Progetto di *Programmazione Web*

NOTE PROGETTUALI – Ultima modifica: 12 aprile 2021

Sommario

[Vue.js 3](#_Toc69132718)

[Plugin utilizzati 3](#_Toc69132719)

[Dipendenze 3](#_Toc69132720)

[Dipendenze di sviluppo 3](#_Toc69132721)

[Test 4](#_Toc69132722)

[Compatibilità 4](#_Toc69132723)

[Vue Router 4](#_Toc69132724)

[Utilizzo di mappe (*Map*) per la gestione dei documenti degli attori sul client 4](#_Toc69132725)

[Utilizzo dei wrapper in *Vue* 4](#_Toc69132726)

[Scambio di informazioni tra i componenti *Vue* 5](#_Toc69132727)

[Token CSRF 5](#_Toc69132728)

[Formato per lo scambio dati con il server 7](#_Toc69132729)

[Elenco attori 8](#_Toc69132730)

[Tabella documenti 8](#_Toc69132731)

[Limiti di prestazioni e miglioramenti possibili sul client 8](#_Toc69132732)

[Bootstrap 9](#_Toc69132733)

[Autenticazione 10](#_Toc69132734)

[Password dimenticata 10](#_Toc69132735)

[Autenticazione con *Firebase* 10](#_Toc69132736)

[Logout 11](#_Toc69132737)

[REST 12](#_Toc69132738)

[JWT 12](#_Toc69132739)

[Sicurezza e Autenticazione 12](#_Toc69132740)

[Meccanismo di autenticazione dei client basato su token e prevenzione dal furto dei token 13](#_Toc69132741)

[CSRF 14](#_Toc69132742)

[Cookie e token utilizzati 14](#_Toc69132743)

[Token di autenticazione 14](#_Toc69132744)

[AuthenticationTokenInvalido 15](#_Toc69132745)

[Cookie 15](#_Toc69132746)

[TOKEN-ID-CLIENT-AUTENTICAZIONE 15](#_Toc69132747)

[CSRF-TOKEN-JWT 15](#_Toc69132748)

[Servizi *stateless* 15](#_Toc69132749)

[Filtri nella *Java WebApplication* 15](#_Toc69132750)

[Gestione dei documenti 16](#_Toc69132751)

[Caricamento di un documento 16](#_Toc69132752)

[Scaricamento di un documento 16](#_Toc69132753)

[Eliminazione di un documento 16](#_Toc69132754)

[E-mail 16](#_Toc69132755)

[Classi Java di supporto 17](#_Toc69132756)

[Web service e client REST 17](#_Toc69132757)

[Client REST 17](#_Toc69132758)

[Creazione, modifica ed eliminazione di un attore nella piattaforma 17](#_Toc69132759)

[Creazione di un Attore 17](#_Toc69132760)

[Registrazione di un attore nella piattaforma 18](#_Toc69132761)

[Modifica di un attore 18](#_Toc69132762)

[Eliminazione di un attore 19](#_Toc69132763)

[Eliminazione di un Consumer 19](#_Toc69132764)

[Eliminazione di un Uploader 19](#_Toc69132765)

[Eliminazione di un Administrator 19](#_Toc69132766)

[Serializzazione JSON 19](#_Toc69132767)

[Logo degli Uploader 19](#_Toc69132768)

[Utilizzo dei metodi HTTP *POST, PUT* e *PATCH* 21](#_Toc69132769)

[Form di login 21](#_Toc69132770)

[File *index.yaml* per gli indici del Datastore 22](#_Toc69132771)

[Deploying dell’applicazione 22](#_Toc69132772)

[Note sul funzionamento della piattaforma 22](#_Toc69132773)

[Primo avvio della piattaforma 22](#_Toc69132774)

[HTTPS 22](#_Toc69132775)

**Servlet container utilizzato: *Google App Engine***

Sono stati implementati tutti i requisiti di progetto.

# Vue.js

Si è utilizzato *Vue 3* per creare una *Single Page Application* che interagisce per mezzo di richieste asincrone con il server e si occupa della rappresentazione dei dati da mostrare agli utenti sul browser.

Durante l’analisi delle specifiche del progetto si è individuata una soluzione consistente nello sviluppo di componenti separati e specializzati, con l’obiettivo di riutilizzare il codice evitandone la duplicazione. Si è quindi scelto di utilizzare *Vue.js* per lo sviluppo client-side, poiché è leggero (in termini di dimensioni) ed efficiente e permette di semplificare la creazione dell’applicazione, con un approccio basato, appunto, su componenti.

Si è utilizzato *npm* come gestore dei pacchetti.

## Plugin utilizzati

* @vue/cli-service Fornisce i comandi per Vue (serve/build/inspect)
* @vue/cli-plugin-babel Polyfill features per JavaScript ed in generale per convertire codice ECMAScript 2015+ in modo che sia retrocompatibile con precedenti versioni di JavaScript.[[1]](#footnote-1)
* @vue/cli-pllugin-eslint validatore EcmaScript
* @vue/cli-plugin-router Fornisce gli strumenti per creare una single page application con Vue.

## Dipendenze

* core-js Modular standard library for JavaScript. Includes polyfills for ECMAScript up to 2021: promises, symbols, collections, iterators, typed arrays, many other features, ECMAScript proposals, some cross-platform WHATWG / W3C features and proposals like URL. You can load only required features or use it without global namespace pollution. [[2]](#footnote-2)
* vue è il framework Vue stesso
* vue-router Single-page application routing
* axios Promise based HTTP client for the browser and node.js

## Dipendenze di sviluppo

* @vue/compiler-sfc Compila i file *Vue Single File Components* in file JavaScript
* @jest/globals Utilizzata per test
* babel-eslint Validatore per il codice Babel utilizzando ESLint
* eslint Validatore per il codice JavaScript
* eslint-plugin-vue Official ESLint plugin for Vue.js. This plugin allows us to check the <template> and <script> of .vue files with ESLint, as well as Vue code in .js files.
  + Finds syntax errors.
  + Finds the wrong use of Vue.js Directives (opens new window).
  + Finds the violation for Vue.js Style Guide (opens new window).

ESLint editor integrations are useful to check your code in real-time. [[3]](#footnote-3)

## Test

Per la creazione dei test sono state utilizzate la libreria **vue-test-utils[[4]](#footnote-4)** ed il framework **Jest[[5]](#footnote-5)**. Si è fatto riferimento al link [[6]](#footnote-6) per l’installazione, in particolare è stato usato il comando **vue add unit-jest** per aggiungere *Jest* al progetto. Il file ***jest.config.js*** permette di configurare *Jest* (ad esempio, qui vanno specificati i test da eseguire).

# Compatibilità

L’applicazione è pensata per essere compatibile con il maggior numero possibile di browser.

*Vue 3* non è compatibile con *Internet Explorer*.

