

Corso di Programmazione Web e Mobile
A.A. 2022-2023

WizardGuard

<https://wizardguard.org>

Michele Castoldi – 893291
Alessandro Fontana – 20540A



Autori: Michele Castoldi, Alessandro Fontana
Ultima modifica: 9 novembre 2023 – versione 1.0.0
Prima modifica: 10 ottobre 2023

Indice

1	Introduzione	2
1.1	Breve analisi dei requisiti	2
1.1.1	Destinatari	2
1.1.2	Motivazione	2
1.1.3	Modello di valore	2
1.2	Flusso dei dati	3
1.3	Aspetti tecnologici	3
1.4	Infrastruttura	3
1.5	Funzionamento del sistema di cifratura	4
2	Interfacce	5
2.1	Form di registrazione	5
2.2	Form di accesso	5
2.3	Cassaforte bloccata	6
2.4	Cassaforte	6
2.5	Elemento della cassaforte	7
2.6	Sessioni	7
2.7	Generatore di password	8
2.8	Profilo utente	8
2.9	Template email di benvenuto	9
2.10	Template email di eliminazione utente	9
3	Architettura	10
3.1	Diagramma dell'ordine gerarchico delle risorse	10
3.2	Diagramma di flusso	10
3.2.1	Backend	10
3.3	Descrizione delle risorse	10
3.4	Autenticazione	12
3.5	Database e modelli	12
3.6	Diagramma di flusso app mobile	14
4	Codice	15
4.1	Struttura del progetto	15
4.1.1	Frontend	15
4.1.2	Backend	15
4.2	Backend	16
4.3	Frontend	18
5	Conclusione	21
5.1	Ringraziamenti e crediti	21
	Bibliografia	22

1. Introduzione

WizardGuard è un servizio multiplatforma per la conservazione sicura e relativa gestione di login (username, password e MFA), carte di pagamento e note.

Il focus principale del progetto è cercare di garantire un buon livello di sicurezza all'utente finale, per farlo ci siamo basati sul sistema di crittografia end-to-end utilizzato da Bitwarden Inc., il noto password manager open source [1].

Le tecnologie utilizzate sono: Node.js per il backend, HTML5, CSS e Javascript per il frontend, MongoDB per la persistenza dei dati e Apache Cordova per l'applicazione Android.

1.1 Breve analisi dei requisiti

1.1.1 Destinatari

Il target di riferimento è rappresentato da chiunque abbia la necessità di organizzare le proprie credenziali di accesso in maniera sicura, centralizzata e accessibile, pertanto la forbice d'età risulta molto ampia, non essendo oltretutto richieste competenze tecniche avanzate.

L'attuale modello implementativo non si rivolge all'ambito aziendale, in cui è richiesta una gestione gerarchica degli utenti e la possibile condivisione (con regole e vincoli) delle credenziali immagazzinate.

Per facilitare l'esperienza d'uso degli utenti, le funzionalità sono espresse con una interfaccia visiva intuitiva, corredata da messaggi di testo guidati e indicazioni d'uso. Inoltre vi è stata una scelta mirata nei colori e nelle regole d'uso degli stessi (es. bottoni di conferma, eliminazione) per garantire coerenza ed evitare confusione.

1.1.2 Motivazione

Le motivazioni che possono spingere un utente ad utilizzare l'applicazione possono essere molteplici:

- **organizzative:** considerando la quantità di credenziali che oggi anche un utente base del web deve avere, l'organizzazione in maniera puntuale e centralizzata nonché accessibile da dispositivi differenti;
- **sicurezza informatica:** un utente con una spiccata sensibilità per la sicurezza online e l'importanza di una corretta gestione di credenziali e dati personali;
- **psicologiche:** la consapevolezza di demandare la scelta (rispettando i requisiti richiesti) e la conservazione delle password ad un applicativo crittografato e sicuro rende l'utente più tranquillo nell'approcciarsi alla navigazione web.

