functions

La cartella **functions** contiene una libreria di funzioni utilizzate dal backend dell’applicazione web per eseguire una serie di operazioni come interfacciarsi con la blockchain, il database e il sistema di pagamento esterno, gestire l’emissione dei biglietti …

generateRandomPassword.js

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

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 da generare, inizialmente vuota.

La stringa casuale viene generata estraendo ripetutamente una cifra da 0 a 9 aleatoriamente e aggiungendole ad un buffer (la stringa vuota definita precedentemente). 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.

wallet.js

Il modulo **wallet.js** si occupa della gestione degli wallet sulla blockchain. Gli wallet sono necessari per istanziare smart contract sulla blockchain o per eseguire transazioni con essi. 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 agli 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 e può essere modificato configurando il file **.env** presente nella directory principale dell’applicazione.

Nel modulo sono definite tre funzioni:

* La funzione **createWallet()** genera un nuovo wallet sulla blockchain associandogli una password casuale. La password casuale 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 nessun parametro e restituisce come risultato un oggetto contenente l’indirizzo del wallet generato e la password necessaria per sbloccarlo.
* La funzione **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**. La funzione richiede come parametro l’indirizzo del wallet da bloccare e restituisce come risultato **undefined**.
* La funzione **unlockWallet()** permette di sbloccare il wallet di un utente in modo tale da consentire nuovamente al suo possessore di effettuare transazioni. La funzione contiene una chiamata al metodo **unlockAccount()** fornito dalla libreria **web3.js**. La funzione richiede come parametro l’indirizzo del wallet da bloccare e la password e restituisce come risultato **undefined**.

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:

* La funzione **compile()** esegue la compilazione dello smart contract che si occupa della gestione dei biglietti per un evento. Il sorgente dello smart contract è memorizzato nel file **ticketOffice.sol** contenuto nella stessa directory del modulo. Il sorgente 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 poi 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 nessun parametro e restituisce come risultato un oggetto contenente **abi** e **bytecode** dello smart contract.Se durante l’esecuzione della funzione della funzione viene lanciata un’eccezione questa viene catturata e viene restituito **undefined**.
* La funzione **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 nessuna transazione, 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 appena istanziato 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 appena creato, passando come parametro l’indirizzo del wallet che istanzia il contratto, precedentemente sbloccato. Tutto il codice viene eseguito all’interno di un blocco **try**. La funzione richiede come parametri:
  + - l’**indirizzo del wallet** di chi istanzia lo smart contract;
    - la **password del wallet** di chi istanzia lo smart contract;
    - l’**abi** **dello smart contract** da istanziare;
    - 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 della funzione viene lanciata un’eccezione questa viene catturata e viene 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.

* La funzione **eseguiTransazione()** permette di invocare un metodo di uno 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 nessuna transazione). 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. In questo modo si ottiene un 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 che chiama il metodo, precedentemente sbloccato. Tutto il codice viene eseguito all’interno di un blocco **try**. La funzione richiede come parametri:
  + - l’**indirizzo del wallet** di chi chiama il metodo dello smart contract;
    - la **password del wallet** di chi chiama il metodo dello smart contract;
    - l’**indirizzo dello smart contract** con cui si vuole interagire;
    - 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 della funzione viene lanciata un’eccezione questa viene catturata e viene 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** che permettono di interrogare un database su MongoDB. Le API sono definite nella libreria esterna **Mongoose**.

Nel modulo sono definite cinque funzioni:

* La funzione **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 indica il nome del campo del record, mentre il valore indica il valore del campo);
  + 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 si salva 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.

* La funzione **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. La funzione 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 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.

* La funzione **find()** permette di trovare tutti i record del database che soddisfano 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 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 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.

* La funzione **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.

* La funzione **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 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.

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:

* la funzione **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.
* la funzione **generazioneQRcode()** genera un **QR code** contenente tutti i dati del biglietto e lo allega al biglietto appena emesso. Il biglietto è codificato come un oggetto in formato JSON. Il codice QR viene 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.
* la funzione **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 sulla blockchain lo smart contract genera un **token non fungibile (NFT)** ed emette un’evento per indicare che l’emissione è avvenuta con successo. La funzione recupera il codice identificativo a partire dall’evento emesso dallo smart contract e lo allega al biglietto. La funzione 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.
* la funzione **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 relative ai 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** che 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:

* La funzione **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.

* La funzione **eseguiPagamento()** va effettivamente ad addebitare il costo dei biglietti al conto PayPal dell’utente che li richiede, 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 èdefinita 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 il parking per verificare 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 mail informative agli utenti del sito web. Le mail vengono inviate al momento dell’iscrizione al sito, al momento 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 tutte le mail inviate in maniera automatica dal sito. Le credenziali di autenticazione associate alla mail sono salvate come variabili d’ambiente e possono essere modificate configurando il file **.env** presente nella directory principale dell’applicazione.

In questo file èdefinita 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:

* La funzione **getCurrentDate()** restituisce la data corrente, cioè la data del momento esatto in cui si invoca la funzione.
* La funzione **getCurrentTime()** restituisce l’ora corrente, cioè l’orario esatto in cui si invoca la funzione.