# Vue Router

Il file *index.js* contenente le *route* utilizzate in *Vue-Router* risulta particolarmente complesso, ma ciò è stato fatto per permettere agli utente l’accesso diretto ad un particolare contenuto dell’applicazione (ad esempio alla scheda relativa ad un attore) senza dover navigare nell’applicazione per arrivarci.

# Utilizzo di mappe (*Map*) per la gestione dei documenti degli attori sul client

Nello sviluppo *front-end* è stato spesso utilizzato l’oggetto *Map* (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map>) utilizzando (nella rappresentazione di una lista di documenti) come chiave di ogni entry l’identificativo univoco di un documento e come corrispettivo valore l’oggetto in cui ogni *property* è una proprietà del documento; in modo analogo per rappresentare una lista di *Attori* si è utilizzato un oggetto *Map* in cui ogni *entry* ha per chiave l’identificativo univoco di un attore e come corrispettivo valore l’oggetto con tutte le proprietà *dell’Attore*.

# Utilizzo dei wrapper in *Vue*

Nell’applicazione *Vue* spesso si è ricorso all’utilizzo di *wrapper*: quando il componente padre scambia un’informazione (una variabile) al figlio c’è unidirezionalità: se poi il padre cambia la variabile, nel figlio permane il vecchio valore (a meno che non si aggiorni la pagina con l’obiettivo di ricostruire tutti i componenti). Per ovviare a questo valore, tutte le proprietà (*prop*) ereditate nei componenti dal componente padre sono associate ad un *watch*, il quale permette al componente figlio di accorgersi di eventuali modifiche nella variabile nel componente padre; tuttavia non è possibile propagare in modo diretto tali modifiche nella *prop* del figlio perché è di sola lettura, quindi si è creata una variabile *wrapper* nel figlio che inizialmente è una copia della *prop* ereditata dal padre, poi, quando il *watch* rileva delle modifiche, viene aggiornata col nuovo valore rilevato dal *watch*.

## Scambio di informazioni tra i componenti *Vue*

Nell’applicazione spesso è stato necessario permettere lo scambio di informazioni tra componenti diversi. Per fare ciò sono stati usati:

* + 1. l’attributo *props* dei componenti *Vue*: nel componente figlio si è aggiunta a questo attributo la proprietà contenente l’informazione proveniente dal padre;
    2. un *watch* (Fonte: <https://v3.vuejs.org/guide/reactivity-computed-watchers.html#watch>) nel componente figlio per controllare se una certa proprietà (dichiarata in *props* nello stesso componente figlio) è stata modificata dal componente padre;
    3. un *wrapper* (definito in *data()* nel componente figlio) per ciascuna proprietà soggetta a modifiche da parte del padre: quando una *prop* viene modificata, tale modifica viene osservata dal *watch* del componente figlio che provvede ad aggiornare la variabile corrispondente (gli elementi definiti in *props* non possono essere sovrascritti, quindi è stato necessario definire un elemento *wrapper* in *data()* ed è quest’ultimo elemento che viene sovrascritto dal *watch*);
    4. la direttiva *v-bind* di *Vue* per permettere al componente padre di inviare dei dati al figlio;
    5. gli eventi per permettere al componente figlio di trasmettere un’informazione al componente padre.

### Token CSRF

Il token CSRF verificato dal server deve coincidere con quello presente nell’apposito *cookie*, ciò significa che quando un *form* richiede un nuovo token CSRF, il componente che contiene tale *form* deve informare (tramite la generazione di un apposito evento) il componente padre, che provvederà a sua volta a propagare l’informazione (generando un ulteriore evento) fino al componente radice. A questo punto, il componente radice deve informare tutti i componenti figli di questa modifica (potrebbe essere presente un token CSRF anche in un altro componente): per fare ciò, i componenti figli ricevono tramite *props* il nuovo token CSRF ed i figli dispongono di un *watch* che ne osserva le eventuali modifiche. A questo punto, i figli aggiornano il valore del token definito nella variabile *wrapper* (in *data()*).

Per aumentare il livello di sicurezza del sistema, ogni *form* ha un token CSRF diverso, anche se in realtà potrebbe essere sufficiente uno per tutta la sessione (infatti, quando viene richiesto un nuovo token CSRF, la modifica del vecchio token CSRF viene propagata all’intera applicazione); in ogni caso ciò garantisce un ricambio frequente del token.



# Formato per lo scambio dati con il server

Spesso il client richiede al server dei dati, come ad esempio l’elenco di file destinati ad un Consumer: ognuno di questi file è costituito da molte proprietà, ma il client non ne conosce la struttura (altrimenti si avrebbe duplicazione di codice tra client e server). In questo progetto, quando il client richiede dati al server i quali contengono delle proprietà, esso li invia in forma di mappa, avente per chiave un identificativo e per valore un oggetto con tutte le proprietà di interesse, ad esempio:

**[Client 🡪 Server]** *GET request all’url* /getElencoDocumenti

**[Server 🡪 Client]** *response:*

{

abc5B :

{

nomeDocumento : "fattura idraulico",

dataCaricamento : "11 gennaio 2021",

destinatario : "Luigi Bianchi"

},

1234s :

{

nomeDocumento : "preventivo per Mario",

dataCaricamento : "20 dicembre 2020",

destinatario : "Luigi Bianchi"

}

}

Il client non conosce la struttura dei dati che gli arrivano, quindi a priori non sa come interpretarli, ma può chiederlo al server, ad esempio:

**[Client 🡪 Server]** *GET request:* /getNomePropDestinatarioDocumenti

**[Server 🡪 Client]** *response:*

destinatario

Ricevuta la riposta dal server, il client sa che se vuole elencare i destinatari di tutti i documenti che ha ricevuto, dovrà cercare tra i valori della mappa ricevuta precedentemente la proprietà destinatario ed estrarne il valore.

In questo modo:

* La struttura degli oggetti inviati dal server al client è decisa solamente dal server.
* Non c’è duplicazione di informazioni (e nemmeno di codice) tra client e server (il client non conosce a priori la struttura dei dati, solo che si tratta di una mappa chiave valore).
* Il client possiede già tutte le informazioni sugli oggetti da mostrare e chiede al server solamente come interpretarli, quindi non c’è overhead di dati scambiati tra client e server (il dato completo – cioè la mappa chiave-valore – viene inviato solo una volta).
* Nella mappa data al client, la chiave può essere qualsiasi, l’importante è che sia univoca nella mappa (una mappa non può, per implementazione in JavaScript, avere due *entry* con la stessa chiave).

