppure la sua versione digitale, ad esempio salvata sul cellulare.

La scansione dei codici QR, quindi la lettura del biglietto, avviene lato client grazie all'ausilio della libreria **jsqr**. L'annullatore nella propria area riservata ha accesso alla fotocamera del dispositivo. Quando questa viene attivata, si attiva anche lo **scanner dei QR code**. Ogni 2 secondi viene effettuato uno **screenshot** del flusso video in ingresso dalla fotocamera e, invocando la **funzione jsQR** della libreria citata precedentemente, si tenta di decifrare il codice presente nell'istantanea appena scattata.

Se il codice viene rilevato, i dati del biglietto vengono inviati al server che provvederà a verificare la validità del sigillo fiscale ed eventualmente a invalidare il biglietto. Se la funzione non riesce ad individuare alcun codice, allora non viene eseguita alcuna operazione. Semplicemente si aspettano altri 2 secondi prima di effettuare un nuovo tentativo.

## Sigillo fiscale

Il biglietto presentato dall'utente al momento dell'ingresso all'evento, per poter essere considerato valido dall'annullatore, deve possedere un sigillo fiscale. Questo sigillo garantisce integrità e autenticità del biglietto.

Il sigillo viene generato lato server subito dopo l'emissione del biglietto, attraverso l'**approccio hash and sign**. Il server calcola l'hash dei dati del biglietto utilizzando l'**algoritmo SHA-256**, per poi firmarlo utilizzando l'algoritmo di cifratura asimmetrica **RSA**. Per eseguire questa procedura ci siamo affidati alla **libreria Crypto** integrata in Node.js.

Il calcolo dell'hash garantisce integrità, dimostrando come il biglietto non sia stato compromesso e alterato, mentre la firma digitale garantisce che il biglietto sia stato rilasciato dal server di **ticketTwo** (solo il server conosce la chiave privata per generare la firma).

Il sigillo fiscale viene memorizzato nel database, insieme ai dati relativi al biglietto, ed è inserito anche all'interno del codice QR che il cliente mostra al momento dell'ingresso all'evento.

I record del database sono protetti con tecniche di offuscamento e firmati con una chiave privata per garantire autenticità e integrità. Il database risulta, quindi, un posto sicuro in cui memorizzare il sigillo. Non risulta, quindi, necessario memorizzare il sigillo fiscale anche sul token associato al biglietto presente sulla blockchain.

## Autenticazione

L'autenticazione alla web app viene svolta dal server con l'ausilio del database. Al momento dell'iscrizione al sito il client invia i dati personali al server. Questi sono protetti grazie all'utilizzo del **protocollo TLS**. Il server, una volta ricevuti i dati dell'utente, calcola l'**hash** della password e la memorizza nel database insieme agli altri suoi dati. Per evitare i cosiddetti attacchi "del dizionario", prima di calcolare l'hash viene aggiunto un **salt** (cioè una sequenza casuale di bit) alla password. Nel momento in cui l'utente accede al sito web non si controllerà la validità della password, ma se ne calcola l'hash e lo si confronta con quello memorizzato nel database.

Inoltre, come già accennato, i dati presenti sul database sono cifrati e firmati, in modo da garantire un ulteriore livello di sicurezza. Per eseguire queste operazioni ci siamo affidati alla **libreria bcrypt**.

Per alcune operazioni critiche abbiamo deciso di inserire un'ulteriore misura di autenticazione, implementando di fatto un approccio **2FA (2 Factor Authentication)**. Quando viene richiesta una pagina web protetta con 2FA, il cliente riceve una e-mail con un **codice OTP a 4 cifre** generato in maniera casuale. Il codice andrà poi inserito in un opportuno form per essere convalidato dal server ed accedere così alla risorsa richiesta. I codici OTP appena generati vengono memorizzati nel database, anche questi in maniera cifrata, e dopo l'autenticazione dell'utente vengono eliminati.

L'utente, per dare prova di essere autenticato, deve mostrare di possedere un **token** di autenticazione (nel caso di autenticazione multi-fattore dovrà dimostrare di possederne due). Il token viene inviato all'utente al termine della procedura di **login** e memorizzato all'interno dei **cookies** allegati alle successive richieste HTTP. I cookies che contengono il token hanno una validità temporanea: un'ora per la normale autenticazione e un minuto per quella multi-fattore. Eliminando i cookies del browser l'utente perderà i propri token e dovrà nuovamente autenticarsi per accedere all'area riservata.

Il modo in cui abbiamo gestito le password (hash + salt + cifratura + firma) e l'impiego dell'autenticazione multi-fattore soddisfano il principio di difesa in profondità presente in molte linee guida per la progettazione sicura.

## Accountability

L'utilizzo della blockchain permette di mantenere un registro immutabile di tutte le transazioni effettuate con lo smart contract, tenendo traccia delle operazioni di emissione e invalidazione effettuate. Abbiamo deciso di mantenere anche un registro parallelo in cui andiamo a memorizzare tutti gli accessi al sito web.

Inoltre, al momento dell'iscrizione, al termine dell'acquisto dei biglietti o dopo la concessione di nuovi privilegi ad un utente, il server invia una e-mail al diretto interessato per informarlo che l'operazione è avvenuta con successo.

L'invio delle e-mail è gestito dalla **libreria Nodemailer** a partire da una casella di posta **Gmail**, mentre la confidenzialità è garantita dall'utilizzo del **protocollo TLS** (lo stesso usato da HTTPS).

## Gestione dei segreti

L'utilizzo di servizi esterni, come l'archiviazione sul database o il trasferimento di denaro tramite PayPal, richiede di memorizzare le credenziali di accesso a questi servizi. Poi l'utilizzo del protocollo HTTPS, e delle primitive crittografiche per cifratura e firma digitale, richiede di memorizzare da qualche parte chiavi segrete e certificati di chiave pubblica.

Mantenere questi segreti in chiaro sul database non è assolutamente sicuro. La soluzione migliore che abbiamo individuato è quella di memorizzarli come variabili d'ambiente all'interno di un **file di configurazione .env**.

La **libreria dotenv** ci permette di importare le variabili d'ambiente presenti nel file di configurazione all'avvio del server.

## Protezione

### Protezione a livello piattaforma

L'utilizzo della blockchain garantisce che il codice degli smart contract sia autentico e immutabile. Inoltre, le transazioni sono possibili solo se il wallet di chi la richiede è sbloccato (bisogna conoscere la password del conto).

Per proteggere il server potrebbe essere utile inserire procedure di autorizzazione ed autenticazione al momento dell'accesso al codice, in modo che solo gli sviluppatori autorizzati possano modificarlo, oltre ad inserire un controllo di integrità del file System per verificare se il codice del server sia stato compromesso.

Potrebbe essere necessario anche introdurre un **firewall** per limitare e controllare le connessioni al server, in modo tale da ridurre il rischio di **attacchi Denial of Service (DoS)**.

### Protezione a livello applicazione

Per quanto riguarda **PayPal** ci affidiamo alle loro policy di sicurezza. Il rischio relativo ai pagamenti viene condiviso con l'azienda statunitense, che in caso di problemi al servizio potrebbe rimborsarci. Inoltre, le ricevute di pagamento spedite da PayPal sono cifrate in modo tale da garantire confidenzialità.

Il nostro database su **MongoDB** è protetto da una procedura di autenticazione. Solo il server può accedere con privilegi di lettura e scrittura alla base di dati. L'autenticazione è garantita dal fatto che solo il server conosce il segreto per autenticarsi.

### Protezione a livello record

I singoli record del database sono cifrati e firmati in modo tale da garantire confidenzialità, autenticità e integrità.

Le misure di protezione sono aggiunte come **plugin** esterno che viene integrato allo schema del record. Come già accennato, il plugin è fornito dalla **libreria Mongoose encryption** e va configurato specificando:

- la chiave segreta da usare per la cifratura (campo **encryptionKey**);
- la chiave segreta da usare per la firma digitale (campo **signingKey**);
- I campi del record da escludere dalle operazioni di cifratura e firma (campo **excludeFromEncryption**).

Il plugin va integrato su ogni **modello** definito per il database dell'applicazione web. I campi che abbiamo scelto di escludere dalla cifratura e dall'autenticazione sono quelli che vengono utilizzati nelle **query** per individuare i record del database da restituire.

Le operazioni crittografiche vengono svolte sul server prima di caricare i dati sul database o dopo che siano stati restituiti da una query. Quindi, i campi necessari per identificare i record, come gli **id**, devono restare in chiaro.

Un'alternativa sarebbe quella di scaricare ogni volta un'intera collezione del database, decifrarla ed eseguire la query in locale sul server. Quest'approccio risulta però estremamente oneroso da applicare, senza portare grossi vantaggi in termini di sicurezza.

## Conformità alle linee guida di progettazione sicura

- **Minimizzare la superficie di attacco**

- ticketTwo si appoggia su un database non relazionale. MongoDB utilizza uno strumento sicuro per l'assemblaggio delle query, basato sul formato **Binary JSON (BSON)**. Non è possibile l'inserimento diretto di stringhe, quindi risultano impossibili attacchi di tipo **SQL injection**. Tuttavia, MongoDB permette di eseguire codice JavaScript lato server all'interno delle operazioni **\$where** e **mapReduce**. Ciò introduce un potenziale vettore di iniezione molto pericoloso. Questo tipo di attacco prende il nome di **NoSQL injection**. Poiché tutte le richieste di lettura e scrittura al database sono controllate e sanificate dal server dell'applicazione web, gli attacchi di tipo NoSQL injection non costituiscono un pericolo per la sicurezza del nostro sistema.

- **Impostazioni predefinite sicure**

- ticketTwo non è configurabile. Di default sono incluse tutte le misure che contribuiscono a rendere l'applicazione sicura. Queste non possono essere disabilitate.
- Un utente non autenticato può consultare solamente il catalogo degli eventi senza eseguire alcuna operazione.
- Un utente, quando tenta di accedere ad una rotta per cui non è autorizzato, viene reindirizzato alla pagina di login; potrà accedere solo dopo aver inserito delle credenziali che abbiano gli opportuni privilegi.

- **Principio del privilegio minimo**

- Al momento dell'iscrizione su ticketTwo, ad ogni utente vengono assegnati i privilegi da cliente. Le uniche operazioni possibili per tale utente sono la consultazione del catalogo e l'acquisto dei biglietti. Per ottenere privilegi superiori, si dovrà fare richiesta allo staff della biglietteria, che deciderà se concedere o meno le autorizzazioni richieste.

- **Difesa in profondità**

- Autenticazione a più fattori per le operazioni critiche.
- Sanificazione degli input lato client e lato server.
- Le richieste di emissione e invalidazione dei biglietti vengono valutate prima dai controller del server e poi dallo smart contract.
- Solo il server conosce il segreto per accedere in modalità scrittura ai record del database e, nonostante ciò, i record sono comunque cifrati e autenticati.

- **Fallire in modo sicuro**

- Se si verifica un'eccezione durante l'accesso o la modifica di una risorsa, il server la gestisce e informa il client con un opportuno messaggio di errore.

- **Separazione dei compiti**

- Non esistono utenti privilegiati che possano fare qualunque cosa. I privilegi associati ad un account permettono di effettuare solamente un numero limitato di operazioni, in base al ruolo che ricopre l'utente. Da un account con privilegi da biglietteria non è possibile acquistare biglietti. Un membro dello staff della biglietteria che intende partecipare ad un evento dovrà possedere anche un altro account con credenziali da cliente.
- Tutti gli utenti accedono allo stesso catalogo degli eventi e hanno a disposizione la stessa barra di navigazione, ma le operazioni consentite dipendono dai privilegi.
- L'event manager visualizza solo gli eventi della società cui appartiene.
- L'utente ospite e il cliente visualizzano solo gli eventi del catalogo per cui sono aperte le vendite.
- Solo lo staff della biglietteria ha accesso all'intero catalogo degli eventi.

- **Open design**

- Evitiamo di conseguire la sicurezza attraverso la segretezza. Per le operazioni crittografiche e i protocolli di consenso della blockchain vengono utilizzati algoritmi standard forniti da librerie esterne. La segretezza è data esclusivamente dalle chiavi private memorizzate come variabili d'ambiente dell'applicazione web.

- **Bilanciare sicurezza e usabilità**

- Gli utenti possono consultare il catalogo degli eventi anche se non sono iscritti a ticketTwo. In questo modo potranno iscriversi solamente nel caso in cui decidano di acquistare i biglietti per un evento.
- L'autenticazione multi-fattore, che potrebbe essere considerata una misura troppo limitante dagli utenti, viene richiesta solamente per alcune operazioni critiche.
- L'interazione con la blockchain per l'emissione e l'invalidazione dei biglietti è gestita interamente dal server e in maniera automatica, sollevando gli utenti da questo compito. Gli indirizzi dei wallet vengono memorizzati all'interno del database, quindi gli utenti dovranno ricordare semplicemente la password ad essi associata.
- L'utilizzo dei token di autenticazione evita di richiedere ogni volta all'utente le proprie credenziali di accesso.



- L'invalidazione del biglietto è costituita da due operazioni: la verifica dell'autenticità del sigillo e la richiesta effettiva dell'invalidazione allo smart contract. Nonostante ciò, per gli utenti la transazione appare semplice, poiché viene svolta scansionando un codice QR.

- **Registrazione azioni utente**

- La blockchain tiene traccia di tutte le transazioni tra gli utenti e le varie istanze degli smart contract.
- All'interno del database vengono registrati tutti gli accessi al sito web.

- **Ridondanza e diversità**

- La blockchain mantiene una copia del registro su ogni nodo validatore. I nodi sono sparsi geograficamente.
- Il database memorizza i dati su tre diversi server, garantendo maggiore affidabilità e continuità di servizio in caso di guasti.

- **Specificare il formato di tutti gli input del sistema**

- I campi presenti all'interno dei form delle pagine web accettano solamente input specifici. Ad ogni campo, a seconda del suo contenuto, viene assegnato un diverso tipo di dato secondo le specifiche di **HTML5**. Abbiamo campi diversi per stringhe generiche, numeri, password, e-mail, date o immagini.
- Prima di inserire i dati all'interno del database, il server controlla se i campi del record sono compatibili con i tipi di dato definiti nel modello della collezione. Per i campi di **tipo stringa** sono definite una **lunghezza minima e una massima**, mentre per quelli di **tipo numerico** sono definiti un **valore minimo e uno massimo**.
- Alcuni campi presenti all'interno dei record sono di **tipo categorico**. Prima di inserire un valore, il server controlla che appartenga all'**insieme dei valori validi** definiti nello schema. Esempi di campi categorici sono il tipo di privilegi dell'utente, la categoria di appartenenza di un evento o il genere di un utente.

- **Suddivisione degli asset**

- Le rotte per l'accesso alle risorse critiche e alle pagine web dell'area riservata sono private e protette. Per ogni rotta viene specificato se questa sia pubblica (disponibile per chiunque) o privata (disponibile solo per utenti autenticati). Nel secondo caso viene specificata anche la lista di privilegi che l'utente deve possedere per accedervi.

