Progetto di *Programmazione Web*

NOTE PROGETTUALI

# Vue.js

PARAGRAFO NON AGGIORNATO

Si è utilizzato *Vue 3*.

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 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*.

# 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 e non stateless. L’implementazione separa adeguatamente le responsabilità circa le richieste REST da quelle relative a generazione e verifica dei token JWT e ciò rende semplice la modifica in un secondo momento. Tra i benefici: la verifica delle autorizzazioni descritte dal JWT non richiede 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 (ciò simile a quanto richiesto dai requisiti di questo progetto), l’utilizzo di token anziché di sessione potrebbe semplificare l’integrazione di ulteriori servizi, 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, allora il client caricherebbe l’interfaccia basandosi su tali informazioni alterate, ma in ogni caso non potrà eseguire 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.

## CSRF

Il form di login utilizza un campo contenente un token utilizzato per scongiurare gli attacchi CSRF. Tale token viene generato casualmente dal server, che lo restituisce al client, quando quest’ultimo ne fa richiesta, insieme a due cookie: il primo contiene un identificativo (cioè un token casuale) per il client, il secondo contiene un token JWT, all’interno del cui payload vi è sia il token CSRF sia l’identificativo del client contenuto nel primo token. I cookie sono inviati in modalità *SameSite* (opzione *Lax*) e *HttpOnly*, per ragioni di sicurezza. (se il server su cui viene fatto il *deploying* della *web application* lo permette, i cookie possono essere impostati per essere di tipo *Secure*). 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> .

In caso di login fallito (credenziali non valide) il token CSRF non viene ricaricato, perché, grazie alla policy *CORS* opportunamente configurata, da un eventuale sito esterno che tenti la forzatura di questa applicazione non sarebbe possibile poter accedere alla risposta, perciò non potrebbe venire a conoscenza del token CSRF usato.

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

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.

## 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)) ed *HttpOnly* (per prevenirne l’accesso da parte del livello applicativo)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 e form senza auto-completamento.
  + I client si autenticano attraverso form e, se la procedura di login ha esito positivo, ricevono dal server un token *JWT* trasmesso come valore di un cookie. Si sarebbe potuto inviare tale token al client direttamente contenuto nel body di una response, ma in tal caso vi sarebbero state due possibili opzioni:
    1. Salvare il token nella memoria accessibile dal browser, cosicché l’applicativo client potesse accedervi: questa opzione è stata evitata perché rendere il token accessibile ad uno script potrebbe esporre l’applicazione alle vulnerabilità XSS (nonostante “*Modern JavaScript frameworks –* tra cui Vue 3 – *have pretty good XSS protection built in*”, Fonte: <https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#bonus-rule-4-properly-use-modern-js-frameworks>, consultata il 5 gennaio 2021).
    2. Salvare il token in una variabile dello script, che sarebbe stata però persa una volta ricaricata la pagina e ciò avrebbe costretto l’utente a ripetere la procedura di login interrompendo quindi ciò che stava facendo prima del ricaricamento della pagina.

**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 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 con la password (*hashed* e *salted*) del client è presente in un claim del token salvato nella memoria del browser: in questo modo il client si autenticherà presso il server presentando il token JWT nell’header *Authorization* 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 la sua versione cifrata con 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 per cifrare il claim con il token, allora si fornirebbero informazioni circa l’hash utilizzato dal server per firmare i token, visto che in questo modo si fornirebbero sia il token in chiaro sia la sua versione cifrata).
  + 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, 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, possono succedere le seguenti cose:

* + se *non* si esegue il comando *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 *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.

La soluzione, a fini di test, in ambiente di sviluppo, è riavviare il server di sviluppo, utilizzando anche il comando *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.

## 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, infine 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 J diventa inutilizzabile. Inoltre il client, al logout, cancella il token dalla memoria a disposizione del browser. Vedere “AuthenticationTokenInvalido” a pagina 5.