# Elenco attori

Gli attori vengono mostrati in ordine alfabetico in base al nominativo. L’elenco degli attori si auto-aggiorna periodicamente, secondo un intervallo temporale di durata predefinita e parametrizzata nelle variabili d’ambiente del progetto *Vue*. Se non vengono notate modifiche nell’istante di aggiornamento (si confrontano gli identificativi ricevuti dal server con quelli già noti al client), allora nulla viene modificato. Potrebbe accadere che pur rimanendo gli stessi attori nell’elenco, uno di questi modifichi una sua *property*: tale modifica non sarà propagata sul client, che non effettuerà alcuna modifica a meno che non noti una discrepanza tra l’array degli identificativi degli attori nell’elenco ad esso noti e quelli ricevuti dal server: per osservare tale modifica sarà necessario ricaricare il componente.

# Tabella documenti

Come Elenco attori, anche Tabella documenti si auto-aggiorna periodicamente. Se nell’istante dell’aggiornamento il numero di documenti noti al client coincide con il numero di documenti noti al server, allora il server risponde con NOT\_*MODIFIED*. Potrebbe succedere che tra due istanti successivi un documento modifichi la sua anteprima -cioè le sue proprietà come nome e data di caricamento- (tale modifica non verrà mostrata all’utente a meno che non ricarichi la pagina).

L’elenco dei documenti viene ordinato dal database in modo ascendente sulla data di caricamento (il più recente davanti); il client è stato programmato sia per ordinare in base alla data di caricamento (il fatto di fornire una lista già ordinata riduce il carico di lavoro per il client) sia a spostare in fondo i documenti già letti, mantenendo l’ordinamento.

# Limiti di prestazioni e miglioramenti possibili sul client

Quando il numero di documenti (o il numero di attori) approccia le migliaia di occorrenze, i metodi di gestione richiedono una quantità esagerata di risorse, al punto che l’applicazione nel client può bloccarsi o comunque manifestare un comportamento inatteso.

Possibile soluzione: cercare di ottimizzare i metodi utilizzati nei componenti e “spezzare” la tabella in più pagine, ognuna delle quali richieda una sottoinsieme dei dati totali, al fine di rendere la gestione più semplice (dal punto di vista della complessità computazione dovuta alla dimensione dell’input) e fornire una migliore esperienza di navigazione all’utente che potrebbe scegliere un sottoinsieme di dati di interessi scegliendo la corretta pagina anziché scorrere una tabella con migliaia di righe.

In ogni caso, tali gravi malfunzionamenti si sono manifestati in condizioni molto sfavorevoli (cioè quando la lista dei documenti (vista da un Consumer e riferita ad un Uploader) si stava auto-aggiornando mentre il client dello stesso Uploader attualmente osservato dal Consumer stava caricando proprio per quel Consumer ed in modo *continuativo* *migliaia* di documenti): caricando la pagina contenente la lista, in condizioni stazionarie – cioè senza che vengano aggiunti nuovi documenti -, anche con migliaia di occorrenze già presenti e con l’utente che interagente con la lista (ad esempio utilizzando i filtri nella lista dei documenti) non sono stati notati malfunzionamenti, ad eccezione di un leggero ritardo (qualche secondo) nel caricamento dell’intera lista e ciò è ragionevole considerando la mole di dati che deve essere comunque elaborata dal client.

# Bootstrap

L’intenzione iniziale era di utilizzare *Bootstrap-Vue* per la parte di stile, purtroppo però, al momento in cui il progetto è stato realizzato, *Bootstrap-Vue* non supportava *Vue3*, perciò è stato incluso come si sarebbe fatto in una *web application* “non – Vue”, dall’elemento *head* del file html contenente l’applicazione.

# Autenticazione

L’autenticazione è di tipo programmatico e non si affida a servizi esterni. Il sistema fa uso di un database di autenticazione di tipo *hashed and salted* in cui viene memorizzato lo username degli utenti, il *salt* e l’*hash* della password (la password viene concatenata con il *salt* -generato contestualmente alla richiesta di creazione di un nuovo account-, prima del calcolo dell’*hash*); in questo modo le password non vengono salvate in chiaro. Nel database di autenticazione è previsto un ulteriore campo di tipo password, anch’essa salvata in modalità *hashed&salted* ed utilizzata solamente in modo temporaneo per il recupero della password (come spiegato nel paragrafo “Password dimenticata”).

Vedere il paragrafo “Sicurezza e Autenticazione” a pagina 12.

Prima di poter utilizzare un account ai fini della procedura di autenticazione, è necessario verificarlo. Vedere il paragrafo “Verifica dell’account” a pagina 18.

## Password dimenticata

Il form di login prevede un link per il reset della password (nel caso in cui questa venisse dimenticata) dell’account il cui username deve essere inserito nel soprascritto form. Lo username viene inviato tramite metodo POST. Quando l’utente clicca su tale link, allora viene generata una nuova password casuale temporanea che viene inviata all’utente via mail. La nuova password temporanea viene salvata nel database di autenticazione senza eliminare la vecchia password.

Dopo aver cliccato sul link di reset password:

* + se l’utente effettua il login con la vecchia password, allora quella temporanea viene eliminata e non sarà più valida;
  + se l’utente effettua il login con la password temporanea, allora quella temporanea diventa a tutti gli effetti l’attuale password dell’utente e la vecchia password viene eliminata.

La password temporanea viene salvata in modalità *hashed&salted* (senza generare un nuovo *salt*) e viene inviata in chiaro via mail all’utente, per questo motivo è fortemente consigliato di modificarla al primo accesso.

## Autenticazione con *Firebase*

*Firebase* è stato utilizzato per permettere agli utenti di usare il loro account *Google* per autenticarsi nell’applicazione, nel seguente modo:

* + 1. l’utente accede all’applicazione
    2. l’utente clicca sull’opzione di autenticazione tramite *Google*
    3. il client esegue il codice JavaScript che permette, tramite *Firebase*, di ottenere le informazioni su quell’utente tramite il suo account *Google* (ipotizzando che l’utente si autentichi correttamente con *Google*)
    4. *Firebase* restituisce un token *JWT* firmato contente le informazioni dell’utente
    5. il client inoltra tale token al server dell’applicazione
    6. il server dell’applicazione recupera l’e-mail dell’utente dal token ricevuto (firmato da *Firebase*)e cerca un attore registrato nel sistema avente tale e-mail (non si interroga l’*Authentication Database* perché ci si fida del token rilasciato da *Firebase* nel quale viene indicata l’email associata all’utente che si è appena autenticato sul client tramite *Firebase*)
    7. se nel sistema esiste un attore con quell’e-mail, allora il server considera autenticato quel client e rilascia token e cookie di autenticazione (come se il client avesse presentato username e password corretti, vedere il paragrafo “Autenticazione” a pagina 10 e “Sicurezza e Autenticazione” a pagina 12).

**Attenzione**: Il login tramite provider esterni (sfruttando *Firebase*) è possibile solamente se l’e-mail usata dal provider esterno è la stessa registrata nel sistema.

