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

L’idea è di informare tramite alert la possibile incompatibilità della web app con il browser attualmente usato dall’utente e suggerirgli di cambiare browser, tramite alert. Come fare?

1. Client invia la prima richiesta al server (non c’è ancora una sessione)
2. Il server funziona regolarmente
3. Il server verifica tramite un filtro d’uscita se il parametro della sessione userAgent è:
   1. *NOT\_COMPATIBLE || COMPATIBLE* allora il filtro d’uscita non fa nulla, altrimenti
   2. *null* allora inietta nella response uno script per determinare lo user agent [[1]](#footnote-1), altrimenti
   3. tra quelli incompatibili allora inietta nella response uno script che produrrà un alert per informare l’utente della possibile incompatibilità con la web app ed imposta il parametro *userAgent* della request ad *NOT\_COMPATIBLE*, altrimenti
   4. tra quelli compatibili allora imposta il parametro *userAgent* della request ad *COMPATIBLE*

L’idea è di fare richieste Ajax per riempire la pagina cosicché la webapp sia una single page application. Ecco come riempire una tabella [[2]](#footnote-2):

1. fetch('<some URL>')
2. .then((response) => {
3. let data = response.json(); // Let supposed the data is in this format [{ id: 1 }, { id: 2 }, { id: 3 }]
4. let tr = '';
5. let tableBody = document.querySelector('#table\_id');
6. data.forEach(function(value) {
7. let tr = document.createElement('tr');
8. tr.textContent = data.id
9. tableBody.appendChild(tr);
10. });
12. }).catch(error => {
13. console.log(error);
14. });

 Vedi anche il codice fatto a lezione: 06ServletPlayground (parte 4) in cui si fa una servlet che invia i dati ad una pagina jsp che si autoaggiorna:

* + Classe *AjaxMessageViewer.java* servlet che riceve la richiesta ed invia la risposta con solo il contenuto da aggiornare nella pagina sul client
  + Classe *SelfUpdatingPartial.jsp* jsp contenente solo il contenuto da aggiungere sul client
  + Classe *SelfUpdating.jsp* jsp con lo scheletro della pagina di risposta, che poi si autoaggiornerà.

# Vue.js

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.[[3]](#footnote-3)
* @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. [[4]](#footnote-4)
* 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. [[5]](#footnote-5)

## Test

Per la creazione dei test sono state utilizzate la libreria **vue-test-utils[[6]](#footnote-6)** ed il framework **Jest[[7]](#footnote-7)**. Si è fatto riferimento al link [[8]](#footnote-8) 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).

# 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.

Da fare: mantenere un database con i token jwt inutilizzabili (es. token csrf usabili una volta sola, token di autenticazione utilizzabili solo fino al login) tale che il server, quando gli si presenta un token, controlli la validità (ed in questo controllo deve essere verificato che il token non sia tra quelli inutilizzabili nel soprascritto database). Predisporre un task periodico che periodicamente (una volta l’ora, ad esempio, elimini tutti i token scaduti (quindi invalidi per definizione) dal soprascritto database, per liberare spazio (perché quei token jwt in ogni caso, essendo scaduti, non dovranno poter essere utilizzabili).

## 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. (anche in modalità seecure?? Se server lo permette). 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[[9]](#footnote-9)) 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 CSRF (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 CSRF. 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 CSRF 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 CSRF, 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 riautenticarsi: 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 CSRF senza il quale il token diventa inutilizzabile. Inoltre il client, al logout, cancella il token dalla memoria a disposizione del browser.

# 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*)

## 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*.
  + **sub** contiene un valore alfanumerico casuale che deve essere lo stesso di quello contenuto nel cookie TOKEN-ID-CLIENT-VERIFICA-CSRF (serve questo doppio controllo sui cookie?) Questo valore è diverso da quello specificato nel token di autenticazione, perché questo token serve solo a validare un token CSRF e *non* ad autenticare il client (non ha nulla a che fare con la procedura di autenticazione).
  + **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.

### TOKEN-ID-CLIENT-VERIFICA-CSRF

Questo cookie contiene un valore alfanumerico casuale associato al client per il solo tempo di utilizzo del token CSRF; questo valore è lo stesso di quello memorizzato nel *claim* **sub** del cookie CSRF-TOKEN-JWT.