# Cookie e token utilizzati

## 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 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*.
  + **hash-TOKEN-ID-CLIENT-AUTENTICAZIONE** contiene il valore di hash (calcolato usando come chiave l’hash della password dell’attore) del token di autenticazione) di un token casuale associato al client: il valore di tale token (in chiaro, senza hash) è 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, comunque 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; 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 (anch’esse indipendenti dalla specifica richiesta).

## 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 inoltre hanno 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, acnhe 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.



# Registrazione, creazione ed eliminazione di un Consumer

## 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 contente un link 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’ l’*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 16): eseguendo il login con la password temporanea, l’account sarà considerato verificato.

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

Tuttavia, 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.

### Creazione di un Attore

Un *Uploader* può creare 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.

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, motivo per cui l’utente è invitato alla modifica della password al primo accesso.

## Eliminazione di un Attore

Un *Uploader* può eliminare un *Consumer*: quest’operazione comporta l’eliminazione di tale *Consumer* dalla lista di quelli gestiti dall’*Uploader* in oggetto e dell’eliminazione di tutti i file caricati dal medesimo *Uploader* per tale *Consumer*. Il *Consumer* non viene eliminato dalla piattaforma, in quanto potrebbe essere comunque servito da un altro *Uploader*.

Un *Administrator* può eliminare completamente un *Uploader* dalla piattaforma e con tale operazione vengono eliminati anche tutti i file ad egli inviati.

Vedere anche il paragrafo “15Eliminazione di un attore” a pagina 15.

# Filtri

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, comunque, 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 dahttps://stackoverflow.com/q/33247716 ).

## Eliminazione

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.

# Utilizzo di mappe 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*.

# E-mail

La gestione delle email è stata fatta usando *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).

# Data-URI per il logo degli *Uploader*

Le immagini “logo” degli *Uploader* vengono:

* + gestite nelle classi Java come array di byte;
  + salvate nel database come attributo dell’entità *Uploader* a cui si riferiscono, in formato *Blob*;
  + restituite al client secondo la specifica “Data-URI” (*RFC2397:* [*https://tools.ietf.org/html/rfc2397*](https://tools.ietf.org/html/rfc2397) *,* [*https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics\_of\_HTTP/Data\_URIs*](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs)): in questo modo è possibile gestire le immagini come stringhe. Ciò è possibile assumendo che un’immagine “logo” abbia una dimensione molto contenuta.[[10]](#footnote-10)

# 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*.

# Pattern per la richiesta dati al 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).

# 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.

# 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.

# 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. 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 (tale modifica non verrà mostrata all’utente a meno che non ricarichi la pagina) o che un documento venga eliminato ed un altro aggiunto: in questo caso il server penserà che non ci sono modifiche e la modifica non si propagherà sul client (aggiornando il componente, la modifica apparirà).

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

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 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.

# Modifica di un attore

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 vi è uno stato inconsistente di 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 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.

## 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 se stesso).

# Serializzazione JSON

Quando i servizi esposti dal server restituiscono un oggetto in formato JSON, la serializzazione viene eseguita da JAX-RS 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

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 stringhe* 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).

# 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.

# 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 *heshed and salted* in cui viene memorizzato lo username degli utenti, il *salt* e l’*hash* della password (concatenata con il *salt*, 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, salvata in chiaro ed utilizzata solamente in modo temporaneo per il recupero della password (come spiegato nel paragrafo “Password dimenticata”).

## Verifica dell’account

Prima di poter utilizzare un account ai fini della procedura di autenticazione, è necessario verificarlo. Vedere il paragrafo “Verifica dell’account” a pagina 8.

## Password dimenticata

Il form di login prevede un link per il reset della password nel caso in cui questa venisse dimenticata dall’account il cui username deve essere inserito nel soprascritto form. Lo username viene inviato tramite 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 chiaro e viene inviata in chiaro via mail all’utente, per questo motivo è fortemente consigliato di modificarla al primo accesso.