## Logout

L’utilizzo di token la cui scadenza è decisa al momento della creazione del token ha reso difficile la progettazione del meccanismo di *logout* degli utenti, realizzato basandosi su quanto descritto nel paragrafo *Sicurezza e Autenticazione*: quando l’utente effettua il logout dal browser, il server sovrascrive il cookie contente il token senza il quale il token JWT di autenticazione diventa inutilizzabile. Inoltre, il client, al logout, cancella (dovrebbe farlo, a meno che non sia stato alterato in modo illecito) il token dalla memoria a disposizione del browser. È stato inoltre implementato un database di token invalidi (la cui messa in funzione è opzionale), Vedere “AuthenticationTokenInvalido” a pagina 15.

In questo modo:

1. l’applicazione può continuare a funzionare anche senza *Firebase* (ad es. a causa di un guasto in *Firebase*)
2. l’applicazione supporta anche il login utilizzando dei *provider* esterni grazie a *Firebase*: ad esempio, un utente può eseguire il login all’applicazione usando il suo account *Gmail*, a condizione che l’indirizzo e-mail sia lo stesso usato per registrarsi nell’applicazione.

# REST

L’applicazione client comunica con il server per mezzo di servizi REST. Nell’implementazione dell’applicativo server si è utilizzata la libreria JAX-RS (implementazione Jersey).

## JWT

I servizi REST fanno uso di asserzioni sotto forma di token JWT per verificare (se necessario) l’autenticazione e l’autorizzazione del client che effettua una certa richiesta. Ciò fornisce un meccanismo sicuro (visto che i *JWT* sono firmati dal server che li emette). Tra i benefici dell’uso di *JWT* si osserva che da esso si possono ottenere numerose informazioni sul client (e sull’attore) a cui è stato rilasciato evitando l’accesso (azione costosa) al database.

In questo progetto si è scelto di realizzare un’applicazione in cui il server non fa uso di sessioni e la comunicazione avviene per mezzo dei token JWT. Pensando ad un’applicazione del mondo reale, affine allo scambio di documenti (cioè simile a quanto richiesto dai requisiti di questo progetto), l’utilizzo di token anziché di sessioni potrebbe semplificare l’integrazione di ulteriori servizi esterni, anche gestiti da server indipendenti, che infatti potrebbero operare basandosi esclusivamente sulle informazioni contenute nei token *JWT*.

Il client utilizza le informazioni contenute nel token JWT (ad esempio il nome dell’utente) per mostrare all’utente i contenuti opportuni, senza però verificare la firma apposta sul token, ciò significa che non viene verificata l’autenticità delle informazioni, quindi, se le informazioni contenute nel token venissero modificate in modo illecito, allora il client caricherebbe l’interfaccia basandosi su tali informazioni alterate, ma in ogni caso non potrà eseguire delle operazioni che non è autorizzato a compiere, perché il server verifica sempre la validità del token JWT presentato da un client prima di svolgere un’operazione per la quale è richiesta un’autorizzazione.

Esiste un database con i token JWT inutilizzabili tale che il server, quando gli si presenta un token, controlla la validità (ed in questo controllo deve essere verificato che il token non sia tra quelli inutilizzabili nel soprascritto database). È inoltre predisposto un task periodico che periodicamente (periodo configurabile) elimina tutti i token scaduti (quindi invalidi per definizione) dal soprascritto database, per liberare spazio dal database. Poiché l’accesso al database è costoso e poiché quest’ultima funzionalità appena descritta richiederebbe l’accesso ogni volta che il server riceve un token (quasi ad ogni richiesta), allora tale funzionalità può essere disabilitata, modificando un flag della classe *Java* che la implementa (*AuthenticationTokenInvalido.java*) [Vedere il paragrafo “AuthenticationTokenInvalido” a pagina 15].

## Sicurezza e Autenticazione

Sono state adottate le seguenti misure precauzionali:

* + Tutti i form comprendono un campo generato casualmente per evitare attacchi CSRF (nel progetto ci si riferisce a tale valore casuale con il nome di *token CSRF*).
  + I cookie hanno gli attributi “*SameSite*” (di tipo *Lax*, per prevenire gli utilizzi di tipo *Cross-Site*, spesso sfruttati negli attacchi informatici[[7]](#footnote-7)), *HttpOnly* (per prevenirne l’accesso da parte di *client-side script*) e *Secure* (per essere inviati solo con il protocollo HTTPS, in ambiente di produzione)impostati.
  + È stata implementata una *Cross Origin Policy* che limita le richieste esaudite alle sole provenienti dalla medesima origine.
  + I token di autenticazione hanno breve durata
  + I form di autenticazione non hanno la funzione di auto-completamento.
  + I client si autenticano attraverso form e, se la procedura di login ha esito positivo, ricevono dal server un token *JWT* ed un cookie che permette di verificare che il token di autenticazione venga utilizzato solo dal client a cui è stato rilasciato.

### Meccanismo di autenticazione dei client basato su token e prevenzione dal furto dei token

La soluzione adottata, intesa come compromesso ragionevole, in questo progetto è stata: salvare il token JWT di autenticazione nella memoria a disposizione del browser (in *localStorage*) ed impostare allo stesso tempo un cookie, del tipo non accessibile dal livello applicativo (*HttpOnly*), con valore un token (valore alfanumerico casuale) il cui hash -calcolato usando come chiave la password (*hashed* e *salted*) dell’utente- è presente in un claim nel payload del *JWT* salvato nella memoria del browser: in questo modo il client si autenticherà presso il server presentando il token JWT nell’header *Authorization* (*Bearer Authentication*) ma allo stesso tempo gli invierà il cookie contenente il token. In questo modo:

* + il server potrà fidarsi circa la veridicità del token JWT di autenticazione (in quanto è firmato dal server stesso)
  + il client è tutelato dal possibile furto del token che sarebbe rifiutato dal server se non accompagnato dal cookie col valore corretto (non rubabile in quanto non accessibile dal livello applicativo e non ottenibile dal token JWT in cui presente solo il suo hash calcolato usando come chiave l’hash della password del client).
  + Se il client cambia la password, tutti i token precedentemente rilasciatigli e non ancora scaduti diventano automaticamente inutilizzabili.
  + La chiave segreta del server è meno esposta a possibili tentativi di decrittazione (se si usasse l’hash della chiave del server come chiave per il calcolo dell’hash contenuto nel cookie, allora si fornirebbero delle informazioni circa crittografiche sulla chiave segreta del server).
  + L’applicativo client è a conoscenza di tutte le informazioni di autenticazione dell’utente necessarie per predisporre opportunamente l’interfaccia grafica.

I token hanno una scadenza temporale, dopo la quale l’utente dovrà ri-autenticarsi: ciò può comportare un fastidio nell’utente, tuttavia può essere considerato un ulteriore meccanismo di sicurezza per evitare che un utente permanga autenticato per uno tempo indefinito.