- **Mediazione completa**

- Il server, ad ogni richiesta di accesso, verifica il token di autenticazione dell'utente e i suoi privilegi, così da evitare la modifica involontaria o la cancellazione malevola delle risorse.
- Non viene utilizzato alcun tipo di cache.

# IMPLEMENTAZIONE DEL FRONTEND DELL'APPLICAZIONE WEB

## Il Frontend

Durante la fase di progettazione sicura abbiamo individuato le diverse tipologie di attori che potrebbero interagire con ticketTwo. Tra questi vi sono:

- il **cliente**, cioè l'attore che ha come obiettivo quello di acquistare biglietti per partecipare a uno degli eventi disponibili sul nostro sito web;
- l'**event manager**, ovvero l'attore che si occupa di organizzare eventi di intrattenimento, scegliendo e prenotando il luogo in cui si svolgerà l'evento, fissando una data, contattando gli artisti;
- lo **staff della biglietteria**, cioè gli attori che si occupano di mettere in vendita i biglietti per un dato evento;
- l'**annullatore dei biglietti**, ovvero l'attore che si occupa di scansionare i codici QR dei biglietti prima che un cliente possa accedere all'evento, verificandone la validità;
- Il **sistema di pagamento**, cioè l'attore che si occupa di gestire le transazioni per l'acquisto dei biglietti e l'invio delle corrispondenti ricevute.

Ogni diversa tipologia di attore dovrebbe accedere solo a specifiche risorse ed eseguire solamente certe operazioni.

La necessità di garantire i requisiti di autenticazione e autorizzazione ha fatto emergere il bisogno di inserire un ulteriore attore: l'**utente ospite**, cioè quel particolare tipo di attore che non si è ancora autenticato con l'applicazione web. Le uniche operazioni che può svolgere sono:

- la **consultazione del catalogo** degli eventi;
- l'**autenticazione al sito**, tramite una procedura di login;
- l'**iscrizione al sito**, nel caso sia la prima volta che l'utente visita il sito web.

Attraverso l'impiego dei diagrammi i\* abbiamo deciso di scomporre la nostra applicazione in più moduli software. Ognuno di questi si occupa di interagire con una diversa tipologia di attore umano. Abbiamo individuato quattro diversi sottomoduli:

- **Modulo amministrazione evento**, che si occupa dell'interazione con gli organizzatori degli eventi;
- **Modulo gestione biglietti**, per l'interazione con lo staff della biglietteria;
- **Modulo vendite biglietti**, che si occupa dell'interazione con il cliente;
- **Modulo annullatore biglietti**, per l'interazione con l'annullatore dei biglietti;
- **Modulo utente ospite**, che si occupa dell'interazione con gli utenti non autenticati.

Il sistema centrale (server) delega, quindi, la comunicazione con gli attori esterni a questi sottomoduli, i quali costituiscono il **frontend** dell'applicazione.

Ciascun attore interagisce con l'applicazione navigando all'interno di una serie di pagine web. Ogni pagina, ad eccezione della pagina di login e di quella di iscrizione al sito, include una **barra di navigazione**. La barra di navigazione è disponibile per tutte le diverse tipologie di attori, tranne che per l'annullatore dei biglietti, poiché l'unica operazione a lui permessa è la scansione dei codici QR associati ai biglietti.

Nella **barra di navigazione**, oltre al logo dell'applicazione web, sono presenti un insieme di pulsanti che permettono di consultare le diverse sezioni del catalogo degli eventi. Ancora più a destra, con un click sul pulsante Account, si apre un menu a tendina che mostra una serie di azioni che l'utente può eseguire.

Tutti gli utenti che si sono autenticati hanno disponibili tre operazioni, ovvero:

- la **consultazione del proprio profilo utente**;
- la **modifica della password** del proprio account;
- il **logout** dall'area riservata.

In aggiunta a queste, a seconda dei privilegi che possiedono, possono effettuare anche altre operazioni.

## Struttura del codice

Il codice relativo al frontend è contenuto nel percorso ***ticketTwo/front end*** nella directory principale del progetto. La directory è organizzata in sottocartelle secondo la seguente struttura:

- Sottocartella **CSS**, contiene tutti i file di stile CSS allegati alle pagine web;
- Sottocartella **HTML elements**, contiene una serie di elementi grafici HTML utilizzati all'interno delle pagine web del sito (sono codificati in linguaggio javascript lato server);
- Sottocartella **images**, contiene tutte le immagini utilizzate all'interno delle pagine web del sito (come loghi, sfondi);
- Sottocartella **scripts**, contiene tutto il codice che viene eseguito lato client dal browser dell'utente (script allegati alle pagine html);
- Sottocartella **web pages**, contiene i sottomoduli software che si interfacciano con i client che visitano il sito web (sono codificati in linguaggio javascript lato server).

## HTML elements

### widget.js

Il file **widget.js** è il cuore di tutto il frontend. In esso viene dichiarata la classe **Widget**, che permette di costruire in maniera dinamica un elemento HTML da aggiungere ad una pagina web.

Il costruttore della classe richiede come parametri il tipo di tag da generare e il testo da includere nel tag (parametro opzionale). Il compito del costruttore è quello di allocare in memoria lo spazio per quattro strutture dati:

- **attributi**, un oggetto le cui chiavi sono i nomi degli attributi del tag HTML, mentre i valori sono stringhe che rappresentano l'attributo stesso;
- **children**, un array in cui verranno inseriti i figli del tag che si vuole generare;
- **scripts**, un set in cui sono inseriti i nomi dei moduli javascript necessari al funzionamento dell'elemento HTML (sono i moduli contenuti nella relativa cartella);
- **stile**, un oggetto le cui chiavi sono i nomi delle proprietà CSS da associare al tag HTML, mentre i valori sono stringhe che rappresentano la proprietà associata.

La classe fornisce inoltre una serie di metodi, tutti pubblici, che possono essere utilizzati per personalizzare il tag:

- **addChild()**, permette di aggiungere un figlio al tag HTML;
- **addScript()**, permette di importare un modulo javascript nella pagina web in cui viene inserito il tag;
- **getScripts()**, restituisce un set contenente tutti i moduli javascript utilizzati dall'elemento HTML e dai figli;
- **get()**, genera una stringa contenente il tag HTML richiesto a partire dai parametri dell'oggetto;
- **setAttribute()**, permette di impostare il valore di uno degli attributi del tag;
- **setProperty()**, permette di impostare il valore di una delle proprietà CSS da associare al tag.

La funzione **get()** è la più importante della classe. Inizializza una stringa a cui aggiunge come prefisso il nome del tag, racchiuso tra parentesi angolari, e come suffisso il nome del tag, preceduto da slash tra parentesi angolari. All'interno del prefisso vengono aggiunti tutti gli attributi del tag, seguendo la sintassi del linguaggio HTML.

Tra questi vi è anche l'attributo **style**, che va a contenere come valore una stringa con tutte le proprietà CSS del tag. Nella parte centrale della stringa vengono aggiunti tutti i figli del tag ed eventualmente il testo del tag (opzionale). I figli sono aggiunti invocando la funzione **get()** in maniera ricorsiva per ogni figlio.

## HTMLpage.js

All'interno di **HTMLpage.js** è definita la classe **HTMLpage**, che permette di generare una pagina web in maniera dinamica, prima che questa venga inviata al client.

Nel costruttore della classe viene istanziato un oggetto della classe **Widget**. Questo oggetto rappresenta la radice dell'albero DOM associato alla pagina HTML (tag HTML).

A questo tag, utilizzando la funzione **addChild()**, vengono aggiunti due figli:

- Un tag di tipo **head**, che rappresenta l'intestazione della pagina web;
- Un tag di tipo **body**, che rappresenta il corpo della pagina web.

All'interno dell'intestazione vengono impostati alcuni parametri della pagina web, come il titolo, la descrizione, la codifica di caratteri da impiegare (in questo caso UTF-8) e i file di stili CSS da utilizzare.

La classe **HTMLpage** fornisce una serie di metodi, tutti pubblici, che possono essere utilizzati per aggiungere elementi alla pagina web e inviarla come risposta ad una richiesta del client:

- **addChild()**, permette di aggiungere elementi HTML al corpo della pagina web (oggetti della classe **Widget**);
- **send()**, invia la pagina web al client come risposta ad una richiesta HTTP di tipo GET.

La funzione **send()** ottiene una lista di tutti i moduli necessari al funzionamento degli elementi in essa contenuti, invocando in maniera ricorsiva la funzione **getScripts()** per ogni figlio del body. Per ciascun modulo individuato viene aggiunto un tag di tipo **script** nell'intestazione, in cui si specifica di importare le funzioni definite in esso. La funzione **send()** invoca poi il metodo **get()** sulla radice dell'albero DOM, che in maniera ricorsiva va a generare il documento HTML, aggiungendo i vari figli dei tag fino alle foglie. Il documento HTML viene poi allegato alla risposta da inviare al client.

## Elementi grafici

Gli elementi grafici di base utilizzati per riempire il corpo delle pagine web del sito sono tutti contenuti nella cartella **widgets**. Ogni elemento grafico è associato ad una classe che eredita dalla classe **Widget**.

- All'interno del file **bar.js** viene definita la classe **Bar**, che si occupa di generare la barra di navigazione contenuta nella maggior parte delle pagine disponibili sul sito web. Quando viene invocato il costruttore, viene generato un tag di tipo "div" a cui sono aggiunti come figli tutti gli elementi che saranno mostrati nella barra di navigazione:
  - il logo del sito web;
  - i pulsanti che permettono di navigare all'interno delle sezioni del catalogo degli eventi;
  - il pulsante Account per aprire il menu a tendina dell'utente;

- il menu a tendina stesso (che non viene visualizzato fino a quando non si fa click sul pulsante precedente).



Figura 12 Barra di navigazione del sito web

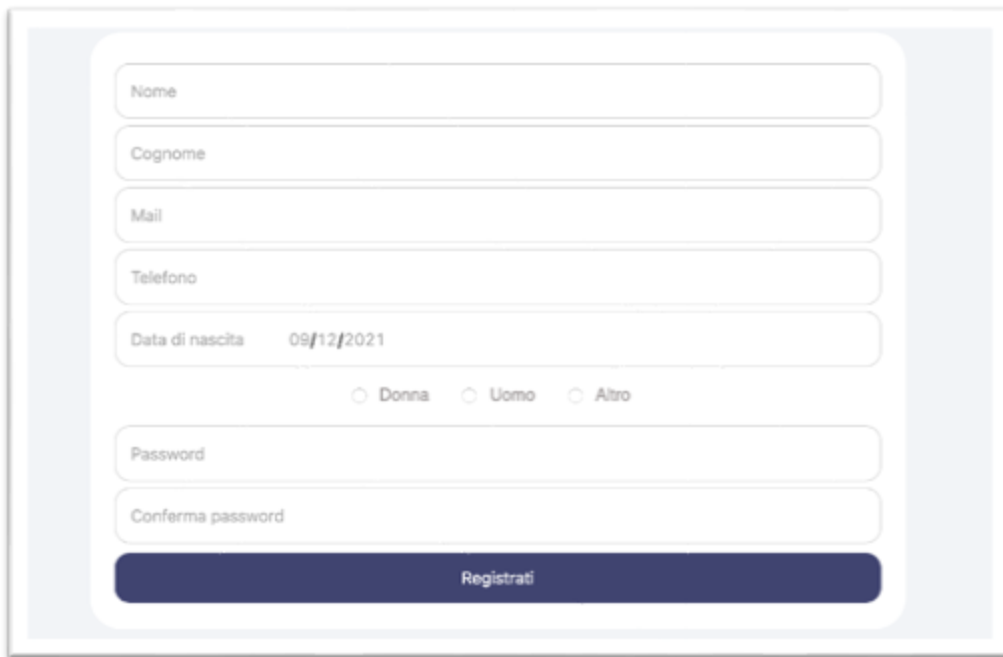
- All'interno del file **card.js** è definita la classe **Card**, la quale costruisce un elemento HTML utilizzato per visualizzare i dati relativi ad un evento o ad un biglietto. L'elemento è organizzato in tre colonne: nella colonna di sinistra viene visualizzata l'immagine dell'evento, in quella centrale il titolo e i dati salienti, in quella di destra è presente un contenitore che può essere personalizzato inserendo altri elementi HTML, come ad esempio dei pulsanti.



Figura 13 Elemento HTML che visualizza i dati dell'evento

- All'interno del file **form.js** è definita la classe **Form**, la quale crea un form HTML che permette all'utente di inserire dati sulla pagina web ed inviarli al server. I dati possono essere di qualsiasi tipo: di iscrizione, di autenticazione, o dati relativi ad un evento.



A registration form with a light blue background and rounded corners. It contains several input fields: 'Nome', 'Cognome', 'Mail', 'Telefono', 'Data di nascita' (with a date picker showing '09/12/2021'), and two password fields labeled 'Password' and 'Conferma password'. Below the date field are three radio buttons labeled 'Donna', 'Uomo', and 'Altro'. At the bottom is a dark blue button with the text 'Registrati'.

*Figura 14 Form HTML per l'inserimento dei dati*

- All'interno del file **info.js** è definita la classe **Info**. Questa permette di visualizzare un messaggio formattato all'interno di una pagina web (tag di tipo "H5").
- All'interno del file **logo.js** è definita la classe **Logo**, che permette di visualizzare il logo del sito web all'interno di una pagina HTML (viene usato nella pagina di login).



*Figura 15 Classe Logo nella pagina di login*

- All'interno del file **QRscanner.js** è definita la classe **QRscanner**. La classe implementa uno scanner di codici QR che permette di scansionare i codici dei biglietti al momento dell'ingresso all'evento.

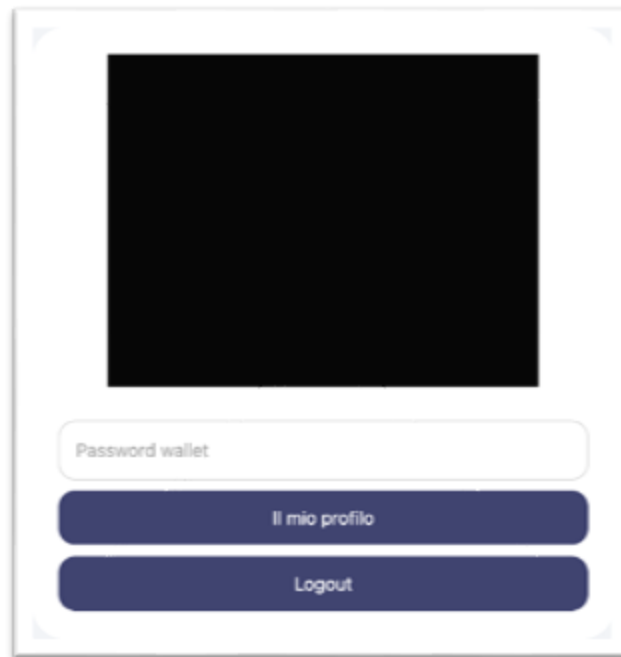


Figura 16 Scanner di codici QR utilizzato dall'annullatore

- All'interno del file **scrollView.js** è definita la classe **ScrollView**. Questa non fa altro che costruire una lista di oggetti di tipo **Card**, mostrati all'interno di una pagina web tramite una scrollview. Permette di scorrere il catalogo di eventi che si trova nella home page del sito e la lista dei biglietti acquistati da un cliente.
- All'interno del file **table.js** è definita la classe **Table**, che permette di visualizzare in una tabella i dati contenuti nel database della web app.