## 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 sia autenticato 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 si afferma che l’email è effettivamente associata all’utente che si è appena autenticato sul client)
    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 16, “Sicurezza e Autenticazione” a pagina 3 e “Cookie e token utilizzati” a pagina 5).

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.

Questo sistema è ritenuto sicuro, in quanto l’informazione contente l’e-mail dell’utente giunge al server dell’applicazione contenuta in un token JWT firmato, quindi è accertata l’associazione tra l’utente e l’e-mail: se tale indirizzo e-mail è salvato nel sistema, poiché l’indirizzo e-mail è utilizzato dall’applicazione come strumento di verifica dell’identità di un utente (infatti: 1. un account a seguito della registrazione nel sistema deve essere verificato e tale verifica avviene tramite e-mail (si veda il paragrafo “Verifica dell’account” a pagina 8); 2. il reset della password di un account è possibile grazie all’indirizzo e-mail (si veda “Password dimenticata” a pagina 16)).

**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.

# Limitazione della piattaforma: non possono esistere più account associati alla stessa e-mail

Poiché si è utilizzato *Firebase* per l’autenticazione come descritto nel paragrafo “Firebase” a pagina 16, il server di questa applicazione fa uso dell’e-mail ottenuta tramite *Firebase* per riconoscere quale utente si sta autenticando, perciò non possono esistere più account nella piattaforma con la stessa e-mail.

**Nota:** prima di utilizzare *Firebase*, potevano esistere più account con la stessa email perché l’autenticazione era basata sullo *username* dell’attore, il quale era univoco (e più *username* potevano essere associati allo stesso indirizzo e-mail).

# 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.

## Metodo *POST*

Dalla specifica RFC7231 relativa al metodo *POST*, sezione 4.3.3 (https://tools.ietf.org/html/rfc7231#section-4.3.3), si legge:

The POST method requests that the target resource process the

representation enclosed in the request according to the resource's

own specific semantics. For example, POST is used for the following

functions (among others):

o Providing a block of data, such as the fields entered into an HTML

form, to a data-handling process;

o Posting a message to a bulletin board, newsgroup, mailing list,

blog, or similar group of articles;

o Creating a new resource that has yet to be identified by the

origin server; and

o Appending data to a resource's existing representation(s).

An origin server indicates response semantics by choosing an

appropriate status code depending on the result of processing the

POST request; almost all of the status codes defined by this

specification might be received in a response to POST (the exceptions

being 206 (Partial Content), 304 (Not Modified), and 416 (Range Not

Satisfiable)).

If one or more resources has been created on the origin server as a

result of successfully processing a POST request, the origin server

SHOULD send a 201 (Created) response containing a Location header

field that provides an identifier for the primary resource created

([Section 7.1.2](https://tools.ietf.org/html/rfc7231#section-7.1.2)) and a representation that describes the status of the

request while referring to the new resource(s).

Responses to POST requests are only cacheable when they include

explicit freshness information (see [Section 4.2.1 of [RFC7234]](https://tools.ietf.org/html/rfc7234#section-4.2.1)).

However, POST caching is not widely implemented. For cases where an

origin server wishes the client to be able to cache the result of a

POST in a way that can be reused by a later GET, the origin server

MAY send a 200 (OK) response containing the result and a

Content-Location header field that has the same value as the POST's

effective request URI ([Section 3.1.4.2](https://tools.ietf.org/html/rfc7231#section-3.1.4.2)).

If the result of processing a POST would be equivalent to a

representation of an existing resource, an origin server MAY redirect

the user agent to that resource by sending a 303 (See Other) response

with the existing resource's identifier in the Location field. This

has the benefits of providing the user agent a resource identifier

and transferring the representation via a method more amenable to

shared caching, though at the cost of an extra request if the user

agent does not already have the representation cached.