Quindi, affinché un *form* dotato di CSRF token venga considerato valido dal server, il client deve presentare il cookie CSRF-TOKEN-JWT all’interno di cui deve esserci il valore del subject che deve coincidere con il valore di questo cookie (TOKEN-ID-CLIENT-VERIFICA-CSRF); inoltre, nel cookie CSRF-TOKEN-JWT, deve esserci anche il claim con il valore del CSRF token che deve coincidere con quello nel form presentato dal client.

# 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 Consumer

Un *Consumer* può registrarsi alla piattaforma in modo spontaneo attraverso l’apposita pagina.

## 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*.

## Eliminazione di un Consumer

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.

# 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* [[10]](#footnote-10). 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 sul client

## Mappa { hashtag => [ *lista documenti con quell’hashtag* ] }

## Mappa { idDocumento => [ *properties di quel documento* ] }

## Classe *MappaDocumenti*

Osservazione: il client su cui si collega un *Consumer* riceve una mappa in cui ogni *entry* è associata ad un *Uploader* (quindi un *Consumer* appena effettua il login riceve subito tutte le informazioni di tutti gli *Uploaders*), invece un *Uploader* appena effettua il login riceve subito un *array* con solo gli identificativi dei C*onsumer*. Questa scelta è stata fatta supponendo che generalmente un *Consumer* faccia riferimento a pochi *Uploader* ( decine), invece un *Uploader* generalmente gestisce molti *Consumer* (), quindi nel caso di un *Uploader* il caricamento della schermata iniziale potrebbe diventare molto lento e non porterebbe a grandi miglioramenti, se non che si potrebbero ordinare alfabeticamente i *Consumer*, tuttavia ciò richiederebbe uno sforzo computazionale, oltre che uno spreco di banda nel trasferire informazioni che forse non verranno nemmeno utilizzate.

# Accessi al database

## Registrazione nuovo *Consumer*

La registrazione di un nuovo consumer richiede:

* + 1 query di ricerca per verificare che quell’attore *non* sia già registrato (se si, allora la procedura termina e non sono necessari ulteriori accessi al database);
  + 2 accessi in scrittura: uno per memorizzare le informazioni dell’Attore, l’altro per aggiungere le sue credenziali nel Authentication database (si tratta di entità[[11]](#footnote-11) separate).

## Login utente già registrato (verifica delle credenziali)

La verifica delle credenziali di un utente già registrato (per il login) richiede:

* + 1 accesso in lettura (per chiave) al database di autenticazione (se non presente, la procedura termina qui).

## Autenticazione attore

Per ogni richiesta che giunge al server in cui il client richiede una risorsa protetta, il server effettua *N* accessi al database per: … .

# 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 [[12]](#footnote-12).

# 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.[[13]](#footnote-13)

# 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*.

# 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:

* + molti servizi, utilizzati dall’applicazione *Vue*, erano già stati implementati secondo l’approccio REST;
  + SOAP utilizza XML e ciò comporta maggiore overhead;
  + La libreria *JAX-WS* dedicata ai web service in Java non è supportata da *Google AppEngine*.

## 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>

# 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 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.

# 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.

# 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.

# 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.

1. <https://stackoverflow.com/a/9851769>

   1. // Opera 8.0+
   2. var isOpera = (!!window.opr && !!opr.addons) || !!window.opera || navigator.userAgent.indexOf(' OPR/') >= 0;
   4. // Firefox 1.0+
   5. var isFirefox = typeof InstallTrigger !== 'undefined';
   7. // Safari 3.0+ "[object HTMLElementConstructor]"
   8. var isSafari = /constructor/i.test(window.HTMLElement) || (function (p) { return p.toString() === "[object SafariRemoteNotification]"; })(!window['safari'] || (typeof safari !== 'undefined' && window['safari'].pushNotification));
   10. // Internet Explorer 6-11
   11. var isIE = /\*@cc\_on!@\*/false || !!document.documentMode;
   13. // Edge 20+
   14. var isEdge = !isIE && !!window.StyleMedia;
   16. // Chrome 1 - 79
   17. var isChrome = !!window.chrome && (!!window.chrome.webstore || !!window.chrome.runtime);
   19. // Edge (based on chromium) detection
   20. var isEdgeChromium = isChrome && (navigator.userAgent.indexOf("Edg") != -1);
   22. // Blink engine detection
   23. var isBlink = (isChrome || isOpera) && !!window.CSS;