Nome	Cognome	Privilegi	Telefono	Mail	Data di nascita	Genere
Mario	Rossi	Annullatore	33344765432	mario.rossi@gmail.com	1970-12-24	Uomo
Giuseppe	Bianchi	Annullatore	2247634865	beppe.bianchi@gmail.com	1978-6-27	Uomo

Figura 17 Tabella generica per la visualizzazione dei dati dal database

## Form personalizzati

All'interno del sito web sono presenti svariati form che permettono di aggiungere o modificare record del database. Questi form sono implementati come classi che ereditano dalla classe più generica **Form** e sono contenuti all'interno della cartella **forms**.

- All'interno del file **formMFA.js** è definita la classe **FormMFA**. Qui è definito un particolare tipo di form, che viene mostrato all'utente quando deve inserire un codice OTP per completare la procedura di autenticazione a due fattori.

The image shows a web form for OTP verification. At the top, there is a logo consisting of a blue diamond shape next to the text "ticketTwo" in a bold, sans-serif font. Below the logo, the text "Autenticazione tramite OTP" is displayed in a smaller font. The form contains a white input field with the placeholder text "OTP". Below the input field is a dark blue button with the text "Verifica codice OTP" in white.

*Figura 18 Form per l'inserimento dell'OTP*

- All'interno del file **gestioneVendite.js** è definita la classe **FormGestioneVendite**. Nella classe è definito un particolare tipo di form, che viene visualizzato quando un membro dello staff della biglietteria richiede di aprire o di chiudere le vendite dei biglietti per un certo evento. All'interno del form è presente una casella di testo in cui l'utente deve inserire la password del proprio wallet in modo da autorizzare l'operazione.

The image shows a web form for wallet password entry. At the top, there is a logo consisting of a blue diamond shape next to the text "ticketTwo" in a bold, sans-serif font. Below the logo, the text "Gestione vendite" is displayed in a smaller font. The form contains a white input field with the placeholder text "Password wallet". Below the input field is a dark blue button with the text "Apri vendite" in white.

*Figura 19 Form per l'inserimento della password del wallet*

- All'interno del file **formModificaPassword** è definita la classe **FormModificaPassword**, nella quale è definito un particolare tipo di form, che viene mostrato all'utente quando richiede la modifica della password associata al suo account.

The image shows a web form titled "ticketTwo Il mio profilo". It contains three input fields: "Vecchia password", "Password", and "Conferma password". Below these fields is a dark blue button labeled "Modifica password".

Figura 20 Form per la modifica della password

- All'interno del file **profiloUtente.js** è definita la classe **ProfiloUtente**. Nella classe è definito un particolare tipo di form, che permette ad un utente di visualizzare i dati associati al proprio profilo.

The image shows a web form titled "ticketTwo Il mio profilo". It displays user profile information in a list format:

- Privilegi: Annullatore
- Mail: mario.rossi@gmail.com
- Nome: Mario
- Cognome: Rossi
- Data di nascita: 1970-12-24
- Genere: Uomo
- Telefono: 33344765432
- Indirizzo wallet: 0x21FD32B5725E737B1539cDEa224e420a76A1A98

At the bottom of the form are three buttons: "Modifica profilo", "Indietro", and "Modifica password".

Figura 21 Visualizzazione dei dati del proprio account

- All'interno del file **modificaProfiloUtente.js** è definita la classe **ModificaProfiloUtente**, dove viene definito un particolare tipo di form, che permette ad un utente di modificare alcuni dei dati associati al proprio profilo, incluso l'indirizzo del wallet. Non è possibile modificare né l'e-mail né i privilegi associati all'utente.



The image shows a web form for editing a user profile. At the top left is the 'ticketTwo' logo with the text 'Il mio profilo' below it. The form contains five input fields with the following values: 'Mario', 'Rossi', '26 / 11 / 1980', '3403475973', and a long alphanumeric string '0x86416F77C4EF311d993Fc38bf347681E5e113df'. Below these fields are three radio buttons labeled 'Donna', 'Uomo' (which is selected), and 'Altro'. At the bottom is a dark blue button labeled 'Salva modifiche'.

ticketTwo  
Il mio profilo

Mario

Rossi

26 / 11 / 1980

3403475973

0x86416F77C4EF311d993Fc38bf347681E5e113df

☐ Donna ☒ Uomo ☐ Altro

Salva modifiche

Figura 22 Form per la modifica dei dati del proprio account

## Scripts

- All'interno del file **bar.js** sono definite le due funzioni che permettono di aprire e chiudere il menu a tendina presente nella navigation bar del sito ticketTwo:
  - La funzione **openMenu()**, quando viene invocata, apre il menu a tendina. La funzione non fa altro che impostare a **block** il valore della proprietà CSS **display** del tag associato al menu a tendina;
  - La funzione **closeMenu()** invece chiude il menu a tendina. Questa funzione imposta a **none** il valore della proprietà CSS **display** dello stesso tag.
- All'interno del file **form.js** è definita una serie di funzioni che implementano la logica dei form dell'applicazione web:
  - La funzione **showInfo()**, quando viene invocata, mostra un messaggio di errore all'interno del form ad essa associato. Il testo del messaggio va specificato come parametro della funzione;
  - La funzione **sendDataToServer()** raccoglie il valore di tutti i campi contenuti nel form e li inserisce all'interno di un oggetto **JSON**. L'oggetto contenente i dati del form viene inviato al server tramite una richiesta HTTP. Se il server risponde che la richiesta è stata servita con successo, il client viene reindirizzato ad una nuova pagina web, altrimenti nel form viene visualizzato un messaggio che spiega il motivo per cui la richiesta è fallita. L'**url dell'api** a cui sono inviati i dati del form, il **tipo di richiesta HTTP** e l'**url di reindirizzamento** vengono specificati come parametri della funzione. Se la risposta del server contiene dei token di autenticazione, la funzione li memorizza tra i **cookies** del client;
  - La funzione **confermaPassword()** verifica se la stringa contenuta nel campo **password** del form coincide con quella contenuta nel campo **conferma password**. In caso negativo visualizza un messaggio di errore e impedisce al client di inviare il contenuto del modulo al server;
  - La funzione **read()** legge un file dal disco locale del client e ne carica il contenuto all'interno del modulo, in modo che possa essere inviato al server insieme agli altri dati. Viene utilizzato per fare l'**upload** delle immagini degli eventi;
  - La funzione **showField()** mostra una casella di testo all'interno del form quando si seleziona un determinato **radio button**. Questa viene utilizzata nella pagina per la concessione dei privilegi: quando si seleziona come **privilegio** da concedere l'opzione "**Organizzatore eventi**", essa visualizza la casella di testo in cui inserire il nome dell'organizzatore.
- All'interno del file **logout.js** è definita la funzione **logout()**, che permette di eseguire il logout dall'applicazione web. La funzione semplicemente elimina i **cookies** relativi al sito web (i token di autenticazione sono contenuti nei cookies) e lo reindirizza alla pagina di **login**. Per eliminare i cookies si imposta la loro scadenza ad una data già trascorsa (1° gennaio 1970 - Unix epoch).

- All'interno del file **QRscanner.js** sono definite le funzioni necessarie per scansionare i QR code associati ai biglietti nel momento dell'ingresso all'evento:
  - La funzione **scan()** accende la fotocamera del dispositivo del client e abilita la funzione per decodificare i QR code. Questa viene eseguita ogni 2 secondi, in maniera periodica, fino a quando non viene disabilitata;
  - La funzione **stop()** spegne la fotocamera del dispositivo del client e disabilita la funzione per decodificare i QR code;
  - La funzione **decode()** esegue la decodifica dei codici QR. Per prima cosa scatta un'istantanea del flusso video proveniente dalla fotocamera, poi effettua una chiamata alla libreria **jsqr** per tentare di decodificare il codice QR presente nell'immagine, se presente. Quando viene rilevato un QR i dati in esso contenuti vengono inviati al server, che provvederà ad eseguire l'invalidazione del biglietto.
- All'interno del file **table.js** sono definite le due funzioni che permettono di evidenziare le righe di una tabella quando ci si passa sopra con il cursore del mouse:
  - La funzione **enableHover()** modifica il colore di una riga della tabella, rendendola un po' più scura;
  - La funzione **disableHover()** ripristina il normale colore quando il cursore viene tolto dalla riga.
- La cartella **jsqr** contiene la libreria esterna che viene utilizzata lato client per scansionare i codici QR associati ai biglietti.

## Web pages

La cartella Web pages contiene cinque moduli software. In ognuno viene definita la struttura delle pagine web visualizzate dagli utenti che si interfacciano con quel modulo.

### areaRiservata.js

I moduli Amministrazione evento, Gestione biglietti e Vendite biglietti contengono un file chiamato **areaRiservata.js**. All'interno del file viene definita la classe **AreaRiservata**, implementata in maniera diversa in ciascuno dei tre moduli.

La classe eredita da **HTMLpage**. Il costruttore non fa altro che invocare il costruttore della superclasse per generare una nuova pagina web. A questa viene poi aggiunta, come figlio, la barra di navigazione che consente di navigare il catalogo ed aprire il menu a tendina.

Il menu a tendina della barra viene quindi personalizzato aggregando una serie di operazioni che l'utente può eseguire dalla propria area riservata. Le voci sono aggiunte invocando il metodo **addItem()** della classe **Bar**. Tale menu viene personalizzato in modo diverso in ciascuno dei tre moduli, poiché gli utenti che interagiscono con esso hanno privilegi diversi. Tutti comunque presentano delle operazioni comuni:

- **Visualizza profilo utente**, permette all'utente di visualizzare i dati del suo profilo ed eventualmente di modificarli;
- **Cambia password**, permette di modificare la password del proprio account;
- **Logout**, permette all'utente di eseguire il logout dall'applicazione web.

Le pagine web del sito, definite nei cinque moduli, sono rappresentate come classi javascript che ereditano dalla classe **HTMLpage** (annullatore e utente ospite) o dalla classe **AreaRiservata** (staff biglietteria, cliente, organizzatore eventi). Il corpo delle pagine web viene poi riempito aggiungendo oggetti della classe **Widget** o di sue sottoclassi.

In questa sezione non viene approfondita la struttura delle pagine web, che sono descritte in maniera più dettagliata all'interno della Guida Utente di ticketTwo.



# IMPLEMENTAZIONE DEL BACKEND DELL'APPLICAZIONE WEB

## Introduzione al backend e struttura

Il backend, assieme allo smart contract, è ciò che si nasconde dietro l'interfaccia grafica di ticketTwo.

Questo, tramite l'ausilio del frontend, gestisce tutte le interazioni tra:

- utente e applicazione
- applicazione e database
- applicazione e contratto

Il backend è strutturato secondo un pattern architetturale Model View Controller (MVC), per garantire una maggiore modularità al codice e offrire ausilio all'applicazione come webserver HTTPS, fornendo le risorse necessarie come: la logica dietro la maggior parte delle operazioni, l'accesso allo strato di persistenza dei dati (database) e l'interazione con la blockchain e lo smartcontract.

La cartella di lavoro del server è ***ticketTwo/system***, la quale contiene il modulo software principale che svolge la funzione di server. All'interno della cartella *system* sono presenti quattro subdirectory:

- **controllers**
  - Contiene tutte le logiche per interagire e gestire le risorse messe a disposizione da ticketTwo
- **functions**
  - Libreria interna del software, all'interno di questa directory sono presenti files contenenti le funzioni necessarie ai controllers e ai middlewares
- **middlewares**
  - Contiene tutti i middlewares integrati sulle rotte messe a disposizione dal server
- **model**
  - Contiene i files per interagire con lo strato di persistenza dei dati

## Librerie utilizzate e package.json

Di seguito sono riportate tutte le dipendenze dell'applicativo software direttamente scaricabili tramite npm, e quindi con il comando "npm i <nome libreria>":

- bcryptjs: <https://www.npmjs.com/package/bcryptjs>
- dotenv: <https://www.npmjs.com/package/dotenv>
- express: <https://www.npmjs.com/package/express> (<https://expressjs.com/it>)
- html-pdf: <https://www.npmjs.com/package/html-pdf>
- jsonwebtoken: <https://www.npmjs.com/package/jsonwebtoken>
- mongoose: <https://www.npmjs.com/package/mongoose> (<https://mongoosejs.com>)
- mongoose-encryption: <https://www.npmjs.com/package/mongoose-encryption>
- mongoose-sequence: <https://www.npmjs.com/package/mongoose-sequence>
- nodemailer: <https://www.npmjs.com/package/nodemailer>
- nodemon: <https://www.npmjs.com/package/nodemon>
- paypal-rest-sdk: <https://www.npmjs.com/package/paypal-rest-sdk>
- qrcode: <https://www.npmjs.com/package/qrcode>
- solc: <https://www.npmjs.com/package/solc>
- web3: <https://www.npmjs.com/package/web3>

Queste dipendenze sono tutte specificate nel file "package.json" presente nella root directory del progetto. Il file package serve per installare queste dipendenze, in modo automatico, semplicemente digitando "npm i" nel terminale.

## app.js

"app.js" è il primo file Javascript che viene eseguito in ticketTwo; esso costituisce il cuore del backend e si occupa di:

- Caricare le variabili d'ambiente presenti nel file ".env" necessarie al funzionamento di vari componenti dell'applicazione
  - Righe 7 - 11
- Istanziare un web server ssl per gestire le richieste dei client
  - Righe 16 - 30
- Effettuare la connessione al database remoto
  - Righe 35 - 43
- Mappare le rotte della web app, quindi rendere disponibili gli endpoint per accedere e manipolare le risorse
  - Riga 48 - 50

In dettaglio, prima di tutto si effettua un caricamento delle variabili d'ambiente tramite l'utilizzo della libreria "**dotenv**", che permette di caricare le variabili d'ambiente da un file ".env" (hard coded e presente nella directory principale di ticketTwo) e di metterle a disposizione nella variabile "process.env".

Saranno quindi estratte da questa variabile e inserite nelle porzioni di codice dove necessario, come ad esempio nella connessione al DB o nell'istanziamento del server ssl, fornendo rispettivamente la stringa di connessione, la chiave privata e il certificato.

```
5
6 // Carica le variabili d'ambiente =====
7 console.log('- environment variables loading...');
8
9 const dotenv = require('dotenv');
10 dotenv.config()
11 process.env.TICKET_TWO = process.cwd()+"/ticketTwo"
12 // =====
13
```

Figura 23

Successivamente l'applicazione web viene istanziata tramite l'ausilio della libreria “**Expressjs**”.

Express è un framework che permette di creare e gestire con facilità applicazioni web fornendo metodi di richiesta HTTP (come GET e POST) e middleware, in modo da creare delle API per manipolare e gestire con facilità le risorse messe a disposizione dal server.