## Metodo *PUT*

Dalla specifica RFC7231 relativa al metodo *PUT*, sezione 4.3.4 (https://tools.ietf.org/html/rfc7231#section-4.3.4), si legge:

The PUT method requests that the state of the target resource be

created or replaced with the state defined by the representation

enclosed in the request message payload. A successful PUT of a given

representation would suggest that a subsequent GET on that same

target resource will result in an equivalent representation being

sent in a 200 (OK) response. However, there is no guarantee that

such a state change will be observable, since the target resource

might be acted upon by other user agents in parallel, or might be

subject to dynamic processing by the origin server, before any

subsequent GET is received. A successful response only implies that

the user agent's intent was achieved at the time of its processing by

the origin server.

If the target resource does not have a current representation and the

PUT successfully creates one, then the origin server MUST inform the

user agent by sending a 201 (Created) response. If the target

resource does have a current representation and that representation

is successfully modified in accordance with the state of the enclosed

representation, then the origin server MUST send either a 200 (OK) or

a 204 (No Content) response to indicate successful completion of the

request.

An origin server SHOULD ignore unrecognized header fields received in

a PUT request (i.e., do not save them as part of the resource state).

An origin server SHOULD verify that the PUT representation is

consistent with any constraints the server has for the target

resource that cannot or will not be changed by the PUT. This is

particularly important when the origin server uses internal

configuration information related to the URI in order to set the

values for representation metadata on GET responses. When a PUT

representation is inconsistent with the target resource, the origin

server SHOULD either make them consistent, by transforming the

representation or changing the resource configuration, or respond

with an appropriate error message containing sufficient information

to explain why the representation is unsuitable. The 409 (Conflict)

or 415 (Unsupported Media Type) status codes are suggested, with the

latter being specific to constraints on Content-Type values.

For example, if the target resource is configured to always have a

Content-Type of "text/html" and the representation being PUT has a

Content-Type of "image/jpeg", the origin server ought to do one of:

a. reconfigure the target resource to reflect the new media type;

b. transform the PUT representation to a format consistent with that

of the resource before saving it as the new resource state; or,

c. reject the request with a 415 (Unsupported Media Type) response

indicating that the target resource is limited to "text/html",

perhaps including a link to a different resource that would be a

suitable target for the new representation.

HTTP does not define exactly how a PUT method affects the state of an

origin server beyond what can be expressed by the intent of the user

agent request and the semantics of the origin server response. It

does not define what a resource might be, in any sense of that word,

beyond the interface provided via HTTP. It does not define how

resource state is "stored", nor how such storage might change as a

result of a change in resource state, nor how the origin server

translates resource state into representations. Generally speaking,

all implementation details behind the resource interface are

intentionally hidden by the server.

An origin server MUST NOT send a validator header field

([Section 7.2](https://tools.ietf.org/html/rfc7231#section-7.2)), such as an ETag or Last-Modified field, in a

successful response to PUT unless the request's representation data

was saved without any transformation applied to the body (i.e., the

resource's new representation data is identical to the representation

data received in the PUT request) and the validator field value

reflects the new representation. This requirement allows a user

agent to know when the representation body it has in memory remains

current as a result of the PUT, thus not in need of being retrieved

again from the origin server, and that the new validator(s) received

in the response can be used for future conditional requests in order

to prevent accidental overwrites ([Section 5.2](https://tools.ietf.org/html/rfc7231#section-5.2)).

The fundamental difference between the POST and PUT methods is

highlighted by the different intent for the enclosed representation.

The target resource in a POST request is intended to handle the

enclosed representation according to the resource's own semantics,

whereas the enclosed representation in a PUT request is defined as

replacing the state of the target resource. Hence, the intent of PUT

is idempotent and visible to intermediaries, even though the exact

effect is only known by the origin server.

Proper interpretation of a PUT request presumes that the user agent

knows which target resource is desired. A service that selects a

proper URI on behalf of the client, after receiving a state-changing

request, SHOULD be implemented using the POST method rather than PUT.

If the origin server will not make the requested PUT state change to

the target resource and instead wishes to have it applied to a

different resource, such as when the resource has been moved to a

different URI, then the origin server MUST send an appropriate 3xx

(Redirection) response; the user agent MAY then make its own decision

regarding whether or not to redirect the request.

