Architettura e scelte tecnologiche

Architettura software

Durante la stesura dei diagrammi i\*, in particolare nello **strategic rational 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 software esterni e di controllare e gestire gli accessi alle risorse della web app.

Gli altri moduli fungono da interfaccia tra gli attori esterni 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**.

Operazioni come l’emissione e l’invalidazione dei biglietti sono molto critiche e delicate. Questi servizi devono essere caratterizzata da alta disponibilità. Inoltre, i dati e il codice associati a questi servizi devono rispettare i requisiti di autenticità e integrità. L’impiego di una **blockchain** e degli **smart contract** è senz’altro la scelta più adatta per implementare questi servizi. In particolare la scelta è ricaduta su una blockchain privata permissioned.

L’utilizzo di una blockchain privata permissioned ci permette di definire policy più stringenti, poiché ci da la possibilità di eseguire transazioni private, di scegliere i nodi validatori e gli utenti della rete.

I dati relativi alle transazioni e lo stato degli smart contract in una blockchain sono ridondati e distribuiti. Ogni nodo della blockchain possiede una **copia del registro distribuito**. Memorizzare tutti i dati necessari al funzionamento della web app sulla blockchain sarebbe molto oneroso. Abbiamo scelto di memorizzare nella blockchain solo i dati critici, mettendo tutto il resto all’interno di un database.

L’utilizzo di un database semplifica anche la gestione delle procedure di autenticazione e autorizzazione necessarie per proteggere l’accesso alle risorse.

Il modulo **System** viene, quindi, progettato come composizione di due diverse architetture:

* un **Distributed ledger (DLT)**, per implementare la componente software che si occupa dell’emissione e dell’invalidazione dei biglietti;
* Un’architettura **Model View Controller (MVC)**, per gestire la comunicazione con il database della web app.

Scelte tecnologiche

Scelta della blockchain

La prime scelte tecnologica da effettuare 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**. Per configurare e distribuire la nostra blockchain Quorum abbiamo deciso di utilizzare il tool a riga di comando **Quorum-wizard**.

Questo tool ci ha permesso di stanziare 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 del contratto.

L’utilizzo della blockchain garantisce ridondanza, diversità e distribuzione (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. NodeJs presenta diversi vantaggi:

* grazie a **Node.js** permette di usare lato server lo stesso linguaggio di programmazione usato lato client;
* permette di effettuare in maniera molto 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.

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

Scelta del sistema operativo

Per quanto riguarda il sistema operativo, la scelta è ricaduta su Linux o Mac Os X, poiché **Quorum-wizard** gira nitidamente su questi sistemi operativi, senza la necessità di installare una macchina virtuale. Tra i due abbiamo preferito optare per una distribuzione Linux, essendo 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’utilizzo della libreria **Mongoose**.

L’utilizzo di un database ci permette di archiviare i dati personali degli utenti richiesti al momento dell’iscrizione al sito, i dati necessari per l’autenticazione, i dati relativi agli eventi disponibili sul sito e le ricevute di pagamento dei biglietti.

Abbiamo utilizzato il database anche per archiviare, temporaneamente, i codici OTP inviati agli utenti tramite mail per eseguire la procedura di autenticazione a due fattori e per mantenere un registro di tutti gli eventi che avvengono nel sito (accessi, iscrizioni, acquisti, …).

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 server che ospita il database i nostri dati sono comunque al sicuro. Di default quando si inizializza un database su MongoDB vengono, infatti, assegnati tre diversi server, uno primario e due di scorta, per memorizzare i dati.

MongoDB fornisce, quindi, un servizio affidabile garantendo la **ridondanza** dei nostri dati. I server però sono situati tutti nello stesso data center, quindi non è garantita la distribuzione. In caso di incidenti sull’intero data center c’è il rischio 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 dei dati su esso archiviati.

Il database è stato configurato in maniera tale che solo l’account utilizzato dal nostro server per accedere alla base di dati abbia 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 nella base di dati.

La libreria **Mongoose Encryption** ci permette di cifrare e autenticare i dati prima di memorizzarli sul database, in modo tale da rendere impossibili letture o alterazioni malevole direttamente dalla gui fornita da MongoDB. La libreria utilizza due chiavi segrete una per la cifratura e una per la firma digitale dei dati.

Scelta del sistema di pagamento esterno

Il pagamento dei biglietti è sicuramente una delle operazioni più delicate che avvengono nella nostra web app. Implementare un sistema di pagamento da zero è un’impresa molto ardua e comporta altissimi rischi. Abbiamo deciso di optare per un sistema di pagamento esterno. Oltre a poter fare affidamento su un sistema già ampiamente collaudato abbiamo il vantaggio di condividere 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.

PayPal, inoltre, mette a disposizione una **sandbox** che ci permette di testare i pagamenti nella 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 stabilire una connessione sicura è necessario garantire mutua autenticazione tra client e server (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 autorilasciato.

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 autorilasciato risulta poco attendibile, quindi nel momento in cui metteremo la nostra web app online dovremo provvedere a farci rilasciare un certificato da un’autorità garante. Ciò comporterà costi aggiuntivi, ma garantisce maggiore affidabilità alla 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 **QR code**. In questo modo, un cliente per accedere ad un evento 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 contenente il QR code.