Poiché è un'applicazione web basata su un'architettura client-server, è stato necessario abilitare lo scambio di risorse su domini differenti tramite l'utilizzo della libreria “**cors**”. In questo modo il client può richiedere le risorse nonostante si trovino in un differente dominio.

Dopo aver abilitato lo scambio di risorse e aver creato un'applicazione express, si procede con la creazione del server ssl effettivo. Come parametri sono passati la chiave privata, il certificato (entrambi generati da noi tramite open ssl) e l'applicazione web stessa, creata prima del metodo **createServer** messo a disposizione dalla libreria **HTTPS**.

```
14
15 // Crea un'applicazione web usando la libreria express =====
16 const express = require('express');
17 const cors = require('cors');
18
19 const app = express();
20 app.use(express.json({ limit: '1MB' }));
21 app.use(cors());
22 console.log('\n' + '----- | TICKETTWO\'S SERVER | -----' + '\n');
23 // =====
24
25
26 // creazione server ssl con chiave e certificati =====
27 const https = require('https');
28
29 const sslServer = https.createServer({key: process.env.RSA_PRIVATE_KEY, cert: process.env.CERTIFICATE}, app);
30 sslServer.listen(process.env.PORT, () => console.log('- listening on port ' + process.env.PORT));
31 // =====
32
```

Figura 24

Penultimo passo è il collegamento al database remoto **MongoDB** tramite l'ausilio della libreria **mongoose**. Questa, oltre ad offrire il metodo “**connect**” per effettuare la connessione tramite stringa salvata nelle variabili d'ambiente, mette a disposizione delle metodologie relativamente semplici per manipolare le risorse dell'applicazione secondo un formato non relazionale.

```
33
34 // db connection =====
35 const mongoose = require('mongoose');
36
37 mongoose.connect(
38   process.env.DB_CONNECTION_STRING,
39   { useNewUrlParser: true, useUnifiedTopology: true },
40   () => console.log('- connected to database')
41 );
42
43 mongoose.connection.on('error', console.error.bind(console, 'MongoDB connection error:'));
44 // =====
45
```

Figura 25

Infine, per accedere alle risorse e manipolarle tramite metodi http, si mappano gli endpoint (d'ora in avanti chiamati per comodità “rotte”) che sono presenti nella directory **/routes** e devono essere raggiungibili dal client tramite frontend (o il tool Postman, usato in fase di sviluppo).

```
46
47 // Mappa le rotte del sito web =====
48 console.log('- mapping routes...');
49
50 Object.entries(require("./routes")).forEach(rotta => app.use(rotta[0], rotta[1]))
51 // =====
52
```

Figura 26

## Database

La scelta del database per la nostra applicazione web è ricaduta su **MongoDB**, un database di tipo non relazionale e gratuito, che si relaziona facilmente con progetti Node.js tramite l'utilizzo della libreria **Mongoose**.

Mongodb è basato sul “NoSQL document store model” ed è quindi un modello di storage orientato ai documenti. I dati vengono salvati come **documenti**, separati all'interno di una **collezione**, secondo un formato JSON.

Possiamo interpretare i documenti come tuple e le collezioni come tabelle di un ipotetico modello relazionale tradizionale. Quindi per ogni entità esiste una collezione all'interno dello spazio di storage in cloud offerto da MongoDB.

Questo approccio è stato scelto in quanto il modello offerto da MongoDB è scalabile, efficiente, disponibile e semplice da utilizzare.

Il database è stato utilizzato per archiviare ed effettuare operazioni sui dati di:

- Utenti
- Eventi
- Biglietti
- Ricevute di pagamento
- Codici OTP
- Accessi

I documenti sono quindi istanze dei modelli e i modelli sono compilati a partire dagli schemi. Nella nostra applicazione web tutti i modelli delle entità sono disponibili nella directory ***ticketTwo/system/model***.

# MODELLI

## Descrizione del modello User

Il modello **User** serve per tenere traccia di tutti gli utenti, dei loro database, dei loro privilegi e del loro indirizzo di wallet.

Il modello possiede i seguenti attributi:

- **Privilegi**
  - Tipo: Stringa, valore default: "cliente", richiesto
  - Può assumere solamente i seguenti valori: Cliente, Organizzatore eventi, Staff biglietteria, Annullatore
  - Serve per distinguere i privilegi tra le varie tipologie di attori che possono interagire con la piattaforma web.
- **Organizzatore**
  - Tipo: Stringa, facoltativo, lunghezza min: 1 char, lunghezza max: 24 char
  - Campo per stabilire se un utente è organizzatore di un evento o meno.
- **Nome**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char
- **Telefono**
  - Tipo: Stringa, richiesto, lunghezza min: 10 char, lunghezza max: 15 char
- **Cognome**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char
- **Data\_di\_nascita**
  - Tipo: Stringa, richiesto, lunghezza min: 8 char, lunghezza max: 10 char
- **Genere**
  - Può assumere solamente i seguenti valori: Donna, Uomo, Altro
- **Mail**
  - Tipo: Stringa, richiesto, lunghezza min: 6 char, lunghezza max: 64 char
- **Password**
  - Tipo: Stringa, richiesto, lunghezza min: 8 char, lunghezza max: 100 char
  - Non contiene la password vera e propria, ma un suo hash (archiviare le password in chiaro non sarebbe sicuro).

- **Indirizzo\_wallet**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 128 char
  - Indirizzo del wallet della blockchain associato al cliente. Viene automaticamente creato un wallet appena completata la registrazione di un account.
- **registerDate**
  - Tipo: Stringa, lunghezza min: 8 char, lunghezza max: 10 char
  - La data di registrazione dell'utente.
- **registerTime**
  - Tipo: Stringa, lunghezza min: 3 char, lunghezza max: 5 char
  - L'ora di registrazione dell'utente.
- **userID**
  - Tipo: Numerico, richiesto (inserito automaticamente)
  - Chiave primaria del documento dell'Utente in MongoDB. Utilizzata per effettuare le query ed accedere al documento interessato.

I campi **userID** e **Mail** sono salvati in chiaro per facilitare l'accesso al documento.

## Descrizione del modello Event

Il modello **Event** è necessario per la gestione dei diversi tipi di eventi, già presenti o che si possono inserire nel sito, e per la gestione delle vendite dei relativi biglietti. Contiene inoltre le informazioni necessarie alla comunicazione con la blockchain.

Il modello possiede i seguenti attributi:

- **Type**
  - Tipo: Stringa, richiesto
  - Può assumere solamente i seguenti valori: Cinema, Teatro, Musei, Partite, Concerti
  - Rappresenta il tipo di evento.
- **Stato**
  - Tipo: Numerico, valore default: 0, richiesto
  - Può assumere solamente i seguenti valori numerici: 0, 1, 2
  - Serve per indicare se è aperta la vendita dei biglietti per un determinato evento; dunque, se lo stato è 0, i biglietti non sono disponibili agli utenti e l'evento non sarà visualizzato nel sito.
- **Icona\_evento**
  - Tipo: Stringa, richiesto
  - Contiene l'immagine della locandina dell'evento, memorizzata come stringa binaria base64.
- **Nome**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char
- **Luogo**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 100 char
- **Data\_evento**
  - Tipo: Stringa, richiesto, lunghezza min: 8 char, lunghezza max: 10 char
  - Data dello svolgimento dell'evento.
- **Posti\_totali**
  - Tipo: Numerico, richiesto, valore min: 1, valore max: 114000
- **Posti\_disponibili**
  - Tipo: Numerico, richiesto, valore min: 1, valore max: 114000 char
- **Orario**
  - Tipo: Stringa, richiesto, lunghezza min: 5 char, lunghezza max: 5 char
- **Organizzatore**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char



- **Artisti**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 128 char
  - Tutti gli artisti partecipanti all'evento.
- **Prezzo**
  - Tipo: Numerico, richiesto, valore min: 1, valore max: 1000
- **eventCreationDate**
  - Tipo: Stringa, lunghezza min: 8 char, lunghezza max: 10 char
  - Data di creazione dell'evento nel sito.
- **eventCreationTime**
  - Tipo: Stringa, lunghezza min: 3 char, lunghezza max: 5 char
  - Ora di creazione dell'evento nel sito.
- **ContractAbi**
  - Interfaccia ABI del contratto. Tramite l'ABI e l'indirizzo del contratto la biglietteria può richiamare i metodi necessari alla gestione dell'evento (nei limiti dei suoi privilegi).
- **Indirizzo\_contratto**
  - Tipo: Stringa, facoltativo (creazione evento offchain, non ha indirizzo), lunghezza min: 1 char, lunghezza max: 128 char
  - Indirizzo del contratto nella blockchain. Serve in combinazione con l'ABI per interagire con il contratto dell'evento tramite web3.
  - Viene istanziato un contratto per ogni evento.
- **WalletAutomaticTicketOffice & PasswordAutomaticTicketOffice**
  - Tipo: Stringa, facoltativo, lunghezza min: 1 char, lunghezza max: 128 char
  - L'emissione dei biglietti deve essere svolta in maniera automatica dal server, senza l'interazione dello staff della biglietteria, ogni volta che un cliente termina la procedura di pagamento. Prima di istanziare uno smart contract viene creato un wallet speciale a cui vengono concessi i privilegi per emettere i biglietti (wallet della biglietteria automatica). Le credenziali (indirizzo e password) vengono salvate nel DB insieme ai dati dell'evento. Solo il server sarà in grado di eseguire transazioni da questo wallet. L'esigenza di creare un wallet specifico per l'emissione dei biglietti nasce dal fatto che l'unico utente umano ad interagire col sistema durante la fase di acquisto è il cliente. Abbiamo individuato tre possibili alternative:
    - Permettere al cliente di emettere i biglietti da solo (procedura assolutamente non sicura, l'utente potrebbe emettere i biglietti senza pagarli);
    - Far svolgere l'emissione ad un membro dello staff della biglietteria (la procedura richiederebbe ogni volta l'intervento di un operatore umano; potrebbe passare molto tempo tra il pagamento e l'emissione dei biglietti);
    - Utilizzare un unico wallet per la biglietteria automatica (procedura sicura e automatica, potrebbe essere una valida alternativa)

- **eventID**

- Tipo: Numerico, richiesto (inserito automaticamente)
- Chiave primaria del documento dell'evento presente in MongoDB. Utilizzata per effettuare le query ed accedere al documento interessato.

I campi **eventID**, **type**, **stato** e **organizzatore** sono salvati in chiaro per facilitare l'accesso al documento.

## Descrizione del modello Ticket

Il modello **Ticket** è necessario per la gestione dei biglietti. Ogni biglietto è associato ad un utente e ad un evento. Tramite l'attributo `isUsed` il sistema è in grado di capire se un biglietto è stato già utilizzato.

Il modello possiede i seguenti attributi:

- **isUsed**
  - Tipo: Booleano, valore default: FALSE, richiesto
  - Booleano che indica lo stato del biglietto. Se il biglietto è invalidato è impostato a TRUE, altrimenti se il biglietto è valido è impostato a FALSE.
- **userID**
  - Tipo: Numerico, richiesto, valore min: 0
  - Codice identificativo dell'utente che ha acquistato il biglietto in questione.
- **eventID**
  - Tipo: Numerico, richiesto, valore min: 0
  - Codice identificativo dell'evento al quale l'utente vuole partecipare.
- **Codice\_identificativo**
  - Tipo: Numerico, richiesto, valore min: 0
  - Chiave primaria del documento del biglietto presente nel contratto. Serve per associare un ID univoco al biglietto emesso per un determinato evento.
- **Qrcode**
  - Tipo: Stringa, richiesto
  - Stringa contenente le informazioni per generare il codice QR, necessario all'attore Annullatore per invalidare il biglietto al momento dell'ingresso del cliente all'evento.
- **Sigillo\_fiscale**
  - Tipo: Stringa, richiesto
  - Sigillo fiscale applicato sul biglietto prima dell'emissione. Serve per confermare la validità e l'originalità del biglietto.
- **Data\_emissione**
  - Tipo: Stringa, lunghezza min: 8 char, lunghezza max: 10 char
  - Data di emissione del biglietto verso il wallet dell'utente.
- **Orario\_emissione**
  - Tipo: Stringa, lunghezza min: 3 char, lunghezza max: 5 char
  - Ora di emissione del biglietto.

- **Data\_invalidazione**
  - Tipo: Stringa, lunghezza min: 8 char, lunghezza max: 10 char
  - Data di invalidazione del biglietto, cioè dell'ingresso del cliente all'evento.
- **Orario\_invalidazione**
  - Tipo: Stringa, lunghezza min: 3 char, lunghezza max: 5 char
  - Ora di invalidazione del biglietto.
- **ticketID**
  - Tipo: Numerico, richiesto (inserito automaticamente)
  - Chiave primaria del documento del biglietto presente in MongoDB. Utilizzata per effettuare le query ed accedere al documento interessato.

Va specificato che questo modello contiene due ID relativi allo stesso biglietto: **Codice\_identificativo** e **ticketID**.

Il **codice identificativo** è un ID associato dallo smart contract al biglietto e quindi corrisponde ad un numero intero che va da zero al numero di posti disponibili per quell'evento (meno uno). Esso rappresenta l'ID del biglietto relativo ad un determinato evento e di conseguenza è possibile che due eventi distinti abbiano biglietti con codici identificativi identici. Questo ID è necessario all'annullatore per invalidare il biglietto a livello di blockchain.

Il **ticketID** è la chiave primaria effettiva del biglietto nel database ed è univoca per ogni biglietto presente nel database. Serve dunque per effettuare operazioni di lettura e scrittura sul record interessato.

I campi **userID**, **eventID** e **ticketID** sono salvati in chiaro per facilitare l'accesso al documento.

## Descrizione del modello **Receipt**

Il modello **Receipt** è necessario per la gestione delle ricevute di pagamento dei clienti. Le ricevute vengono emesse dopo l'avvenuto acquisto di un biglietto. Comprende le informazioni relative al cliente che lo ha acquistato, alla quantità di biglietti presi e all'evento al quale intende partecipare.

Il modello possiede i seguenti attributi:

- **Codice\_ricevuta**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 128 char
  - Chiave primaria della ricevuta.
- **Metodo\_di\_pagamento**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char
  - Metodo di pagamento adottato dall'utente. Essendo il sistema di pagamento fittizio, è valido solamente il metodo Paypal.
- **Via**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 40 char
- **Città**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char
- **Stato**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char
- **CAP**
  - Tipo: Stringa, richiesto, lunghezza min: 4 char, lunghezza max: 16 char
- **E-mail**
  - Tipo: Stringa, richiesto, lunghezza min: 6 char, lunghezza max: 64 char
- **Nome**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char
- **Cognome**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char
- **eventID**
  - Tipo: Numerico, richiesto, valore min: 0
  - corrispettivo evento del biglietto.
- **userID**
  - Tipo: Numerico, richiesto, valore min: 0
  - chiave primaria dell'utente che ha acquistato il biglietto.
- **Prezzo**
  - Tipo: Numerico, richiesto, valore min: 1, valore max 1000
- **Valuta**
  - Tipo: Stringa, richiesto, lunghezza min: 1 char, lunghezza max: 24 char
- **Numero\_biglietti**
  - Tipo: Numerico, richiesto, valore min: 1, valore max: 4
  - Numero di biglietti richiesti dal cliente.
- **Data\_emissione**
  - Tipo: Stringa, lunghezza min: 8 char, lunghezza max: 10 char

- **Orario\_emissione**
  - Tipo: Stringa, lunghezza min: 3 char, lunghezza max: 5 char
- **receiptID**
  - Tipo: Numerico, richiesto (inserito automaticamente)
  - Chiave primaria del documento della ricevuta presente in MongoDB. Utilizzata per effettuare le query ed accedere al documento interessato.