A PUT request applied to the target resource can have side effects on

other resources. For example, an article might have a URI for

identifying "the current version" (a resource) that is separate from

the URIs identifying each particular version (different resources

that at one point shared the same state as the current version

resource). A successful PUT request on "the current version" URI

might therefore create a new version resource in addition to changing

the state of the target resource, and might also cause links to be

added between the related resources.

An origin server that allows PUT on a given target resource MUST send

a 400 (Bad Request) response to a PUT request that contains a

Content-Range header field ([Section 4.2 of [RFC7233]](https://tools.ietf.org/html/rfc7233#section-4.2)), since the

payload is likely to be partial content that has been mistakenly PUT

as a full representation. Partial content updates are possible by

targeting a separately identified resource with state that overlaps a

portion of the larger resource, or by using a different method that

has been specifically defined for partial updates (for example, the

PATCH method defined in [[RFC5789](https://tools.ietf.org/html/rfc5789)]).

Responses to the PUT method are not cacheable. If a successful PUT

request passes through a cache that has one or more stored responses

for the effective request URI, those stored responses will be

invalidated (see [Section 4.4 of [RFC7234]](https://tools.ietf.org/html/rfc7234#section-4.4)).

## Metodo *DELETE*

Dalla specifica RFC7231 relativa al metodo *DELETE*, sezione 4.3.5 (https://tools.ietf.org/html/rfc7231#section-4.3.5), si legge:

The DELETE method requests that the origin server remove the

association between the target resource and its current

functionality. In effect, this method is similar to the rm command

in UNIX: it expresses a deletion operation on the URI mapping of the

origin server rather than an expectation that the previously

associated information be deleted.

If the target resource has one or more current representations, they

might or might not be destroyed by the origin server, and the

associated storage might or might not be reclaimed, depending

entirely on the nature of the resource and its implementation by the

origin server (which are beyond the scope of this specification).

Likewise, other implementation aspects of a resource might need to be

deactivated or archived as a result of a DELETE, such as database or

gateway connections. In general, it is assumed that the origin

server will only allow DELETE on resources for which it has a

prescribed mechanism for accomplishing the deletion.

Relatively few resources allow the DELETE method -- its primary use

is for remote authoring environments, where the user has some

direction regarding its effect. For example, a resource that was

previously created using a PUT request, or identified via the

Location header field after a 201 (Created) response to a POST

request, might allow a corresponding DELETE request to undo those

actions. Similarly, custom user agent implementations that implement

an authoring function, such as revision control clients using HTTP

for remote operations, might use DELETE based on an assumption that

the server's URI space has been crafted to correspond to a version

repository.

If a DELETE method is successfully applied, the origin server SHOULD

send a 202 (Accepted) status code if the action will likely succeed

but has not yet been enacted, a 204 (No Content) status code if the

action has been enacted and no further information is to be supplied,

or a 200 (OK) status code if the action has been enacted and the

response message includes a representation describing the status.

A payload within a DELETE request message has no defined semantics;

sending a payload body on a DELETE request might cause some existing

implementations to reject the request.

Responses to the DELETE method are not cacheable. If a DELETE

request passes through a cache that has one or more stored responses

for the effective request URI, those stored responses will be

invalidated (see [Section 4.4 of [RFC7234]](https://tools.ietf.org/html/rfc7234#section-4.4)).

## Metodo *PATCH*

Dalla specifica RFC5789 relativa al metodo *PATCH* (https://tools.ietf.org/html/rfc5789), si legge:

This specification defines the new HTTP/1.1 [[RFC2616](https://tools.ietf.org/html/rfc2616)] method, PATCH,

which is used to apply partial modifications to a resource.

A new method is necessary to improve interoperability and prevent

errors. The PUT method is already defined to overwrite a resource

with a complete new body, and cannot be reused to do partial changes.