L’utente può scaricare o 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 o digitale, salvato ad esempio 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 proprio smartphone. Quando questa viene attivata, viene attivato 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 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 invalidare il biglietto. Se la funzione non riesce a rilevare alcun codice, allora non viene eseguita nessuna 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’invalidatore dei biglietti 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 asimmetria **RSA**. Per eseguire questa procedura ci siamo affidati alla libreria **Crypto** integrata in Node.js.

Il calcolo dell’hash garantisce integrità, dimostrando che il biglietto non sia stato compromesso e alterato, mentre la firma digitale garantisce che il biglietto sia stato rilasciato dal server della web app (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. Abbiamo stabilito non fosse necessario memorizzare il sigillo fiscale anche non 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 propri dati personali al server. Questi sono protetti grazie all’utilizzo del protocollo TLS.

Il server una volta ricevuti i dati dell’utenti calcola l’hash della password e la memorizza nel database insieme agli altri dati dell’utente. Per evitare attacchi del dizionario prima di calcolare l’hash viene aggiunto un salt alla password.

Come già accennato i dati presenti sul database sono cifrati e firmati, quindi viene garantito un ulteriore livello di sicurezza.

Nel momento in cui l’utente accede al sito web non si controlla la validità della password, ma se ne calcola l’hash e lo si confronta con quello memorizzato nel database.

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 client riceve una 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 alla risorsa richiesta.

I codici **OTP** appena generati vengono memorizzati nel database, anche questi in maniera cifrata. 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 2). 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 ed un minuto per quella multi-fattore. Eliminando i cookie del browser l’utente perderà il proprio 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 con lo smart contract, tenendo tracce 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 sito web, al termine dell’acquisto dei biglietti o dopo la concessione di nuovi privilegi ad un utente il server invia una mail al diretto interessato per informarlo che l’operazione è avvenuta con successo. L’invio delle mail è gestito dalla libreria **Nodemailer** a partire da una casella di posta **Gmail**. La confidenzialità delle mail 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. Inoltre, 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. Si potrebbe pensare di inserire anche un controllo di integrità del file System per verificare se il codice del server sia stato compromesso.

Potrebbe essere necessario 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**. Il plugin 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. I campi necessari per identificare i record, come gli **id**, devono, quindi 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.

Dati come gli id dei record o le mail degli utenti registrati al sito non sono confidenziali.

Conformità alle linee guida di progettazione sicura

* Minimizzare la superficie di attacco
  + L’applicazione web si appoggia su un database non relazionale. MongoDB utilizza uno strumento sicuro per l’assemblaggio delle query, basato sul formato **Bynary 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
  + L’applicazione web non è configurabile. Di default sono incluse tutte le misure che contribuiscono a rendere l’applicazione sicura.
  + Un utente non autenticato può consultare solamente il catalogo degli eventi senza eseguire nessuna operazione.
  + Un utente non autorizzato quando tenta di accedere ad una rotta per cui non è autorizzato viene reindirizzato alla pagina di login. Solo dopo aver inserito credenziali con opportuni privilegi potrà accedere.
* Principio del privilegio minimo
  + Al momento dell’iscrizione al sito web ad ogni utente vengono assegnati i privilegi da cliente. Le uniche operazioni possibile per un utente con privilegi da cliente sono la consultazione del catalogo e l’acquisto dei biglietti. Un utente per ottenere privilegi superiori dovrà fare richiesta allo staff della biglietteria, che deciderà se concedergli o meno le autorizzazioni richieste.
* Difesa in profondità
  + Autenticazione a più fattori per le operazioni più 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. 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.
* Separazione dei compiti
  + Non esistono utenti privilegiati che possono fare qualunque cosa. I privilegi associati ad un account concedono 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 nel catalogo visualizza solo gli eventi della società cui appartiene.
  + L’utente ospite e il cliente nel catalogo visualizzano solo gli eventi per cui sono in vendita i biglietti.
  + Solo lo staff della biglietteria ha accesso all’intero catalogo degli eventi.
* Open design
  + Evitiamo 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 degli algoritmi è tutta incapsulata all’interno delle chiavi segrete 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 al sito web. In questo modo si iscriveranno solamente nel caso in cui decideranno di acquistare i biglietti per un evento.
  + L’autenticazione multi-fattore, che potrebbe essere considerata una misura troppo stringente dagli utenti, viene richiesta solamente per alcune operazioni.
  + L’interazione con la blockchain per l’emissione e l’invalidazione dei biglietti è gestita interamente dal server in maniera automatica, sollevando gli utenti da questo compito. Gli indirizzi degli wallet vengono memorizzati all’interno del database. Gli utenti devono ricordare semplicemente la password del proprio wallet.
  + 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 scannerizzando 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 registrate le attività dei vari utenti (accessi, iscrizioni, acquisti).
* 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 campi, 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, email, date, immagini …
  + Il server prima di inserire i dati all’interno del database controlla se i campi del record sono compatibili con i tipi di dato definiti nel modello della collezione.
* 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 è pubblica (disponibile per chiunque) o privata (disponibile solo per utenti autenticati) e il una lista di privilegi che l’utente deve possedere per accedere.
* Mediazione completa
  + Il server ad ogni richiesta di accesso, modifica o cancellazione di risorse verifica il token di autenticazione dell’utente e i suoi privilegi.
  + Non viene utilizzato alcun tipo di cache.