I campi **eventID** e **receiptID** sono salvati in chiaro per facilitare l'accesso al documento.

## Descrizione del modello OTP

Il modello **OTP** serve per gestire i codici OTP inviati all'utente che sta effettuando operazioni critiche, cioè quando è necessaria una misura di sicurezza ulteriore.

Il modello possiede i seguenti attributi:

- **userID**
  - Tipo: Numerico, richiesto, valore min: 0
  - ID dell'utente al quale inviare il codice di sicurezza.
- **OTP**
  - Tipo: Stringa, richiesto, valore min: 4, valore max: 4
  - Il codice di sicurezza da inviare all'utente.

Il campo **userID** è salvato in chiaro per facilitare l'accesso al relativo documento.

Da notare che non è presente una chiave primaria, poiché questo documento è temporaneo e viene eliminato dopo l'utilizzo.

## Descrizione del modello **Access**

Il modello **Access** serve per tenere traccia degli accessi al sito. In particolare, memorizza data e ora di accesso di ogni utente iscritto.

Il modello possiede i seguenti attributi:

- **Mail**
  - Tipo: Stringa, richiesto, lunghezza min: 6 char, lunghezza max: 64 char
  - E-mail dell'utente che effettua l'accesso al sito.
- **Data\_accesso**
  - Tipo: Stringa, richiesto, lunghezza min: 8 char, lunghezza max: 10 char
  - Data dell'accesso al sito da parte dell'utente.
- **Orario\_accesso**
  - Tipo: Stringa, richiesto, lunghezza min: 3 char, lunghezza max: 5 char
  - Orario dell'accesso al sito.
- **AccessID**
  - Tipo: Numerico, richiesto (inserito automaticamente)
  - Chiave primaria del documento istanziato dal modello Access presente in MongoDB. Utilizzata per effettuare le query ed accedere al documento interessato.

In questo documento nessun campo è salvato in chiaro.



## Analisi dettagliata di uno schema (Access)

Di seguito è riportata un'analisi dettagliata di un modello; più precisamente si prenderà in esame il modello Access, descritto precedentemente, che serve per tenere traccia degli accessi al sito.

La sintassi degli schemi è semplice e ricorrente ed è molto simile alla definizione di un oggetto JSON. Per definire uno schema si deve utilizzare il metodo **mongoose.Schema({})** e definire all'interno delle parentesi graffe gli attributi dell'entità che si intende modellare.

Nell'immagine sottostante troviamo lo schema come definito nel codice:

```
4 // modello per tracciare gli accessi
5 const accessSchema = new mongoose.Schema({
6
7   Mail: {
8     type: String,
9     min: 6,
10    max: 64,
11    required: true
12  },
13
14  Data_accesso: {
15    type: String,
16    required: true
17  },
18
19  Orario_accesso: {
20    type: String,
21    required: true
22  }
23
24 });
```

Figura 27

Tramite mongoose è semplice specificare dei vincoli sugli attributi che compongono l'entità, come ad esempio il **tipo** di dato, la **lunghezza minima** del dato e quella **massima**. Inoltre, il campo **required** impostato a *true* rende necessaria la presenza dell'attributo durante l'istanziamento del modello, altrimenti sarà restituito un errore a runtime.

Ogni documento ha un proprio ID di default, secondo un formato stringa, ma non è utile per effettuare delle query ed interrogare il database in maniera efficace e veloce. Quindi, tramite l'ausilio del plugin "AutoIncrement" per mongoose, ovviamo al problema inserendo degli ID incrementali che partono da zero; questi verranno ampiamente utilizzati per effettuare operazioni di lettura e scrittura sui dati.

```
25  
26 accessSchema.plugin(AutoIncrement, {inc_field: 'accessID'});  
27
```

Figura 28

Perciò si deve specificare quale sarà l'attributo che si auto incrementerà di 1 ad ogni istanziazione di un nuovo documento. Il campo viene aggiunto anch'esso allo schema del modello.

Mentre, per criptare i dati e renderli illeggibili anche da parte di un amministratore di sistema, è stato utilizzato un ulteriore plugin per mongoose noto come "Encryption".

```
27  
28 accessSchema.plugin(Encryption, {  
29   encryptionKey: process.env.AES_256_CBC,  
30   signingKey: process.env.HMAC_SHA_512  
31 });  
32
```

Figura 29

Qui si devono specificare l'algoritmo di cifratura e la chiave di decifratura come parametri. Questi sono salvati come variabili d'ambiente nel file ".env".

L'operazione cripterà tutti gli attributi di una entità e vanno specificati i campi utili che dovranno essere tralasciati dall'operazione. In questo esempio viene cifrato tutto il documento. Invece nella figura sottostante, presa dallo schema "User", si possono vedere i campi esclusi dalla cifratura:

```
106  
107 userSchema.plugin(Encryption, {  
108   encryptionKey: process.env.AES_256_CBC,  
109   signingKey: process.env.HMAC_SHA_512,  
110   excludeFromEncryption: ['userID', 'Mail']});  
111
```

Figura 30

Una volta definiti tutti i campi dello schema è necessario compilarlo in modo da poterlo utilizzare ed istanziare. Per farlo si usa il comando **mongoose.model()**, che richiede come parametri il nome del modello e lo schema appena creato.

```
32  
33   module.exports = mongoose.model('Access', accessSchema);  
34
```

*Figura 31*

Va specificato che, per comodità, in ogni file presente nella directory “model” compiliamo ed esportiamo gli schemi nella stessa riga.

Infine, basterà importare il modello nella sezione di codice interessata ed utilizzare mongoose per effettuare operazioni CRUD a database.

## CONTROLLERS

I controller sono i principali moduli software che gestiscono le interazioni con la nostra applicazione web. Questi agiscono su richiesta degli attori, tramite gli endpoint “nascosti” dietro l’interfaccia grafica messa a disposizione dal sito.

Esistono diversi moduli, ognuno dei quali gestisce risorse diverse in base alle entità prese in considerazione:

- UserController
- AuthController
- EventController
- TicketController
- ReceiptController

Ogni controller utilizza funzioni e moduli software aggiuntivi presenti nella directory ***system/functions***. In questo modo i controller diventano più modulari e flessibili, in quanto per aggiungere o rimuovere una funzione basterà semplicemente aggiungerla o rimuoverla dalla suddetta directory.

Nella nostra applicazione web tutti i controller sono disponibili nella directory ***ticketTwo/system/controllers***.

## Descrizione del controller UserController

Il modulo software **UserController** prende in carico la **gestione degli utenti** ed offre i metodi per interagire con essi. Questi metodi ricoprono la ricerca degli utenti a DB, la ricerca dei loro accessi al sito, l'aggiornamento dei loro privilegi e la modifica della loro password.

Va notato che non sono presenti i metodi della registrazione (quindi della creazione) di un utente e del login, perché sono presenti nel controller AuthController, che appunto prende in carico le registrazioni e le autenticazioni.

I metodi presenti in UserController, esternamente richiamabili dagli endpoint, sono i seguenti:

- **getAllUsers()**

Questa funzione invoca la funzione **find()**, definita all'interno della libreria dell'applicazione web (cartella *functions*), per trovare i record degli utenti di interesse.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un array contenente tutti gli utenti presenti nel database. Se la richiesta fallisce, viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Non richiede alcun parametro.

- **getAllAccess()**

Questa funzione, come la precedente, utilizza il metodo **find()** presente nella libreria interna dell'applicazione stessa per trovare i record relativi agli accessi degli utenti.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un array contenente tutti gli accessi effettuati dagli utenti presenti nel database. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Non richiede alcun parametro.

- **getUser(id)**

Questa effettua delle query interrogando il database per trovare i dati relativi all'utente richiesto, utilizzando il metodo **findOne()** presente nella libreria interna al fine di trovare lo specifico utente registrato nel database.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e l'utente individuato nel database. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- l'ID dell'utente di cui si vogliono ottenere i dati

- **getAnnullatori()**

Questa funzione è identica a **getAllUsers()** ed utilizza allo stesso modo la funzione **find()**. La differenza consiste nell'effettuare la ricerca interrogando il database e **filtrando** gli utenti che hanno il privilegio di annullatore.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un array contenente tutti gli annullatori presenti nel database. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Non richiede alcun parametro.

- **updateUserData(id, newParams)**

Questa funzione esegue una query nel database per aggiornare i dati di un utente, facendo riferimento alla funzione **update()** presente nella libreria interna all'applicazione per **trovare ed aggiornare** il profilo di un utente specifico. La funzione restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- l'ID dell'utente che vuole aggiornare i dati del proprio profilo
- un JSON contenente i nuovi dati da sostituire ai vecchi, restituisce lo stato della richiesta.

- **updatePrivileges(data)**

La funzione interroga il database sfruttando la funzione **update** per aggiornare i privilegi di un utente. Se l'operazione di modifica ha avuto successo, viene inviata una e-mail di conferma nella casella di posta dell'utente che l'ha richiesta; la funzione restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- JSON contenente e-mail, organizzatore e privilegi

- **modificaPassword(userID, userData)**

Funzione che permette ad un utente di modificare la password del proprio account. Anche questa fa riferimento al metodo **update()**, che non permette l'inserimento di una password identica e, se l'operazione ha successo, cripta la nuova password e la salva nel database.

La funzione restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- ID dell'utente che intende aggiornare la password
- JSON contenente la vecchia e la nuova password.

- **recuperaPassword(email)**

Funzione che permette ad un utente di recuperare la password del proprio account, tramite la generazione di una password che viene inviata all'utente tramite mail. Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- l'e-mail dell'utente che desidera recuperare la password

## Descrizione del controller **AuthController**

Il modulo software **AuthController** prende in carico la **registrazione** o l'**autenticazione** degli utenti che desiderano registrarsi o accedere a ticketTwo; inoltre genera e verifica la correttezza dei codici **OTP** emessi dal sito stesso. Come il precedente, anche questo modulo ha la necessità di interagire con la libreria *functions* interna del sistema, dove trova tutti i metodi di cui ha bisogno.

I metodi presenti in AuthController, esternamente richiamabili dagli endpoint, sono i seguenti:

- **Login(loginData)**

È la funzione che permette agli utenti di autenticarsi ed accedere al sito dopo che si sono registrati; controlla le credenziali di accesso inserite da un utente, ovvero e-mail e password. Se le credenziali sono corrette, sarà assegnato un token di autenticazione all'utente (JsonWebToken) che viene quindi autenticato. Quando l'utente è in possesso del token può compiere più azioni di un semplice utente non loggato e può quindi interagire con alcune rotte private.

Poiché il token firmato con un segreto **TOKEN\_SECRET** è presente nelle variabili d'ambiente del sistema, tutti possono leggere il contenuto del token, ma non possono alterarlo in quanto la signature non corrisponderebbe più con quella salvata.

Il Token contiene i seguenti campi dell'utente che ha effettuato il login, ovvero che risulta essere autenticato:

- userID
- e-mail
- privileges
- walletAddress
- Organizzatore

Il login inoltre tiene conto degli accessi effettuati al sito: ogni volta che un utente effettua l'operazione di login invocando il suddetto metodo, viene aggiunto un nuovo documento al database contenente data, ora e ID dell'utente.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo), un messaggio che notifica l'esito positivo dell'operazione ed il token di autenticazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- Un JSON contenente le credenziali d'accesso dell'utente (e-mail e password)

- **Register(registerData)**

È la funzione che permette agli utenti di registrarsi dopo che hanno inserito i dati nel form di registrazione di ticketTwo. Questa funzione si occupa quindi di memorizzare i dati d'iscrizione del nuovo utente all'interno del database.

Prima si controlla che l'e-mail inserita dall'utente non sia già stata utilizzata, poi si controlla che la password digitata contenga almeno 8 caratteri, con inclusi almeno un carattere speciale ed una maiuscola, e si genera un wallet da assegnare all'account. Successivamente, si passano i dati del form e il wallet alla funzione



**create()**, che permette di memorizzarli sul database. Infine, viene inviata una e-mail all'utente per informarlo dell'avvenuta registrazione al sito.

Il wallet è generato grazie alla funzione **createWallet** definita in *functions/wallet*, che si occupa di generare una password casuale e di istanziare un nuovo account nella blockchain (con la password appena creata) tramite la funzione **newAccount** messa a disposizione da web3.

Va notato che la password dell'utente, prima di essere scritta nel database, viene criptata tramite la funzione **bcrypt.hash**, che si occupa di effettuare l'hashing della password con **salt** per evitare attacchi a dizionario. Inoltre, come già detto nella descrizione del modello user, tutta la collezione è criptata e gli unici campi che sono salvati in chiaro dell'utente sono userID ed e-mail.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) ed un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- Un JSON contenente i dati necessari all'iscrizione, ovvero:
  - Nome
  - Cognome
  - Telefono
  - Data\_di\_nascita
  - Genere
  - Indirizzo\_wallet (non inserito dall'utente)
  - E-mail
  - Password

- **generateOTP(userID, email)**

Questa funzione si occupa di generare i codici temporanei OTP che sono inviati all'utente tramite e-mail durante le operazioni più critiche. Il codice viene creato tramite una funzione di generazione casuale di password a 4 cifre. Successivamente al suo invio si procede all'eliminazione di tutti i precedenti codici OTP richiesti dall'utente e presenti nel database, tramite la funzione **deleteMany()** definita tra le funzioni di libreria, la quale permette di eliminare in cascata diversi documenti. Questa fase di eliminazione risulta necessaria, poiché un utente potrebbe richiedere un codice OTP e non accedere alla risorsa protetta. I codici OTP si rendono necessari in fasi critiche come, ad esempio, la concessione dei privilegi agli utenti.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) ed un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- userID dell'utente
- e-mail dell'utente.

- **getTokenOTP(userID, insertOTP)**

Questa funzione si occupa di verificare la correttezza del codice OTP inserito dall'utente. Se il codice OTP risulta essere corretto, viene rilasciato all'utente un token di autenticazione (JWT) di secondo livello, che potrà utilizzare per accedere alle rotte protette da autenticazione a due fattori. Successivamente al controllo della correttezza del token, si elimina dal database l'OTP relativo all'utente perché non più necessario.

Il JWT in questo caso contiene i seguenti campi:

- userID
- OTP, memorizzato sul database e inviato all'utente tramite e-mail, deve coincidere con il codice che l'utente inserisce nel form. Se l'utente inserisse un codice errato, prima di generare il token verrebbe lanciata un'eccezione.