Otherwise, proxies and caches, and even clients and servers, may get

confused as to the result of the operation. POST is already used but

without broad interoperability (for one, there is no standard way to

discover patch format support). PATCH was mentioned in earlier HTTP

specifications, but not completely defined.

The PATCH method requests that a set of changes described in the

request entity be applied to the resource identified by the Request-

URI. The set of changes is represented in a format called a "patch

document" identified by a media type. If the Request-URI does not

point to an existing resource, the server MAY create a new resource,

depending on the patch document type (whether it can logically modify

a null resource) and permissions, etc.

The difference between the PUT and PATCH requests is reflected in the

way the server processes the enclosed entity to modify the resource

identified by the Request-URI. In a PUT request, the enclosed entity

is considered to be a modified version of the resource stored on the

origin server, and the client is requesting that the stored version

be replaced. With PATCH, however, the enclosed entity contains a set

of instructions describing how a resource currently residing on the

origin server should be modified to produce a new version. The PATCH

method affects the resource identified by the Request-URI, and it

also MAY have side effects on other resources; i.e., new resources

may be created, or existing ones modified, by the application of a

PATCH.

PATCH is neither safe nor idempotent as defined by [[RFC2616](https://tools.ietf.org/html/rfc2616)], [Section](https://tools.ietf.org/html/rfc2616#section-9.1)

[9.1](https://tools.ietf.org/html/rfc2616#section-9.1).

A PATCH request can be issued in such a way as to be idempotent,

which also helps prevent bad outcomes from collisions between two

PATCH requests on the same resource in a similar time frame.

Collisions from multiple PATCH requests may be more dangerous than

PUT collisions because some patch formats need to operate from a

known base-point or else they will corrupt the resource. Clients

using this kind of patch application SHOULD use a conditional request

such that the request will fail if the resource has been updated

since the client last accessed the resource. For example, the client

can use a strong ETag [[RFC2616](https://tools.ietf.org/html/rfc2616)] in an If-Match header on the PATCH

request.

There are also cases where patch formats do not need to operate from

a known base-point (e.g., appending text lines to log files, or non-

colliding rows to database tables), in which case the same care in

client requests is not needed.

The server MUST apply the entire set of changes atomically and never

provide (e.g., in response to a GET during this operation) a

partially modified representation. If the entire patch document

cannot be successfully applied, then the server MUST NOT apply any of

the changes. The determination of what constitutes a successful

PATCH can vary depending on the patch document and the type of

resource(s) being modified. For example, the common 'diff' utility

can generate a patch document that applies to multiple files in a

directory hierarchy. The atomicity requirement holds for all

directly affected files. See "Error Handling", [Section 2.2](https://tools.ietf.org/html/rfc5789#section-2.2), for

details on status codes and possible error conditions.

If the request passes through a cache and the Request-URI identifies

one or more currently cached entities, those entries SHOULD be

treated as stale. A response to this method is only cacheable if it

contains explicit freshness information (such as an Expires header or

"Cache-Control: max-age" directive) as well as the Content-Location

header matching the Request-URI, indicating that the PATCH response

body is a resource representation. A cached PATCH response can only

be used to respond to subsequent GET and HEAD requests; it MUST NOT

be used to respond to other methods (in particular, PATCH).

Note that entity-headers contained in the request apply only to the

contained patch document and MUST NOT be applied to the resource

being modified. Thus, a Content-Language header could be present on

the request, but it would only mean (for whatever that's worth) that

the patch document had a language. Servers SHOULD NOT store such

headers except as trace information, and SHOULD NOT use such header

values the same way they might be used on PUT requests. Therefore,

this document does not specify a way to modify a document's Content-

Type or Content-Language value through headers, though a mechanism

could well be designed to achieve this goal through a patch document.

There is no guarantee that a resource can be modified with PATCH.