**Attenzione**: in ambiente di sviluppo *Google App Engine*, è stato programmato che all’avvio del server alcuni attori vengano creati automaticamente. Nell’*Authentication Database*, il *salt* di ogni entry viene generato casualmente quando si crea l’entry. Se si riavvia il server di sviluppo, in generale, tutti i token rilasciati prima del riavvio diventano invalidi, infatti possono succedere le seguenti cose:

* + se *non* si esegue il comando *mvn clean*, le istanze della base di dati permarranno e al riavvio del server, quelle create automaticamente, verranno ri-create: ciò comporterà dei duplicati che daranno problemi durante l’autenticazione;
  + se si esegue il comando *mvn clean*, le istanze della base di dati verranno eliminate ed al riavvio del server il *salt* delle entità nell’*Authentication* *Database* sarà diverso da quello che si aveva prima del riavvio del server, quindi cambierà l’*hash*, perciò, dopo il riavvio del server di sviluppo vi saranno problemi di autenticazione (i token di autenticazione precedentemente rilasciati diventano invalidi).

La soluzione, a fini di test, per verificare il comportamento del sistema in ambiente di sviluppo (che funzioni come da aspettative), è riavviare il server di sviluppo, utilizzando anche il comando *mvn clean*, quindi durante il *bootstrap* del server modificare manualmente (tramite un *breakpoint*, ad esempio) il *salt* delle istanze che si stanno utilizzando con il valore che si aveva quando tali istanza sono state create (prima del riavvio del server): in questo modo le istanze in uso non avranno problemi di autenticazione.

Si noti che questi problemi insorgono solamente in ambiente di sviluppo, in quanto altrimenti si utilizza un “vero” database, perciò il *salt* di un’istanza non viene modificato al riavvio del server.

## CSRF

La web-application implementa un meccanismo di prevenzione delle vulnerabilità CSRF: ogni form utilizzabile per modificare lo stato dell’applicazione include un campo di *input* di tipo *hidden* contenente un *token CSRF*, il cui valore viene anche salvato nel *payload* di un token *JWT* (emesso e firmato dallo stesso server che ha generato il token *CSRF*) memorizzato in un *cookie* di tipo *HttpOnly*, creato al momento della generazione del token *CSRF*.

L’utilizzo del token CSRF nel *form* di login è motivato da quanto descritto in <https://seclab.stanford.edu/websec/csrf/csrf.pdf> (paragrafo 3) e <https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#login-csrf> .

Anche il form di logout è protetto dagli attacchi CSRF (“*You should also protect your logout mechanism against CSRF. At first it seems that all an attacker can do is logout the user, which would be annoying at worst. However, if you combine this with a phishing attack, the attacker may be able to entice the victim to re-login in using their own form and then capture the credentials. See here for a recent example -* [*LostPass*](http://www.tomsguide.com/us/lastpass-phishing-attacks,news-22139.html)*.[* [*http://www.tomsguide.com/us/lastpass-phishing-attacks,news-22139.html*](http://www.tomsguide.com/us/lastpass-phishing-attacks,news-22139.html)*]*”, Fonte: <https://security.stackexchange.com/a/62797> ).

# Cookie e token utilizzati

Questo progetto fa uso di *cookie* per la prevenzione dagli attacchi CSRF (vedere il paragrafo *CSRF*) e per l’autenticazione (vedere il paragrafo *Sicurezza e Autenticazione*) del client.

## Token di autenticazione

Per permettere al server di verificare l’autenticazione di un client, quest’ultimo allegherà in tutte le richieste HTTP, come valore dello *header Authorization*, un token *JWT* codificato in Base64-url encoded. Tale token JWT contiene almeno i seguenti claim:

* + **sub** contiene l’identificativo dell’attore associato al token
  + **exp** contiene la data di scadenza del token, rappresentata come numero di secondi trascorsi rispetto ad *Unix epoch* (mezzanotte del primo gennaio 1970).
  + **hash-TOKEN-ID-CLIENT-AUTENTICAZIONE** contiene il valore di hash (calcolato usando come chiave l’hash della password dell’attore autenticato) di un token casuale associato al client e memorizzato nel cookie TOKEN-ID-CLIENT-AUTENTICAZIONE e permette al server di effettuare un “controllo incrociato” sull’autenticazione del client (poiché il cookie è di tipo *HttpOnly*, esso non dovrebbe poter essere accessibile dagli script del livello applicativo quindi si assume – se tale controllo incrociato va a buon fine – che il client che attualmente sta presentando il token di autenticazione sia lo stesso a cui era stato rilasciato).
  + **Subject name** contiene il nominativo dell’attore a cui il token è stato rilasciato.
  + **Tipo attore** contiene il tipo di attore autenticato (*Administrator/Uploader/Consumer*)

## AuthenticationTokenInvalido

Come ulteriore misura di sicurezza, in questo progetto si è utilizzato un database per memorizzare i token invalidi. Un’entità di tale database è gestita dalla classe *AuthenticationTokenInvalido*. Per ogni richiesta proveniente dal client contenente un token si verifica che tale token non sia presente nel database dei token invalidi. Quando un token deve essere invalidato (ad esempio quando un utente richiede il logout), esso viene aggiunto al database dei token invalidi. Periodicamente viene eseguito un *task* che si occupa di esaminare il database dei token eliminati per rimuovere da tale database i token scaduti che comunque verrebbero rifiutati se presentati da qualche client. L’utilizzo di questa classe è configurabile, in quanto è molto oneroso in termini di accessi al database (almeno un accesso in lettura per ogni richiesta).

## Cookie

### TOKEN-ID-CLIENT-AUTENTICAZIONE

Questo cookie viene rilasciato dal server a seguito della corretta procedura di login ed è strettamente collegato al token di autenticazione JWT: il token JWT contiene un claim il cui valore è il risultato di una funzione di hash applicata al valore di questo cookie usando come chiave il valore dell’hash della password del client (Nota: né la password dell’attore né il suo valore di *hash* vengono mostrati a seguito di questa procedura).

### CSRF-TOKEN-JWT

Questo cookie contiene come valore un token JWT, codificato in Base64-UrlEncoded, con i seguenti claim:

* + **exp** contiene la data di scadenza del cookie, espressa come numero di secondi trascorsi da *Unix-Epoch*.
  + **CSRF-TOKEN** contiene il valore del token CSRF comunicato al client (il server non mantiene una sessione dei token CSRF, quindi è necessario salvare tale token da qualche parte e si è deciso di salvarlo in questo cookie - di tipo *HttpOnly*).
  + **IP-CLIENT** contiene l’indirizzo IP del client a cui è stato rilasciato il token CSRF: si è considerato che tale indirizzo potrebbe essere quello di un NAT o comunque potrebbe modificarsi, in ogni caso il token CSRF serve solamente per il tempo strettamente necessario alla compilazione di un *form*, quindi si è assunto che l’indirizzo IP del client non venga modificato da quando questo *cookie* viene generato a quando verrà poi ripresentato al server; si è considerato che a causa di tecnologie (quali NAT o l’utilizzo di un *proxy*), lo stesso indirizzo IP potrebbe far riferimento ad un certo numero di client in realtà distinti, ma si è pensato che includerlo nei controlli comunque aumentasse il livello di sicurezza dell’applicazione.