In sostanza il token contiene gli stessi dati del modello OTP.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo), un messaggio che notifica l'esito positivo dell'operazione, nonché il token di autenticazione multi-fattore.

Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- userID dell'utente
- il codice OTP che l'utente ha inserito.

## Descrizione del controller **EventController**

Il modulo software **EventController** ha in carico tutta la parte della **gestione degli eventi**, dalla loro creazione alla loro amministrazione. In particolare, questo modulo rende possibile alcune operazioni fondamentali di ticketTwo come l'istanziamento di un nuovo evento, l'aggiornamento di un evento e l'apertura o chiusura delle vendite dei biglietti.

I metodi presenti in EventController sono i seguenti:

- **getEvent(id, privileges)**

Funzione che interroga il database per restituire l'evento associato all'ID passato per parametro. È necessario come parametro anche il privilegio dell'utente poiché, se colui che richiede la risorsa ha i privilegi da Cliente, l'evento viene restituito solo se le vendite dei biglietti sono aperte, mentre, se ha i privilegi da Event Manager, un evento viene restituito anche se le vendite dei biglietti non sono state ancora aperte.

Restituisce un JSON contenente i dati dell'evento richiesto.

Richiede come parametri:

- l'ID dell'evento da cercare
- i privilegi dell'utente che richiede la risorsa.

- **getEventByType(type, user)**

Funzione che restituisce una lista di eventi contenente tutti quelli di una certa categoria. Il funzionamento è identico a quello di getEvent(), ma con la differenza che la ricerca a database viene effettuata tramite il tipo di un singolo evento e non tramite l'ID.

Restituisce quindi una lista di JSON contenente gli eventi corrispondenti alla ricerca.

Richiede come parametri:

- il tipo dell'evento da cercare
- i privilegi dell'utente che richiede la risorsa.

- **createEvent(eventData, organizzatore)**

Questa funzione si occupa di istanziare nel database un documento relativo ad un evento che si intende creare nel sito web. Si occupa di reperire i dati, immessi nel form dall'organizzatore o dalla biglietteria, e di scriverli nel database tramite il metodo **create()** messo a disposizione nella libreria.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) ed un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- JSON contenete i dati dell'evento:

- Tipo dell'evento
- Icona dell'evento (ovvero il percorso del file della locandina)
- Nome
- Luogo
- Artisti partecipanti
- Data dell'evento
- Posti disponibili totali
- Posti disponibili rimasti
- Prezzo

- Organizzatore dell'evento.

Da notare che la creazione dell'evento è un'operazione che avviene offchain, in quanto l'istanziamento del contratto relativo ad un evento avviene solamente quando saranno aperte le vendite.

- **updateEvent(eventData, eventId)**

Funzione che permette l'aggiornamento dei dati di un evento per il quale non sono ancora state aperte le vendite. La funzione sfrutta il metodo **update()**, a cui viene passata la chiave primaria dell'evento da aggiornare e i nuovi campi da sostituire ai vecchi.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) ed un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- un JSON contenente i campi dell'evento che si desidera aggiornare
- l'ID dell'evento da aggiornare.

Da notare che l'operazione avviene sempre offchain.

- **deleteEvent(id)**

Funzione che interroga il database per cancellare un dato evento. Questo viene trovato ed eliminato tramite la funzione **deleteOne** definita all'interno della libreria di sistema. L'eliminazione è possibile solamente se le vendite dei biglietti per quell'evento sono chiuse.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) ed un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- l'ID dell'evento da aggiornare

Anche questa operazione avviene offchain.

- **apriVendite(eventID, annullatore, manager\_wallet, manager\_password)**

Funzione che apre le vendite per un determinato evento, solo se le vendite dei biglietti sono chiuse, istanziando quindi il relativo smart contract nella blockchain. Una volta fatto, i dati dell'evento non saranno più disponibili per la modifica.

La funzione prevede:

- la **compilazione del contratto** tramite la funzione **compile()** definita in *functions/contract*, che restituisce ABI e bytecode del contratto;
- la **generazione di un wallet** associato all'evento, per permettere alla biglietteria automatica di emettere i biglietti per l'evento;
- il **deploy su blockchain** del contratto tramite la funzione **deploy()**, che sblocca il wallet del manager con le sue credenziali, effettua il deploy istanziando un nuovo contratto, passando l'array args come parametri del costruttore, e blocca il wallet del manager una volta effettuato il deploy;
- l'**aggiornamento dello stato delle vendite**, da chiuso ad aperto, relativo al contratto.

Infine, vengono effettuate delle query per salvare l'indirizzo e l'ABI del contratto nel database, poi delle query per il salvataggio dell'indirizzo del wallet, abilitato all'emissione biglietti, e della relativa password.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) ed un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- l'ID dell'evento di cui si vogliono aprire le vendite
- l'indirizzo del wallet dell'annullatore dei biglietti
- l'indirizzo e la password del wallet della biglietteria che istanzia lo smart contract.

- **chiudiVendite(eventID, manager\_wallet, manager\_password)**

Questa funzione esegue una transazione verso il contratto e permette la chiusura delle vendite dei biglietti per un dato evento. Quindi, sfruttando il metodo **update**, aggiorna il relativo smart contract e lo stato dell'evento salvato nel database. Tutti i tentativi di emissione biglietto successivi alla chiusura delle vendite falliranno. Da questo momento in poi, gli annullatori sono abilitati ad eseguire l'invalidazione dei biglietti.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) ed un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- l'ID dell'evento di cui si vogliono chiudere le vendite
- l'indirizzo e la password del wallet della biglietteria che esegue la transazione.

## Descrizione del controller **TicketController**

Il modulo software **TicketController** si occupa della gestione delle richieste inerenti alla visualizzazione, l'emissione e l'invalidazione dei biglietti. Le richieste che provengono dal client vengono valutate e sanificate per poi essere eventualmente inoltrate allo smart contract che gira sulla blockchain o al database.

I metodi presenti in **TicketController** sono i seguenti:

- **getTicket()**

Funzione che interroga il database per ottenere tutti i dati relativi ad un determinato biglietto (dati del possessore, dell'evento, sigillo fiscale, orario di emissione...). Invoca la funzione **findOne()**, definita nella libreria interna dell'applicazione web (cartella *functions*), per trovare il record associato al biglietto cercato in base al suo **ID**. Il metodo poi verifica se l'utente che richiede di visualizzare i dati del biglietto sia anche colui che lo ha acquistato; in caso negativo la richiesta fallisce. Vengono eseguite altre due interrogazioni al database, stavolta per ottenere i dati relativi all'evento associato al biglietto e quelli relativi al suo possessore. Tutte e tre le interrogazioni restituiscono come risultato un oggetto JSON. I tre JSON vengono fusi insieme ottenendo una descrizione più dettagliata del biglietto.

Richiede come parametri:

- L'ID dell'utente che effettua la richiesta di interrogare il database
- L'ID del biglietto di cui si vogliono ottenere i dati.

La funzione restituisce come risultato una tupla (array) contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un JSON contenente i dati del biglietto. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

- **getTicketsByUser()**

La funzione interroga il database per ottenere tutti i biglietti acquistati da un dato utente. Invoca la funzione **find()**, definita nella libreria interna dell'applicazione web (cartella *functions*), per trovare tutti i record associati ad un certo **user id** (codice univoco assegnato all'utente che possiede il biglietto). La lista di record ottenuta viene poi filtrata scartando i record associati a biglietti già invalidati (campo **isUsed** con valore **true**). Viene eseguita un'altra interrogazione al database, stavolta per ottenere i dati relativi all'evento associato al biglietto. Tutte e due le interrogazioni al database restituiscono come risultato un oggetto JSON. I due JSON vengono fusi insieme ottenendo una descrizione più dettagliata del biglietto.

Richiede come parametro:

- L'ID dell'utente che effettua la richiesta di interrogare il database

La funzione restituisce come risultato una tupla (array) contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e una lista (array) contenente tutti i biglietti posseduti dall'utente. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

- **getTicketsByEvent()**

La funzione interroga il database per ottenere tutti i biglietti emessi per un certo evento. Il metodo invoca la funzione **find()**, definita nella libreria interna dell'applicazione web (cartella *functions*), per trovare tutti i record associati ad un certo **eventid** (codice univoco assegnato agli eventi disponibili sul sito).

Richiede come parametri:

- l'ID dell'evento per cui si vuole ottenere un elenco di tutti i biglietti emessi

La funzione restituisce come risultato una tupla (array) contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e una lista (array) contenente tutti i biglietti emessi per l'evento considerato. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

- **getInvalidatedTicketsByEvent()**

La funzione è identica a **getTicketsByEvent()**, ma prima di restituire il risultato filtra l'elenco dei biglietti scartando quelli che non sono stati ancora invalidati. La lista ottenuta dà informazioni sugli ingressi ad un evento.

- **richiestaBiglietto()**

La funzione controlla la disponibilità sulla blockchain dei biglietti voluti dall'utente, poi genera una richiesta di pagamento che viene inoltrata al servizio PayPal. Se l'utente seleziona un metodo di pagamento diverso da PayPal, viene restituito il **codice HTTP 406** insieme ad un messaggio d'errore. Se sulla blockchain non è disponibile alcun **token** da emettere (numero posti disponibili minore del numero di biglietti richiesti dal cliente), la transazione per verificare la disponibilità dei biglietti fallisce e viene restituito il codice HTTP 500 insieme ad un messaggio d'errore. Lo stesso codice viene restituito anche quando fallisce l'inoltro della richiesta di pagamento a PayPal. In caso di successo viene restituito il codice HTTP 200 insieme al link PayPal su cui eseguire il pagamento.

Richiede come parametri:

- il metodo di pagamento selezionato (PayPal, Visa o Mastercard)
- l'ID dell'utente che vuole acquistare i biglietti
- l'ID dell'evento per cui si vogliono acquistare i biglietti
- il numero di biglietti che si vuole acquistare
- l'indirizzo e la password del wallet del cliente.

- **emissioneBiglietti()**

La funzione esegue la transazione di acquisto addebitando il costo dei biglietti sul conto PayPal del cliente, emette i biglietti associati alla richiesta d'acquisto e invia una e-mail all'utente contenente la ricevuta di pagamento. L'emissione dei biglietti consta di più passi:

- Viene creato un **nuovo record** sul database per memorizzare i dati del biglietto;

- Viene emesso un **token**, che ricoprirà il ruolo di gemello digitale del biglietto cartaceo;
- Il **codice univoco** associato al token viene inserito nel record del database;
- Al record viene aggiunto un **sigillo fiscale** e un **QR code** contenente i dati del biglietto;
- Sul database viene decrementato il **numero di posti disponibili**;
- Il record viene effettivamente salvato sul database.

Se l'utente che richiede l'emissione dei biglietti non è lo stesso che ha effettuato la richiesta di pagamento, viene restituito il **codice HTTP 401** insieme ad un messaggio d'errore. Se l'emissione del token sulla blockchain fallisce, viene ugualmente restituito il codice HTTP 401 insieme ad un messaggio d'errore. In caso di successo viene restituito il codice HTTP 200 insieme ad un JSON contenente i dati della ricevuta di pagamento.

Richiede come parametri:

- l'ID PayPal dell'utente che ha acquistato i biglietti
- l'ID del pagamento effettuato dal cliente
- l'e-mail del cliente (dove viene inviata la ricevuta di pagamento)
- l'ID dell'utente che ha acquistato i biglietti
- l'indirizzo del suo wallet.

- **invalidaBiglietto()**

La funzione verifica l'integrità e l'autenticità del sigillo fiscale, poi invoca il metodo dello smart contract che si occupa dell'invalidazione del **token** associato al biglietto. Terminata l'operazione, viene aggiornato il database aggiungendo l'orario e la data di invalidazione. Se il sigillo fiscale non è autentico o l'invalidazione del token sulla blockchain fallisce, viene restituito il codice HTTP 500 insieme ad un messaggio d'errore. In caso di successo viene restituito il codice HTTP 200 insieme ad un JSON contenente i dati del biglietto invalidato.

Richiede come parametri:

- i dati del biglietto codificati all'interno del QR code
- l'indirizzo del wallet dell'annullatore dei biglietti
- la password del wallet dell'annullatore dei biglietti.



## Descrizione del controller **ReceiptController**

Il modulo software **ReceiptController** si occupa della **gestione delle ricevute di pagamento** restituite dal sistema PayPal. Questo è un modulo relativamente semplice, in quanto è composto da due metodi soltanto che servono per ottenere e memorizzare le informazioni relative alle ricevute.

I metodi presenti in **ReceiptController** sono i seguenti:

- **getReceiptsByEvent(eventID)**

La funzione si occupa di restituire tutte le ricevute di pagamento relative ad un certo evento tramite la funzione **find()** definita all'interno della libreria.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) ed un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri:

- l'ID dell'evento di cui si vogliono visualizzare le ricevute

- **emissioneRicevuta(paymentData, email)**

La funzione memorizza nel database la ricevuta di pagamento ottenuta da PayPal dopo l'avvenuto acquisto di uno o più biglietti. Inoltre, si occupa anche di inviare al cliente una e-mail con le informazioni della transazione appena effettuata.

Questa funzione, tramite il metodo **create** definito nella libreria di sistema, genera nel database un documento contenente le informazioni relative a:

- codice della ricevuta
- metodo di pagamento
- utente che esegue il pagamento
- indirizzo di spedizioni
- transazione
- orario e data di emissione della ricevuta.

Restituisce come risultato un array contenente lo stato della richiesta (codice HTTP 200 in caso di successo) ed un messaggio che notifica l'esito positivo dell'operazione. Se la richiesta fallisce viene restituito lo stato di fallimento (codice HTTP 500) e una stringa contenente un messaggio di errore.

Richiede come parametri un JSON contenente:

- info sul codice della ricevuta
- info sul metodo di pagamento
- info sull'utente che esegue il pagamento (e-mail, nome, cognome)
- info sull'indirizzo di spedizione (via, città, stato, cap)
- info sulla transazione (eventID, userID, prezzo, valuta e numero biglietti)
- info su orario e data di emissione della ricevuta.

## MIDDLEWARES

Un middleware è un software che si trova al centro di altri due strati software. I middleware Express sono funzioni che vengono eseguite durante il ciclo di vita di una richiesta al server Express.

Un middleware può essere aggiunto ad ogni route che vogliamo sia protetta o privata, o comunque che implementi un livello di sicurezza maggiore rispetto alle altre rotte. Ognuno ha accesso alla richiesta e alla risposta HTTP per ogni rotta al quale è collegato. Inoltre, può terminare la richiesta HTTP o passarla a un'altra funzione middleware usando il **next()**. Questo "concatenamento" di middleware conferisce al software una grande modularità, dal momento che li rende comodamente riutilizzabili.

Tutti i middleware implementati in ticketTwo si trovano nella directory **system/middlewares** e sono i seguenti:

- verifyToken
- checkLogin
- checkPrivileges
- verifyOTP

### Descrizione del middleware verifyToken

verifyToken ha lo scopo di verificare la **validità del token di autenticazione**; rende le rotte protette e accessibili solamente agli utenti autenticati. Il middleware cerca tra i **cookies** allegati alla richiesta il token "auth-token" generato durante la fase di login e ne verifica l'autenticità; se il token è valido l'utente ottiene l'accesso a queste risorse protette, in caso contrario non potrà entrare nel percorso indicato.