   26. var output = 'Detecting browsers by ducktyping:<hr>';
   27. output += 'isFirefox: ' + isFirefox + '<br>';
   28. output += 'isChrome: ' + isChrome + '<br>';
   29. output += 'isSafari: ' + isSafari + '<br>';
   30. output += 'isOpera: ' + isOpera + '<br>';
   31. output += 'isIE: ' + isIE + '<br>';
   32. output += 'isEdge: ' + isEdge + '<br>';
   33. output += 'isEdgeChromium: ' + isEdgeChromium + '<br>';
   34. output += 'isBlink: ' + isBlink + '<br>';
   35. document.body.innerHTML = output;

   ## Analysis of reliability

   The [previous method](https://stackoverflow.com/revisions/9851769/1) depended on properties of the rendering engine ([-moz-box-sizing](https://developer.mozilla.org/En/CSS/Box-sizing#Browser_compatibility) and -webkit-transform) to detect the browser. These prefixes will eventually be dropped, so to make detection even more robust, I switched to browser-specific characteristics:

   * Internet Explorer: JScript's [Conditional compilation](https://msdn.microsoft.com/en-us/library/8ka90k2e(v=vs.94).aspx) (up until IE9) and [document.documentMode](https://msdn.microsoft.com/en-us/library/ie/cc196988%28v=vs.85%29.aspx).
   * Edge: In Trident and Edge browsers, Microsoft's implementation exposes the StyleMedia constructor. Excluding Trident leaves us with Edge.
   * Edge (based on chromium): The user agent include the value "Edg/[version]" at the end (ex: "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.16 Safari/537.36 **Edg/80.0.361.9**").
   * Firefox: Firefox's API to install add-ons: [InstallTrigger](https://developer.mozilla.org/en-US/docs/XPInstall_API_Reference/InstallTrigger_Object)
   * Chrome: The global chrome object, containing several properties including a documented [chrome.webstore](https://developer.chrome.com/extensions/webstore) object.
   * Update 3 chrome.webstore is deprecated and undefined in recent versions
   * Safari: A unique naming pattern in its naming of constructors. This is the least durable method of all listed properties and guess what? In Safari 9.1.3 it was fixed. So we are checking against SafariRemoteNotification, which was introduced after version 7.1, to cover all Safaris from 3.0 and upwards.
   * Opera: window.opera has existed for years, but [will be dropped](https://dev.opera.com/blog/300-million-users-and-move-to-webkit/) when Opera replaces its engine with Blink + V8 (used by Chromium).
   * Update 1: [Opera 15 has been released](http://blogs.opera.com/desktop/2013/05/opera-next-15-0-released/), its UA string looks like Chrome, but with the addition of "OPR". In this version the chrome object is defined (but chrome.webstore isn't). Since Opera tries hard to clone Chrome, I use user agent sniffing for this purpose.
   * Update 2: !!window.opr && opr.addons can be used to detect [Opera 20+](https://dev.opera.com/extensions/addons.html) (evergreen).
   * Blink: CSS.supports() [was introduced in Blink](http://caniuse.com/#feat=css-supports-api) once Google switched on Chrome 28. It's of course, the same Blink used in Opera.

   ## Successfully tested in:

   * Firefox 0.8 - 61
   * Chrome 1.0 - 71
   * Opera 8.0 - 34
   * Safari 3.0 - 10
   * IE 6 - 11
   * Edge - 20-42
   * Edge Dev - 80.0.361.9

   [↑](#footnote-ref-1)
2. <https://stackoverflow.com/a/59780781> [↑](#footnote-ref-2)
3. <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-3)
4. <https://www.npmjs.com/package/core-js> [↑](#footnote-ref-4)
5. <https://eslint.vuejs.org/> [↑](#footnote-ref-5)
6. <https://lmiller1990.github.io/vue-testing-handbook/v3/> [↑](#footnote-ref-6)
7. <https://jestjs.io/en/> [↑](#footnote-ref-7)
8. <https://github.com/vuejs/vue-cli/tree/dev/packages/%40vue/cli-plugin-unit-jest> [↑](#footnote-ref-8)
9. “*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-9)
10. 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-10)
11. Se non usassi i servizi di memorizzazione dei dati offerti da *Google*, allora userei dei database fisicamente diversi per memorizzare le informazioni di autenticazione da quelli usati per memorizzare le informazioni degli utenti, per sicurezza. [↑](#footnote-ref-11)
12. La chiave pubblica per la verifica della firma emessa dal server sui token potrebbe essere anche inclusa nel *payload* dei *token JWT*. [↑](#footnote-ref-12)
13. 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-13)