# Servizi *stateless*

L’intera piattaforma è stata implementata basandosi sull’approccio *stateless*: ogni interazione tra client e server è indipendente dalle altre ed il server non tiene traccia di una sessione. I cookie sono stati usati per implementare l’algoritmo di sicurezza (e non per tenere traccia di un’ipotetica sessione) ed i token *JWT* per memorizzare delle informazioni sull’utente (indipendenti dalla specifica richiesta).

# Filtri nella *Java WebApplication*

I filtri dell’applicazione web sono stati realizzati sfruttando l’*annotation* *@WebFilter*. Il *filter-mapping* dei filtri è stato indicato nel file *web.xml*, così da poter definire il loro ordine di esecuzione, che risulta fondamentale, in quanto da essi dipendono i controlli di sicurezza (alterare l’ordine di esecuzione dei filtri può causare l’impossibilità agli utenti di accedere al sistema; in ogni caso, gli utenti non autenticati *non* possono accedere a risorse protette, nemmeno modificando l’ordine dei filtri).

# Gestione dei documenti

## Caricamento di un documento

Il caricamento di un file viene gestito tramite il metodo *POST* [[8]](#footnote-8). Il file può essere caricato dall’attore tramite un campo *input* di tipo *file* posto in un *form*. I dati del form sono stati raccolti in una variabile di tipo *FormData*, quindi inviati al server tramite una richiesta asincrona, avendo avuto cura di aggiunge lo *header* *“Content-Type: multipart/form-data*”. Nel server è stato programmato un servizio di tipo REST che attende i dati del *form* sfruttando l’*annotation @FormDataParam*. In particolare, per il file sono stati utilizzati i tipi *InputStream* (per il contenuto, in seguito convertito in *byte[]* e memorizzato come *blob* nel database) e *FormDataContentDisposition* (per i dettagli del file). Fonte: <https://stackoverflow.com/a/25889454>. L’*annotation* *@FormDataParam* ha richiesto la *dependency* *org.glassfish.jersey.media:jersey-media-multipart* (aggiunta nel file *pom.xml* del progetto) ed è stato modificato il file *web.xml* (come descritto in <https://stackoverflow.com/a/25312655>).

## Scaricamento di un documento

Il file da scaricare viene richiesto al server (tramite metodo HTTP *GET*) e (se l’utente dispone delle autorizzazioni opportune), il server lo fornisce come *entity* di una risposta di tipo *application/octet-stream*. Il client si occupa sia della richiesta (asincrona) del documento sia del download (si è preso esempio da <https://stackoverflow.com/q/33247716>).

## Eliminazione di un documento

Gestita tramite il metodo *DELETE* del protocollo HTTP, can parametro l’identificativo del file da eliminare: quando la richiesta giunge al server, quest’ultimo verifica che il client di provenienza della richiesta disponga delle autorizzazioni necessarie, quindi (se il documento esiste) procede alla sua eliminazione dal database.

# E-mail

La gestione delle email sul server sfrutta la *Mail API* di *Appengine*, basandosi su <https://cloud.google.com/appengine/docs/standard/java/mail/sending-mail-with-mail-api> e <https://github.com/GoogleCloudPlatform/java-docs-samples/blob/master/appengine-java8/mail/> . Il numero di email inviate viene conteggiato da Google ai fini della fatturazione come descritto in <https://cloud.google.com/appengine/quotas#Mail> .



# Classi Java di supporto

Il progetto prevede molte classi di supporto, salvate nel *package \*.utils*. Tra esse, in particolare vi sono:

* una per la gestione dei token JWT, cosicché i *claim* presenti in ogni *token JWT* siano completamente personalizzabili, adattabili ad esigenze particolari o integrabili con altri sistemi;
* una per la gestione dei *cookie*, cosicché le opzioni per i *cookie* siano personalizzabili (ad esempio si possono modificare le opzioni *HttpOnly, Secure, MaxAge, SameSite*);
* una per la gestione della sicurezza: il sistema potrebbe essere esteso per supportare la cifratura a chiave pubblica, così da permettere anche ad altri server di validare un *token JWT* firmato da questo server ed integrare il sistema con servizi esterni [[9]](#footnote-9).

# Web service e client REST

I requisiti di progetto richiedono la realizzazione di un Web Service e di un client che ne dimostri il funzionamento.

In questo progetto si è deciso di seguire l’approccio REST utilizzando la libreria *JAX-RS*, perché:

* + SOAP utilizza XML e ciò comporta maggiore *overhead*;
  + La libreria *JAX-WS* dedicata ai web service SOAP in Java non è supportata da *Google AppEngine*;
  + *Jersey*, cioè l’implementazione di riferimento dell’interfaccia *JAX-RS* per l’implementazione di servizi REST in Java, utilizzata in questo progetto, è ben documentata.

## Client REST

Per la creazione del client REST è stato creato un nuovo progetto *Maven*, in cui è stata inserita la seguente *dependency*:

<dependency>

<groupId>org.glassfish.jersey.core</groupId>

<artifactId>jersey-client</artifactId>

<version>2.33</version>

</dependency>

Riferimenti seguiti:

* + <https://docs.oracle.com/javaee/7/api/javax/ws/rs/client/package-summary.html>
  + <https://docs.oracle.com/javaee/7/tutorial/jaxrs-client001.htm>

Ulteriori informazioni sul client REST (in particolare su come eseguirlo) sono disponibili nel file README del progetto stesso. È possibile distribuire il file *JAR* relativo al client REST. Si veda il file README del progetto.

# Creazione, modifica ed eliminazione di un attore nella piattaforma

## Creazione di un Attore

Un *Uploader* può creare (“aggiungere”) un *Consumer*: se lo username inserito non è già presente nel sistema, allora il *Consumer* viene creato e la password viene generata in modo casuale. Al termine dell’operazione di creazione, il *Consumer* viene notificato con un email all’indirizzo specificato. Se il *Consumer* inserito dall’*Uploader* è già registrato nel sistema, allora viene direttamente associato all’*Uploader*, altrimenti prima viene creato e poi aggiunto; nel caso in cui lo *username* del *Consumer* inserito dall’*Uploader* coincida con lo username di un *Consumer* registrato nel sistema, ma gli altri campi differiscano da quelli memorizzati, allora l’operazione si interrompe e tale anomalia viene segnalata all’*Uploader* indicando i dati del *Consumer* già registrato nel sistema.