Il livello di motivazione è attivo, in quanto l'uso di tale applicazione è dettato da un'esigenza/volontà dell'utente. Rifacendoci al modello di Bates identifichiamo una tipologia di ricerca mirata delle informazioni (attiva e diretta).

1.1.3 Modello di valore

L'applicazione e i servizi ad essa collegati sono erogati in modalità gratuita agli utenti.

Si pensano modelli di business che potranno essere supportati in futuro:

- **Freemium (ambito utente privato):** l'applicativo potrà prevedere l'introduzione di piani d'uso (uno gratuito e uno a pagamento) con funzionalità differenti. In particolare introducendo nel piano gratuito limiti nelle credenziali salvabili nella cassaforte digitale, superati nel piano a pagamento. In più si prevede l'esclusività di alcune funzionalità per il solo piano a pagamento (es. report periodici di verifica di credenziali e carte di pagamento compromesse, condivisione sicura degli elementi ed importazione/esportazione della propria cassaforte);

- **Affiliazione (ambito utente privato):** l'utente di qualsiasi piano avrà la possibilità di condividere un link di affiliazione personale che potrà essere utilizzato da nuovi utenti in fase di registrazione. Nel caso in cui venga sottoscritto il piano a pagamento tramite il link di affiliazione, entrambi gli utenti coinvolti riceveranno dei benefit. L'utente affiliante con piano gratuito riceve la possibilità di fare l'upgrade al piano a pagamento per un periodo limitato di tempo. L'utente affiliante con piano a pagamento e l'utente affiliato, ricevono uno sconto sul rinnovo del piano;
- **Sottoscrizione (ambito aziendale):** si prevede l'implementazione delle funzionalità richieste in ambito aziendale (gestione gerarchica degli utenti e la possibile condivisione, con regole e vincoli, delle credenziali immagazzinate) sottoscrivibili tramite vari piani a pagamento dipendenti dal numero di utenti amministrati.

1.2 Flusso dei dati

Il flusso dei dati risulta generato dagli utenti che ne hanno completa autonomia nella gestione. Per garantire l'accesso da più dispositivi, i dati dell'utente vengono archiviati in modo criptato e inaccessibile, se non dall'utente stesso, in un database. Considerato il modello di lavoro dell'applicativo, il valore dello stesso è costituito dal numero di utenti attivi, rispetto ai dati a loro collegati.

Sui dati verranno applicati i concetti di privacy e security by design (solo le informazioni necessarie verranno salvate sul database) e by default (ogni utente ha accesso alle sole informazioni da lui inserite).

1.3 Aspetti tecnologici

L'applicativo si compone di 3 elementi principali:

- **frontend:** l'interfaccia, condivisa tra sito web e applicazione mobile, è scritta in HTML5 e CSS con l'ausilio del framework Bootstrap[3]. La logica applicativa è sviluppata in Javascript mediante l'utilizzo di jQuery[4]. Le dipendenze sono incluse tramite NPM[5] e compilate mediante Browserify[6], permettendo quindi una gestione più semplice e rapida degli aggiornamenti.
- **backend:** le REST API sono sviluppate da un server NodeJS[7] con framework Express[8]. Le API permettono la gestione degli utenti, delle sessioni ad essi associate e della cassaforte personale mediante la messa a disposizione di una serie di endpoint di chiamata. Per gestire le interazioni con il database si fa uso del framework mongoose[9]. Le dipendenze sono gestite tramite NPM.
- **database:** database NoSQL MongoDB[10] per l'archiviazione persistente dei dati degli utenti.

1.4 Infrastruttura

Si descrivono le scelte infrastrutturali messe in opera per la pubblicazione dell'applicativo.

Per ricondurre il progetto ad un prodotto reale abbiamo registrato un dominio (wizardguard.org) così da rendere accessibile il client web.