### Descrizione del middleware checkLogin

CheckLogin è un middleware che gestisce gli **accessi alla pagina iniziale** (ovvero la pagina disponibile agli utenti non autenticati) ed è presente alla riga 15 nel file *routes/webpages/events*.

Il middleware controlla se l'utente è autenticato, andando a verificare la presenza del token di autenticazione "**auth-token**" nei **cookies** allegati alla richiesta HTTP.

Se il token viene trovato, l'utente è reindirizzato nella propria area riservata, altrimenti viene eseguita la **next function** che lo reindirizza alla home page pubblica, quella accessibile dagli **utenti guest** (di default saranno visualizzati gli eventi di tipo cinema, a meno che non sia specificata a priori un'altra categoria da visualizzare).

## Descrizione del middleware **checkPrivileges**

CheckPrivileges ha lo scopo di rendere le **rotte accessibili** solamente agli utenti che sono in possesso di determinati **privilegi**. I percorsi protetti da checkPrivileges non possono essere utilizzati se l'utente non è in possesso dei privilegi necessari.

Questo middleware controlla se il tipo di privilegio dell'utente sia adeguato, in tal caso viene garantito l'accesso alla risorsa chiamando la funzione next(). I privilegi dell'utente sono specificati all'interno del suo token di autenticazione. Il middleware viene invocato dopo **verifyToken**, cioè quando il token è già stato decodificato e l'utente autenticato.

Invece, se il privilegio dell'utente corrisponde a quello dell'annullatore, tale middleware si occuperà di reindirizzare l'annullatore verso l'appropriata pagina per annullare i biglietti, subito dopo il login.

Se l'utente non possiede il token per accedere alla risorsa, allora verrà reindirizzato alla home page (area riservata).

CheckPrivileges è presente in tutte le rotte, tranne quelle pubbliche. Ad esempio, si trova in quella relativa alla registrazione o quella riguardante il recupero della password, poiché un qualsiasi attore dovrebbe essere in grado di registrarsi al sito o di recuperare la propria password in caso di problemi.

## Descrizione del middleware **verifyOTP**

Questo middleware ha lo scopo di verificare la validità del token di autenticazione multi-fattore. In sostanza, è identico a verify token, solo che verifica i codici **OTP**.

Se il token è valido permette l'accesso alla rotta, altrimenti esegue la funzione **generateOTP()**, che genera un codice OTP di 4 cifre e lo invia per e-mail all'utente, e reindirizza il client alla pagina con il form in cui inserire il codice OTP.

La funzione **eval()** esegue il codice che gli viene passato per parametro, il quale deve essere una stringa in cui sono scritte le istruzioni javascript da eseguire. In questo caso viene costruita una stringa diversa a seconda dei privilegi del cliente:

- Se l'utente ha privilegi da cliente, viene costruito l'oggetto **MFA\_Cliente**;
- Se l'utente ha privilegi da biglietteria, viene costruito l'oggetto **MFA\_Staff\_biglietteria**.

La logica usata è la stessa di una banale clausola **if**. A seconda dei privilegi viene costruita ed inviata al client la pagina giusta. La differenza tra le due pagine web risiede nel menu a tendina, accessibile dalla barra di navigazione: utenti con privilegi diversi accederanno a un menu con voci differenti.

# SMART CONTRACT

## Lo Smart Contract

Uno **smart contract** è una macchina a stati, che di volta in volta aggiorna il suo stato interno in base alle operazioni che avvengono sui suoi attributi. Per facilità di comprensione si può vedere come un programma che gira su una blockchain (nel nostro caso GoQuorum). Esso non è altro che un insieme di codice (le sue funzioni) e di dati (il suo stato) che risiede in un indirizzo specifico della blockchain dopo l'avvenuto deployment.

I contratti non sono controllati da un utente, ma sono distribuiti sulla rete e si occupano solo di rispondere agli input in base ai loro metodi e alla loro programmazione; non possono essere cancellati e le interazioni con essi sono irreversibili. Dunque, le interazioni non avvengono direttamente con il contratto stesso, ma tramite l'ausilio del server di ticketTwo, che si occupa appunto di sfruttare i suoi metodi e le risorse messe a disposizione da esso.

Nel nostro caso, per sviluppare ticketTwo abbiamo implementato un solo contratto denominato **ticketOffice.sol** presente nella directory ***TicketTwo/system/functions***.

Il contratto è stato pensato per essere compilato e deployato dagli attori che possiedono il ruolo di biglietteria, dopo che le vendite di biglietti per un determinato evento sono state aperte tramite l'interfaccia grafica.

Lo scopo del contratto è quello di gestire le transazioni che avvengono sul nostro sito a livello di blockchain e di renderle quindi immutabili, trasparenti e tracciabili grazie alle primitive crittografiche intrinseche alla blockchain stessa.

Inoltre, il contratto funge anche da database distribuito in quanto tiene traccia di tutti gli eventi, dei relativi biglietti emessi e degli utenti che li hanno acquistati. Per tenere conto di tutto questo, il contratto possiede al suo interno alcune apposite strutture dati, ovvero:

- **Evento:** per tenere traccia degli eventi, dei posti totali, di quelli disponibili e dello stato delle vendite dei biglietti per un determinato evento;
- **Biglietto:** per tenere traccia dei biglietti emessi, verso quali utenti sono stati emessi e della validità degli stessi.

```

5 .....
6 .....//strutture dati interne al contratto.....
7 .....struct Evento{
8 .....    uint id;
9 .....    uint totali;
10 .....    uint disponibili;
11 .....    bool venditeAperte;
12 .....}
13 .....
14 .....struct Biglietto{
15 .....    uint id;
16 .....    bool isUsed;
17 .....    address cliente;
18 .....}
19 .....

```

Figura 32

Queste due strutture sono state poi istanziate ed utilizzate assieme ad altri attributi, per garantire il funzionamento del contratto. Complessivamente gli attributi del contratto sono i seguenti:

```

22 .....
23 .....//attributi
24 .....Evento private evento;
25 .....mapping(address => uint) public richieste_biglietti;
26 .....uint private idBiglietto = 0;
27 .....Biglietto[] private bigliettiEmessi;
28 .....
29 .....address private operatore_biglietteria;
30 .....address private annullatore;
31 .....address private biglietteria_automatica;
32 .....

```

Figura 33

## Descrizione degli attributi del contratto

- **Evento private evento**
  - come specificato sopra, tiene traccia dei dettagli sullo stato dell'evento che si sta svolgendo
- **mapping(address=> uint) public richieste\_biglietti**
  - attributo che associa ad ogni indirizzo wallet dell'utente il numero di biglietti da lui richiesti; serve per emettere tutti i biglietti richiesti verso l'indirizzo appropriato

- **uint private idBiglietto = 0**
  - ID del biglietto che sarà emesso dal contratto e sarà comunicato al server tramite l'evento **EmissioneTerminata(address cliente, uint idBiglietto)**
- **Biglietto[] private bigliettiEmessi**
  - Array di oggetti di tipo Biglietto: serve per tenere traccia di tutti gli stati dei biglietti emessi e verso quali utenti. Grazie alla funzione **creaNuovoBiglietto(uint \_id, address \_cliente)**, ogni emissione di biglietto inserisce nell'array una nuova entry con i dati necessari
- **address private operatore\_biglietteria**
  - variabile di tipo address che memorizza l'indirizzo della biglietteria. Viene assegnato dal costruttore al momento dell'apertura delle vendite dei biglietti
- **address private annullatore**
  - variabile di tipo address che memorizza l'indirizzo dell'annullatore. Viene assegnato dal costruttore al momento dell'apertura delle vendite dei biglietti
- **address private biglietteria\_automatica**
  - variabile di tipo address che memorizza l'indirizzo della biglietteria automatica (wallet del server)

## Descrizione del costruttore

```

35 // costruttore
36 constructor(uint _postiTotali, uint _id, address _annullatore, address _biglietteria_automatica){
37
38     evento.id = _id;
39     evento.totali = _postiTotali;
40     evento.disponibili = _postiTotali;
41     evento.venditeAperte = true;
42     operatore_biglietteria = msg.sender;
43     annullatore = _annullatore;
44     biglietteria_automatica = _biglietteria_automatica;
45
46 }
47

```

Figura 34

Il **costruttore** è il metodo che viene richiamato durante l'istanziamento del contratto sulla blockchain. Si occupa di impostare il valore base di alcuni attributi del contratto che gli vengono passati dal server. Questi sono:

- **evento.id**

- l'ID che rappresenta l'evento per il quale si intende istanziare lo smart contract
- **evento.totali**
  - i posti totali massimi dell'evento
- **evento.disponibili**
  - viene impostato con lo stesso valore dei posti totali, in quanto all'apertura delle vendite i posti disponibili sono quelli totali
- **annullatore**
  - l'indirizzo dell'annullatore autorizzato ad invalidare i biglietti all'ingresso
- **biglietteria\_automatica**
  - l'indirizzo della biglietteria automatica; è l'unico wallet abilitato ad eseguire l'emissione dei biglietti. La biglietteria normale istanzia lo smart contract e apre o chiude le vendite, mentre la biglietteria automatica emette solamente i biglietti.

Da notare che nel costruttore vengono impostati anche altri due attributi:

- **evento.venditeAperte**
  - rende possibile l'emissione dei biglietti ai clienti per l'evento con l'ID corrispondente
- **operatore\_biglietteria**
  - l'indirizzo della biglietteria che può compiere le azioni riservate ai suoi privilegi. Questo è l'indirizzo che il contratto contatta per primo (quindi lo istanzia)

## Functions

La cartella **functions** contiene una libreria di funzioni utilizzate dal backend di ticketTwo per eseguire una serie di operazioni, come interfacciarsi con la blockchain, con il database o il sistema di pagamento estemo, e gestire l'emissione dei biglietti.

### generateRandomPassword.js

Il modulo **generateRandomPassword.js** si occupa della generazione di stringhe di numeri casuali. In questo file è definita una singola funzione, **generateRandomPassword()**, che richiede come parametro la lunghezza della stringa casuale da generare e restituisce come risultato la stringa creata.

Le operazioni eseguite dalla funzione sono molto semplici. Viene definita una stringa letterale contenente tutte le cifre decimali e una variabile in cui andare a memorizzare la stringa casuale, inizialmente vuota. Questa viene generata estraendo ripetutamente in modo aleatorio una cifra, da 0 a 9, e aggiungendola ad un buffer (la stringa vuota definita in precedenza). L'estrazione continua fino a quando non si ottiene una stringa della lunghezza desiderata.

L'estrazione casuale di cifre avviene utilizzando la libreria **Math** di Node.js.

### checkPassword.js

Il modulo **checkPassword.js** si occupa di verificare la validità delle password inserite dagli utenti al momento della registrazione o del cambio password. In questo file è definita una singola funzione, **checkPassword()**, che richiede come parametro la password che si vuole controllare.

La funzione controlla se la password specificata soddisfa i seguenti criteri:

- Lunghezza di almeno **8 caratteri**;
- Presenza di almeno un **carattere speciale**;
- Presenza di almeno una **lettera maiuscola**.

Se la password non soddisfa i precedenti criteri sarà giudicata debole e verrà lanciata un'eccezione.

### wallet.js

Il modulo **wallet.js** si occupa della gestione dei wallet sulla blockchain, necessari per istanziare smart contract o per eseguire transazioni. Nella nostra web app ad ogni utente è associato un wallet.

Il modulo stabilisce una connessione con un nodo della blockchain, che funge da **provider**, tramite la libreria **web3-eth-personal**. Su questo nodo saranno memorizzati tutti i dati relativi ai wallet generati dall'applicazione web. L'**RPC URL** del nodo a cui il modulo si connette è memorizzato tra le variabili d'ambiente del server; può essere modificato configurando il file **.env** presente nella directory principale dell'applicazione.



Nel modulo sono definite tre funzioni:

- **createWallet()** genera un nuovo wallet sulla blockchain associandogli una password casuale. Questa è una stringa numerica di 6 cifre, generata usando la funzione **generateRandomPassword** definita nel modulo **generateRandomPassword.js**. Il wallet viene effettivamente generato invocando la funzione **newAccount()** fornita dalla libreria **web3.js**. La funzione non richiede alcun parametro e restituisce un oggetto contenente l'indirizzo del nuovo wallet e la password necessaria per sbloccarlo;
- **lockWallet()** permette di bloccare il wallet di un utente in modo tale da impedire che questo possa effettuare transazioni. La funzione contiene una chiamata al metodo **lockAccount()**, fornito dalla libreria **web3.js**, richiede come parametro l'indirizzo del wallet da bloccare e restituisce come risultato **undefined**;
- **unlockWallet()**, al contrario, permette di sbloccare il wallet di un utente in modo da consentire al suo possessore di effettuare nuovamente transazioni. La funzione contiene una chiamata al metodo **unlockAccount()** fornito dalla libreria **web3.js**, richiede come parametri indirizzo e password del wallet da sbloccare e restituisce come risultato **undefined**;
- **isAccount()** verifica se il wallet che viene passato come parametro è registrato all'interno della blockchain a cui il server è connesso.

## contract.js

Il modulo **contract.js** si occupa della distribuzione e dell'interazione con gli smart contract sulla blockchain.

Il modulo stabilisce una connessione con un nodo della blockchain, che funge da **provider**, tramite la libreria **web3-eth-contract**. Su questo nodo saranno memorizzate tutte le transazioni eseguite sugli smart contract. L'**RPC URL** del nodo, a cui il modulo si connette, è memorizzato tra le variabili d'ambiente del server e può essere modificato configurando il file **.env** presente nella directory principale dell'applicazione.

Nel modulo sono definite tre funzioni:

- **compile()** esegue la compilazione dello smart contract. Il sorgente è memorizzato nel file **ticketOffice.sol** contenuto nella stessa directory del modulo e viene letto utilizzando la funzione **readFileSync()**, definita nella libreria **fs**, che permette di interagire con il file system del server in Node.js. Il sorgente viene compilato invocando il metodo **compile()** della libreria **solc** (compilatore solidity). Tutto il codice viene eseguito all'interno di un blocco **try**. La funzione non richiede alcun parametro e restituisce come risultato un oggetto contenente **ABI** e **bytecode** dello smart contract. Se durante l'esecuzione della funzione viene lanciata un'eccezione, questa sarà catturata e verrà restituito **undefined**;
- **deploy()** esegue la distribuzione di uno smart contract sulla blockchain a cui è connessa l'applicazione web. Per prima cosa viene sbloccato il wallet dell'utente che vuole istanziare lo smart contract (un wallet bloccato non può eseguire transazioni, nemmeno il deploy). A questo punto viene istanziato un oggetto

della classe **web3-eth-contract** passando come parametro l'**ABI** del contratto da istanziare. Per effettuare il deploy del contratto è necessario generare un oggetto di tipo transazione, che può essere ottenuto invocando il metodo **deploy()** sull'istanza appena creata e passando come parametri il **bytecode** del contratto e gli argomenti del costruttore. Per eseguire effettivamente il deploy sulla blockchain si invoca il metodo **send()** dell'oggetto transazione creato, passando come parametro l'indirizzo del wallet. Tutto il codice viene eseguito all'interno di un blocco **try**. La funzione richiede come parametri:

- l'**indirizzo e la password del wallet** di chi istanzia lo smart contract;
- l'**ABI e il bytecode dello smart contract** da istanziare;
- gli **argomenti** da passare al costruttore dello smart contract.