Analogamente, un *Administrator* può creare un *Uploader* o un altro *Administrator*.

In questi casi, la password per il nuovo attore viene generata in modo casuale dal server e viene inviata in chiaro nell’e-mail per la verifica dell’account (senza mostrarla all’*Uploader* o all’*Administrator* che ha richiesto la creazione dell’attore), motivo per cui l’utente associato all’account appena creato è invitato alla modifica della password al primo accesso.

### Registrazione di un attore nella piattaforma

Un attore può essere registrato nella piattaforma in due modi:

1. in modo spontaneo, nel ruolo di *Consumer*, tramite l’apposito form;
2. da un *Uploader* o da un *Administrator*, come descritto negli appositi paragrafi.

#### Verifica dell’account

In ogni caso, una volta registrato nel sistema un account deve essere verificato (operazione a carico della classe che gestisce l’*Authentication Database*): l’utente riceverà un link via e-mail e cliccando su esso l’account sarà verificato.

Il sopracitato link fa riferimento ad un servizio del web server che attende come parametri lo *username* dell’utente ed un *token*: quest’ultimo token viene generato dalla classe che si occupa dell’*Authentication Database* nel momento in cui l’*entry* relativa ad un nuovo utente viene creata; nella procedura di verifica dell’account, il token contenuto nel link deve essere lo stesso di quello contenuto nell’ l’*Authentication Database*.

In alternativa, un account può essere verificato dal form di login, inserendo lo username ed utilizzando la funzionalità “Password dimenticata” (vedere il paragrafo “Password dimenticata” a pagina 10): eseguendo il login con la password temporanea, l’account sarà considerato verificato (visto che la procedura di recupero della password richiede l’utilizzo dell’email fornita durante la registrazione, la verifica dell’email è implicita).

In pratica, grazie alla verifica dell’account, il sistema accerta la corrispondenza tra le credenziali di un utente ed il suo indirizzo email.

Il nuovo utente esiste già nel sistema dal momento in cui viene creato, anche se l’account non è ancora stato verificato; ciò significa che, se ad esempio il nuovo utente è un *Consumer*, un *Uploader* potrà caricare dei documenti per il *Consumer* anche prima che quest’ultimo verifichi l’account, ma il *Consumer* non potrà accedere alla piattaforma se non verifica prima l’account.

#### Registrazione di un Consumer

Un *Consumer* può registrarsi alla piattaforma in modo spontaneo attraverso l’apposita pagina. In questo caso, la password viene scelta dal *Consumer* stesso, quindi la password non sarà inviata nella mail per la verifica dell’account.

## Modifica di un attore

Con “modifica di un attore” si intende la modifica delle sue informazioni salvate nel sistema.

Ogni attore può modificare sé, come da requisiti di progetto.

Un *Uploader* può modificare un *Consumer*; un *Administrator* può modificare un *Uploader* o un altro *Administrator*.

Si supponga che un *Administrator* modifichi il nome di un altro attore: tali modifiche saranno subito visibili dall’*Administrator* che le ha effettuate, ma, supponendo che l’attore soggetto di modifiche sia autenticato durante l’effettuazione delle modifiche, quest’ultimo attore vedrà il suo nome modificato solamente dopo essersi ri-autenticato; ciò è dovuto al fatto che il nome mostrato nell’interfaccia grafica viene recuperato dal token JWT di autenticazione dell’utente e la modifica del nome non comporta l’invalidazione del token, seppure la modifica sia già stata effettuata nel database. Si consideri comunque che la modifica delle informazioni di un attore non è considerata un’operazione frequente e comunque lo stato di inconsistenza tra le informazioni (non significative) nel database e quelle mostrate nell’interfaccia grafica dell’utente che subisce le modifiche ha breve durata (non superiore della durata di un token di autenticazione). Questo “fastidio” potrebbe essere risolto aggiungendo un controllo della consistenza delle informazioni contenute nel token rispetto a quelle nel database tramite un filtro intercettatore nel web server, ma ciò comporterebbe un accesso in lettura al database per ogni richiesta, motivo per cui non è stato implementato. Nel caso in cui un utente modifichi il proprio nome, allora la modifica è subito visibile nell’interfaccia grafica.

## Eliminazione di un attore

### Eliminazione di un Consumer

Un *Uploader* può eliminare un *Consumer*. Il *Consumer* eliminato non viene eliminato dal sistema, ma viene rimosso dall’elenco dei *Consumer* dall’*Uploader* che ne ha richiesto la modifica. Si supponga infatti che un *Consumer* abbia ricevuto dei documenti da più *Uploader*, ad es. da una banca e da un’assicurazione, e si supponga che l’assicurazione decida di rimuovere quel *Consumer*: il *Consumer* perderebbe l’accesso anche a tutti i documenti caricatigli dalla banca e, pensando ad un’applicazione del mondo reale, si è pensato che tale comportamento fosse poco realistico. Se un *Uploader* richiede l’eliminazione di un *Consumer*, i documenti caricati da quell’*Uploader* vengono eliminati ed il *Consumer* non potrà più vedere quell’*Uploader* nell’elenco. I documenti non vengono effettivamente eliminati dal database: viene cancellato il contenuto *blob* (per risparmiare spazio di memoria), mantenendo le altre informazioni utilizzate per il calcolo del *Resoconto* relativo ad un *Uploader* da mostrare ad un *Administrator* (come da specifiche).

### Eliminazione di un Uploader

Un *Administrator* può eliminare un *Uploader*. L’*Uploader* verrà fisicamente eliminato dal sistema e verranno anche eliminati tutti i documenti caricati da quell’*Uploader*.

### Eliminazione di un Administrator

Un *Administrator* può eliminare un *Administrator* (anche sé).

# Serializzazione JSON

Quando i servizi esposti dal server restituiscono un oggetto in formato JSON, la serializzazione viene eseguita da *Jersey* che, nel caso di classi derivate (ereditarietà), provvede ad aggiungere un attributo “*type*” il cui valore è il nome della classe derivata di appartenenza dell’oggetto serializzato, scritta in minuscolo.

# Logo degli Uploader

Le immagini “logo” degli *Uploader* vengono:

* + gestite nelle classi *Java* come array di byte;
  + restituito agli utenti con il *content-type* dedotto dall’estensione del file contenente il logo attraverso un servizio REST.