Further, it is expected that different patch document formats will be

appropriate for different types of resources and that no single

format will be appropriate for all types of resources. Therefore,

there is no single default patch document format that implementations

are required to support. Servers MUST ensure that a received patch

document is appropriate for the type of resource identified by the

Request-URI.

Clients need to choose when to use PATCH rather than PUT. For

example, if the patch document size is larger than the size of the

new resource data that would be used in a PUT, then it might make

sense to use PUT instead of PATCH. A comparison to POST is even more

difficult, because POST is used in widely varying ways and can

encompass PUT and PATCH-like operations if the server chooses. If

the operation does not modify the resource identified by the Request-

URI in a predictable way, POST should be considered instead of PATCH

or PUT.

PATCH /file.txt HTTP/1.1

### A Simple PATCH Example

Host: www.example.com

Content-Type: application/example

If-Match: "e0023aa4e"

Content-Length: 100

[description of changes]

This example illustrates use of a hypothetical patch document on an

existing resource.

Successful PATCH response to existing text file:

HTTP/1.1 204 No Content

Content-Location: /file.txt

ETag: "e0023aa4f"

The 204 response code is used because the response does not carry a

message body (which a response with the 200 code would have). Note

that other success codes could be used as well.

Furthermore, the ETag response header field contains the ETag for the

entity created by applying the PATCH, available at

http://www.example.com/file.txt, as indicated by the Content-Location

response header field.

La specifica del metodo *PATCH* fornisce maggiori informazioni, qui omesse.

# Disclaimer

## Utilizzo di cookie

L’applicazione fa uso di cookie, senza i quali non può funzionare. Non si tratta di cookie di profilazione.

## Proprietà degli *Uploader*

Le proprietà “pubbliche” (quali nominativo e logo, ad esempio) degli *Uploader* sono accessibili a tutti gli utenti autenticati, indipendentemente che essi siano *Consumer*, *Uploader* o *Administrator* ed indipendentemente se sono o meno in relazione con l’*Uploader* per il quale richiedono le proprietà.

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

Il file src/main/webapp/WEB-INF/index.yaml è un file di configurazione per *AppEngine*: 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 23);

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

per creare il file *WAR* dell’applicazione ed eseguirne il deploying, 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 username [***admin***]e password [***\*\*\*\****]. In ambiente di sviluppo compariranno ulteriori entità utilizzate per finalità di test.

## Creazione/modifica/eliminazione di un Consumer

Un *Consumer* può essere associato a zero (se si è registrato autonomamente) o più *Uploader* (se uno o più *Uploader* hanno aggiunto quel *Consumer* alla lista di quelli da essi gestiti). Un *Uploader* può aggiungere un *Consumer* inserendo i suoi campi: se quel *Consumer* 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.

Un *Uploader* può modificare un *Consumer* e tali modifiche sono visibili sia al *Consumer* in oggetto sia a tutti gli *Uploader* che lo gestiscono.

Un *Uploader* può eliminare un *Consumer* nel senso che quel *Consumer* non comparirà più nell’elenco di quelli gestiti dall’*Uploader* che ne ha richiesto l’eliminazione, ma quel *Consumer* resterà ancora registrato nel sistema (potrebbe essere associato anche ad altri *Uploader*).

# Client REST

È possibile distribuire il file *JAR* relativo al client REST. Si veda il file README del progetto.

# 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*. 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>)

    <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>

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)
10. Secondo <https://caniuse.com/datauri>, la maggior parte dei browser (escluso *Internet Explorer* nelle versioni precedenti la 8) supportano *Data URI*s, utilizzato per le immagini, con lunghezza massima dell’URL:

    nel caso peggiore, di 32 KB (browser: *IE 8* );

    di 4 GB (browser: *IE 9-11, Edge 12-18*);

    non specificata in altri browser.

    Per quanto appena scritto, la lunghezza massima accettabile per le immagini “logo” è controllata nel codice Java, considerando i precedenti valori massimi, riferiti agli *uri* già codificati (non alla dimensione dell’immagine caricata). [↑](#footnote-ref-10)