La funzione restituisce come risultato l'indirizzo sulla blockchain dello smart contract appena distribuito. Se durante l'esecuzione della funzione viene lanciata un'eccezione, questa verrà catturata e sarà restituito **undefined**. Insieme al try catch è presente anche una clausola **finally** che viene sempre eseguita, sia quando si verifica un'eccezione sia quando avviene la normale esecuzione. All'interno della clausola finally viene invocata la funzione per bloccare lo smart contract. In questo modo, a prescindere dall'esito della transazione, lo smart contract rimane sempre protetto.

- **eseguiTransazione()** permette di invocare un metodo dello smart contract distribuito sulla blockchain. Per prima cosa viene sbloccato il wallet dell'utente che vuole eseguire la transazione (un wallet bloccato non può eseguire transazioni). A questo punto viene istanziato un oggetto della classe **web3-eth-contract**, passando come parametro l'**ABI** del contratto distribuito e il suo indirizzo sulla blockchain. Così si ottiene il riferimento ad un contratto distribuito in precedenza. L'attributo **methods** di questo oggetto contiene un dizionario con tutti i metodi dello smart contract. Inserendo il nome del metodo che si vuole invocare come chiave del dizionario viene restituito un oggetto transazione. Per eseguire effettivamente la transazione sulla blockchain si invoca il metodo **send()** dell'oggetto transazione appena creato, passando come parametro l'indirizzo del wallet. Tutto il codice viene eseguito all'interno di un blocco **try**. La funzione richiede come parametri:

- l'**indirizzo e la password del wallet** di chi chiama il metodo dello smart contract;
- l'**indirizzo e l'ABI dello smart contract** con cui si vuole interagire;
- il **nome del metodo** che si vuole chiamare;
- gli **argomenti** da passare al metodo dello smart contract.

La funzione restituisce come risultato un oggetto contenente i dati della transazione appena eseguita. Se durante l'esecuzione della funzione viene lanciata un'eccezione, questa sarà catturata e verrà restituito **undefined**. Insieme al try catch è presente anche una clausola **finally** che viene sempre eseguita, sia quando si verifica un'eccezione sia quando si ha una normale esecuzione della funzione. All'interno della clausola finally viene invocata la funzione per bloccare lo smart contract. In questo modo, a prescindere dall'esito della transazione, lo smart contract rimane sempre protetto.

## query.js

Il modulo **query.js** si occupa dell'interazione tra l'applicazione web ed il database. In particolare, il modulo implementa una serie di metodi che permettono di interrogare il database per creare, aggiornare, cancellare o trovare un record del database.

Le funzioni definite nel modulo sono dei **wrapper** che contengono chiamate alle **API**, le quali permettono di interrogare un database su MongoDB. Le API sono definite nella libreria esterna **Mongoose**.

Nel modulo sono definite cinque funzioni:

- **create()** permette di aggiungere un nuovo record al database. Tutto il codice viene eseguito all'interno di un blocco **try**. La funzione richiede come parametri:
  - il **modello** del record contenente lo schema con cui organizzare i dati all'interno del database (è rappresentato come un oggetto di tipo **Schema**, definito nella libreria esterna **Mongoose**);
  - i **dati** da inserire all'interno del record, codificati in formato JSON (la chiave, ovvero il nome del campo del record, e il suo valore);
  - una **callback** da eseguire prima di salvare il record sul database.

La funzione crea un nuovo schema per il record e inserisce tutti i dati al suo interno. Successivamente viene eseguita la callback passata come parametro e salvato il record sul database invocando il metodo **save()** dell'oggetto schema. La funzione restituisce come risultato una tupla (array) contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un riferimento allo **schema del record**. Se durante l'esecuzione della funzione viene lanciata un'eccezione, questa viene catturata e viene restituita una tupla contenente lo stato di fallimento (codice HTTP 500) e una stringa che specifica il tipo di errore che si è verificato.

- **deleteOne()** permette di eliminare un record dal database. Tutto il codice viene eseguito all'interno di un blocco **try**. La funzione richiede come parametri:
  - il **modello** del record contenente lo schema con cui sono organizzati i dati all'interno del database (è rappresentato come un oggetto di tipo **Schema**, definito nella libreria esterna **Mongoose**);
  - la **query** da eseguire per trovare il record (è rappresentata da un JSON, dove le coppie chiave-valore esprimono i filtri da utilizzare per individuare il record).

La funzione trova il primo record del database che soddisfa la query e lo rimuove. Se il record non viene trovato la funzione non fa nulla. Restituisce come risultato una tupla (array) contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un **messaggio** che indica che l'operazione è stata eseguita con successo. Se durante l'esecuzione della funzione viene lanciata un'eccezione, questa viene catturata e viene restituita una tupla contenente lo stato di fallimento (codice HTTP 500) e una stringa che specifica il tipo di errore che si è verificato.

- **find()** permette di trovare tutti i record del database che soddisfano determinate condizioni. Il codice viene eseguito all'interno di un blocco **try**. La funzione richiede come parametri:
  - il **modello** del record contenente lo schema con cui sono organizzati i dati all'interno del database (è rappresentato come un oggetto di tipo **Schema**, definito nella libreria esterna **Mongoose**);
  - la **query** da eseguire per trovare tutti i record che soddisfano determinate condizioni (è rappresentata da un JSON, dove le coppie chiave-valore esprimono i filtri da utilizzare per individuare i record);
  - una **callback** da eseguire al termine dell'interrogazione al database.

La funzione interroga il database per individuare tutti i record che soddisfano la query. Successivamente viene eseguita la callback passata come parametro. La funzione restituisce come risultato una tupla (array) contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e una **lista di tutti i record** (array) che soddisfano la query. Se durante l'esecuzione della funzione viene lanciata un'eccezione, questa viene catturata ed è restituita una tupla contenente lo stato di fallimento (codice HTTP 500) e una stringa che specifica il tipo di errore che si è verificato.

- **findOne()** permette di trovare un record del database che soddisfa determinate condizioni. Tutto il codice viene eseguito all'interno di un blocco **try**. La funzione richiede come parametri:
  - il **modello** del record contenente lo schema con cui sono organizzati i dati all'interno del database (è rappresentato come un oggetto di tipo **Schema**, definito nella libreria esterna **Mongoose**);
  - la **query** da eseguire per trovare il primo record che soddisfa determinate condizioni (è rappresentata da un JSON, dove le coppie chiave-valore esprimono i filtri da utilizzare per individuare i record);
  - una **callback** da eseguire al termine dell'interrogazione al database.

La funzione interroga il database per individuare il primo record che soddisfa la query. Successivamente viene eseguita la callback passata come parametro. La funzione restituisce come risultato una tupla (array) contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e il **record** (codificato in formato JSON) che soddisfa la query. Se durante l'esecuzione della funzione viene lanciata un'eccezione, questa viene catturata e viene restituita una tupla contenente lo stato di fallimento (codice HTTP 500) e una stringa che specifica il tipo di errore che si è verificato.

- **update()** permette di aggiornare i dati contenuti all'interno di un record del database. Tutto il codice viene eseguito all'interno di un blocco **try**. La funzione richiede come parametri:
  - il **modello** del record contenente lo schema con cui sono organizzati i dati all'interno del database (è rappresentato come un oggetto di tipo **Schema**, definito nella libreria esterna **Mongoose**);
  - la **query** da eseguire per trovare il primo record che soddisfa determinate condizioni (è rappresentata da un JSON, dove le coppie chiave-valore esprimono i filtri da utilizzare per individuare i record);
  - una **callback** da eseguire al termine dell'interrogazione al database.

La funzione interroga il database per individuare il primo record che soddisfa la query. Successivamente viene eseguita la callback passata come parametro. La callback è necessaria per modificare i dati contenuti nel record. Una volta eseguita la callback, si salvano le modifiche sul database invocando il metodo **save()** dell'oggetto schema. La funzione restituisce come risultato una tupla (array) contenente lo stato della richiesta (codice HTTP 200 in caso di successo) e un riferimento allo **schema del record**. Se durante l'esecuzione della funzione viene lanciata un'eccezione, questa viene catturata ed è restituita una tupla contenente lo stato di fallimento (codice HTTP 500) e una stringa che specifica il tipo di errore che si è verificato.

## ticket.js

Il modulo **ticket.js** fornisce dei metodi per eseguire alcune operazioni critiche durante la fase di emissione ed invalidazione dei biglietti.

Nel modulo sono definite quattro funzioni:

- **apposizioneSigillo()** genera un sigillo fiscale autentico e lo allega al biglietto appena emesso. Il biglietto è codificato come un oggetto in formato JSON. Il sigillo viene generato facendo un **hash** del biglietto per poi calcolarne la **firma** (hash and sign). Entrambe le operazioni vengono eseguite invocando la funzione **sign()** fornita dalla libreria **crypto**. Il sigillo generato è espresso come stringa binaria esadecimale che viene aggiunta al JSON del biglietto. L'hash del biglietto viene calcolato utilizzando l'algoritmo **SHA-256** e firmato usando **RSA**. La chiave privata usata per generare la firma è memorizzata come variabile d'ambiente. La funzione richiede come parametro il biglietto in formato JSON su cui apporre il sigillo. Non restituisce alcun valore.
- **generazioneQRcode()** genera un QR code contenente tutti i dati del biglietto e lo allega al biglietto appena emesso, che viene codificato come un oggetto in formato JSON. Il codice QR è generato invocando la funzione **toDataURL()** della libreria esterna **qrcode**. Questa restituisce un'immagine contenente il codice QR codificata come una stringa binaria **base64**. La funzione richiede come parametro il biglietto in formato JSON su cui apporre il codice QR. Non restituisce alcun valore.
- **generazioneCodiceIdentificativo()** aggiunge al biglietto un codice identificativo da utilizzare come riferimento al suo gemello digitale (token) presente sulla blockchain. Al termine dell'emissione del biglietto, lo smart contract genera un **token non fungibile (NFT)** e un evento per indicare che l'emissione è avvenuta con successo. La funzione recupera il codice identificativo a partire dall'evento prodotto dallo smart contract e lo allega al biglietto. Richiede come parametro il biglietto in formato JSON su cui apporre il codice identificativo e la transazione in cui avviene l'emissione del token. Non restituisce alcun valore.
- **verificaSigillo()** verifica l'autenticità e l'integrità del sigillo fiscale allegato ad un biglietto. A partire dal JSON contenente i dati del biglietto, viene estrapolato il sigillo fiscale e convertito in un oggetto di tipo buffer. Il sigillo viene poi dato in pasto alla funzione **verify()** definita nella libreria esterna **crypto**, che "decifra" la firma contenuta nel sigillo fiscale e la confronta con l'hash attuale del biglietto per vedere se i due coincidono. La chiave pubblica usata per generare la firma è memorizzata come variabile d'ambiente. La funzione richiede come parametro il biglietto in formato JSON. Non restituisce alcun valore.

## paypal.js

Il modulo **paypal.js** fornisce i metodi necessari per interfacciare l'applicazione web con il sistema di pagamento esterno. Il modulo stabilisce una connessione con la sandbox di **PayPal** utilizzata per testare le transazioni dei pagamenti, utilizzando il **client ID** ed il **client secret** ad essa associati. Entrambi sono memorizzati come variabili d'ambiente, che possono essere configurate modificando il file **.env** presente nella directory principale dell'applicazione.

Le funzioni definite nel modulo sono dei **wrapper** che contengono chiamate alle **API**, le quali permettono di interagire con il servizio di pagamento PayPal. Le API sono definite nella libreria esterna **paypal-rest-sdk**.

Nel modulo sono definite due funzioni:

- **richiestaPagamento()** genera una richiesta di pagamento che viene inoltrata ai server di PayPal, invocando la funzione **payment.create()** definita nella libreria esterna **paypal-rest-sdk**. La funzione richiede come parametri:
  - L'**ID dell'utente** che effettua la richiesta di pagamento;
  - L'**ID dell'evento** per cui l'utente vuole acquistare i biglietti;
  - Il **prezzo** di un singolo biglietto;
  - Il **numero dei biglietti** che l'utente vuole acquistare.

La funzione restituisce come risultato il link PayPal su cui eseguire il pagamento, che viene generato dinamicamente una volta approvata la richiesta.

- **eseguiPagamento()** va ad addebitare concretamente il costo dei biglietti sul conto PayPal dell'acquirente, invocando la funzione **payment.execute()** definita nella libreria esterna **paypal-rest-sdk**. La funzione richiede come parametri:
  - L'**ID dell'utente** che esegue il pagamento;
  - L'ID associato alla **richiesta di pagamento** dei biglietti.

La funzione restituisce come risultato una ricevuta di pagamento contenente tutti i dati relativi all'acquisto dei biglietti. L'autenticità della ricevuta è garantita da PayPal.

### [getToken.js](#)

Il modulo **getToken.js** si occupa dell'estrazione dei token di autenticazione a partire dalla richiesta HTTP inviata da un client. In questo file è definita una singola funzione, **getToken()**, che richiede come parametro la richiesta HTTP inviata dal client e il tipo di token da cercare (token di autenticazione standard o multi-fattore).

La funzione estrapola i **cookies** allegati alla richiesta del client, che sono contenuti nell'intestazione, e ne esegue la verifica per controllare se sia presente o meno il token di autenticazione richiesto.

La funzione restituisce il token, se questo viene trovato, altrimenti restituisce undefined.

### [mailer.js](#)

Il modulo **mailer.js** si occupa dell'invio di e-mail informative agli utenti che sono iscritti al sito web. Le e-mail vengono inviate al momento dell'iscrizione al sito, a quello della concessione di privilegi ad un utente o al termine della procedura di acquisto dei biglietti per un evento.

Il modulo stabilisce una connessione con una casella di posta **Gmail**, che funge da mittente per ogni e-mail inviata in maniera automatica dal sito. Le credenziali di autenticazione associate all'e-mail sono salvate come variabili d'ambiente; possono essere modificate configurando il file **.env** presente nella directory principale dell'applicazione.

In questo file è definita una singola funzione, **sendMail()**, che richiede come parametri:

- l'**oggetto** della mail da inviare;
- il **destinatario** della mail da inviare;
- il **testo** della mail da inviare.

La funzione non restituisce alcun valore.

### [timeFunctions.js](#)

Il modulo **timeFunctions.js** definisce una libreria di funzioni che permettono di ottenere la data e l'ora corrente.

Nel modulo sono definite due funzioni:

- **getCurrentDate()** restituisce la data corrente, cioè la data in cui si invoca la funzione.
- **getCurrentTime()** restituisce l'ora corrente, cioè l'orario esatto in cui si invoca la funzione.