Il logo degli Uploader viene restituito ai client dopo che questi ne fanno richiesta: l’uri a cui richiedere il logo è l’attributo *src* di un elemento HTML di tipo *img*. Poiché il logo di un *Uploader* è una proprietà modificabile e poiché si desidera vedere le modifiche immediatamente dopo averle effettuate, per evitare fenomeni di cache indesiderati e per forzare il browser a richiedere il nuovo logo appena modificato, l’uri a cui richiedere l’immagine logo (cioè l’attributo *src*) è stato così costruito:

src=”uriImaggineLogo + '?' + new Date().getTime()”

In questo modo, le richieste faranno riferimento allo stesso servizio del server, ma avranno delle *query-string* che permettono alle modifiche di propagarsi con effetto immediato ogni volta che il logo viene modificato (in particolare, in questa applicazione la modifica del logo viene segnalata tramite la generazione dell’evento *logo-attore-modificato*: il componente *Vue* preposto a mostrare il logo dell’*Uploader*, quando riceve tale evento, modificherà la *query string* come spiegato sopra, provocando la richiesta del nuovo logo – infatti il valore *new Date().getTime()* è diverso ad ogni istante).

# Utilizzo dei metodi HTTP *POST, PUT* e *PATCH*

In questo progetto è stato utilizzato il metodo *POST* nelle richieste di creazione e di modifica delle entità salvate nel database perché gli URI specificati nelle richieste non fanno mai riferimento ad una particolare risorsa, bensì al servizio (alla servlet) che si occuperà di creare o modificare la risorsa in oggetto, pervenuta tramite il corpo della request e, con riferimento alle specifiche del protocollo HTTP, il metodo POST è stato ritenuto il più adeguato.

## Form di login

Nei form di *login* si è utilizzato il metodo POST per trasmettere le credenziali, seppur semanticamente sarebbe stato corretto usare il metodo GET. Supponendo di usare HTTP, in entrambi i casi le credenziali verrebbero trasmesse in chiaro (nel corpo della richiesta usando POST, oppure come query-string concatenata all’url usando GET). Usando il metodo GET in HTTPS, le credenziali potrebbero essere trasmesse in chiaro durante la risoluzione DNS oppure potrebbero venire memorizzate in chiaro nella cache o nella cronologia del browser; invece, usando POST, le credenziali verrebbero trasmesse nel corpo della richiesta dopo aver già instaurato il canale TLS, quindi risulta più sicuro usare il metodo POST.

# File *index.yaml* per gli indici del Datastore

Il file src/main/webapp/WEB-INF/index.yaml è un file di configurazione per *App Engine Datastore*: esso definisce gli indici composti necessari al *Datastore* per eseguire le query “complesse”. Gli indici necessari per l’esecuzione delle query “semplici” vengono creati automaticamente. (Fonte: <https://cloud.google.com/appengine/docs/standard/java/configuring-datastore-indexes-with-index-yaml>).

In questo progetto, gli indici composti sono stati generati automaticamente dall'emulatore locale del *Datastore* (avviando il server locale -in modalità di sviluppo- e testando tutte le funzionalità dell’applicazione in modo da eseguire tutte le possibili query) ed automaticamente salvati in target/[nomeprogetto]/WEB-INF/appengine-generated/datastore-indexes-auto.xml durante la stessa fase di test. In seguito, il contenuto di tale file XML è stato tradotto nel file src/main/webapp/WEB-INF/index.yaml, seguendo la documentazione riportata nel link sopraindicato. Il deploying di quest’ultimo file (index.yaml) deve essere fatto manualmente (oltre al normale deploying dell’applicazione) utilizzando il comando

gcloud app deploy src\main\webapp\WEB-INF\index.yaml

eseguito dal terminale posizionato nella cartella radice del progetto.

# Deploying dell’applicazione

Per eseguire il deploying dell’applicazione nei server di *Google App Engine*, bisogna eseguire i seguenti comandi dal terminale posizionato nella cartella radice del progetto *Maven*:

* + - 1. gcloud app deploy src\main\webapp\WEB-INF\index.yaml

per e seguire il deploying del file *index.yaml* (vedere paragrafo “File *index.yaml* per gli indici del Datastore” a pagina 22);

* + - 1. mvn clean validate compile test package site verify appengine:deploy

per creare il file *WAR* dell’applicazione ed eseguirne il deploying nel servlet container (*Google App Engine* in questo progetto), seguendo tutte le fasi *Maven* (*clean, validate, compile, test, package, site, verify*).

# Note sul funzionamento della piattaforma

## Primo avvio della piattaforma

All’avvio della piattaforma per la prima volta, ci sarà un solo utente amministratore, con credenziali impostate nel file src/main/webapp/WEB-INF/credenziali/credenzialiPrimoAmministratoreAllAvvioDellaPiattaforma.json

In ambiente di sviluppo, all’avvio del server vengono create ulteriori entità utilizzate per finalità di test (vedere classe (*listener*) *StarterDatabase.java*).

# HTTPS

Il web server è stato configurato per funzionare con il protocollo HTTPS; ciò è stato fatto aggiungendo il seguente *serity constraint* nel file *web.xml*.

    <security-constraint>  
        <web-resource-collection>  
            <web-resource-name>profile</web-resource-name>  
            <url-pattern>/\*</url-pattern>  
        </web-resource-collection>  
        <user-data-constraint>  
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>  
        </user-data-constraint>  
    </security-constraint>

Il server di sviluppo non supporta HTTPSe l’applicazione può essere testata usando HTTP. Nel caso in cui un client tenti di utilizzare il protocollo HTTP, viene reindirizzato cosicché utilizzi HTTPS.(Fonte: <https://cloud.google.com/appengine/docs/standard/java/config/webxml#secure-urls>)

1. <https://babeljs.io/docs/en/#:~:text=Babel%20is%20a%20JavaScript%20compiler,and%20older%20browsers%20or%20environments.&text=Polyfill%20features%20that%20are%20missing,(through%20%40babel%2Fpolyfill)> [↑](#footnote-ref-1)
2. <https://www.npmjs.com/package/core-js> [↑](#footnote-ref-2)
3. <https://eslint.vuejs.org/> [↑](#footnote-ref-3)
4. <https://lmiller1990.github.io/vue-testing-handbook/v3/> [↑](#footnote-ref-4)
5. <https://jestjs.io/en/> [↑](#footnote-ref-5)
6. <https://github.com/vuejs/vue-cli/tree/dev/packages/%40vue/cli-plugin-unit-jest> [↑](#footnote-ref-6)
7. “*Lax replaced None as the default value in order to ensure that users have reasonably robust defense against some classes of cross-site request forgery (CSRF) attacks”* <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite> *.* [↑](#footnote-ref-7)
8. Si è preferito usare il metodo POST perché l’url di destinazione della richiesta indica il servizio che si occupa del caricamento: tale url, non è un identificativo per la risorsa, quindi si è ritenuto più corretto (dal punto di vista semantico) l’utilizzo del metodo POST anziché di PUT. [↑](#footnote-ref-8)
9. La chiave pubblica per la verifica della firma emessa dal server sui token potrebbe essere anche inclusa nel *payload* dei *token JWT*. [↑](#footnote-ref-9)