Per il frontend web si è provveduto a sfruttare il servizio cloud Render[11], che mette a disposizione in maniera gratuita la pubblicazione di uno static site tramite un server http Node.js. Sfruttando le impostazioni messe a disposizione da Render si è provveduto a configurare, con specifici record DNS, la base uri del dominio di progetto.

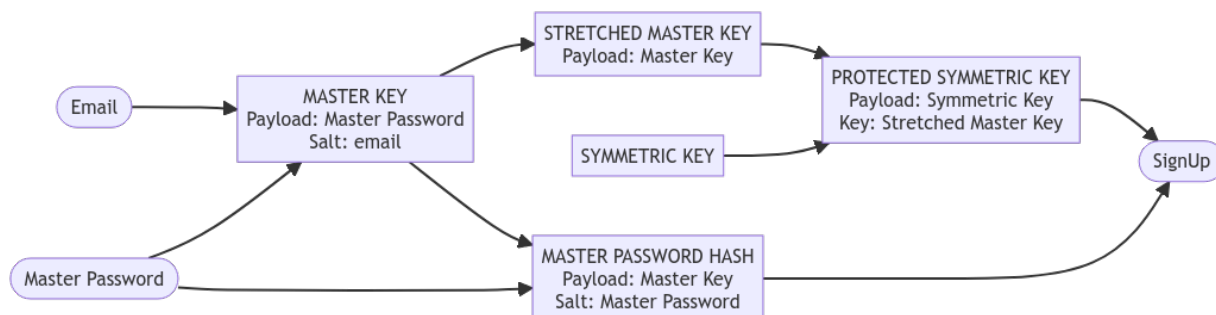
Simile soluzione è stata adottata per il backend, il quale è stato reso accessibile su un dominio di terzo livello dedicato (api.wizardguard.com).

Il database è stato pubblicato sul servizio DBaaS Atlas offerto da MongoDB Inc. e risulta accessibile tramite uno specifico indirizzo di host e credenziali di accesso (username e password).

Il codice di frontend e backend è pubblicato su due differenti e dedicati repository Git[12], ospitati su GitHub[13], da cui Render acquisisce automaticamente i sorgenti da pubblicare.

Per l'invio delle mail agli utenti è stato configurato opportunamente un account Zoho[14], che offre un server SMTP gratuito, associabile al proprio dominio.

1.5 Funzionamento del sistema di cifratura



Il sistema di cifratura opera interamente lato client e tutti i dati trasmessi da e verso i server sono crittografati end-to-end, anche qualora l'utente effettui l'accesso da un nuovo dispositivo. In questo modo, anche ipotizzando una compromissione dei server, i dati rimarranno comunque riservati. La sicurezza degli stessi è quindi demandata alla robustezza della password scelta dall'utente in fase di creazione dell'account (la quale richiede determinati requisiti di complessità).

È possibile schematizzare la fase di registrazione nei seguenti punti:

- Viene generata la *Master Key* utilizzando la *Master Password* come payload e l'indirizzo email come salt;
- Viene creata la *Stretched Master Key* utilizzando la *Master Key* come payload;
- Viene generata la *Symmetric Key* a 512-bit utilizzando una funzione randomica sicura. Questa è la chiave di cifratura dei dati dell'utente;
- Viene creata la *Protected Symmetric Key* cifrando la *Symmetric Key* mediante la *Stretched Master Key* e un vettore di inizializzazione casuale. Questa chiave verrà condivisa al server e ai dispositivi in cui l'utente effettua l'accesso.

Poiché la *Master Password* deve essere mantenuta segreta, in fase di registrazione viene inviato al backend l'hash della stessa (oltre che la *Protected Symmetric Key* chiamata nell'uso *Magic Key*) e questo provvederà a conservare il suo ulteriore hash insieme ad un salt. L'hash della *Master Password*, generato dal client, può essere quindi considerato come il criterio di identificazione dell'utente, mantenendo al contempo la riservatezza dei suoi dati cifrati.

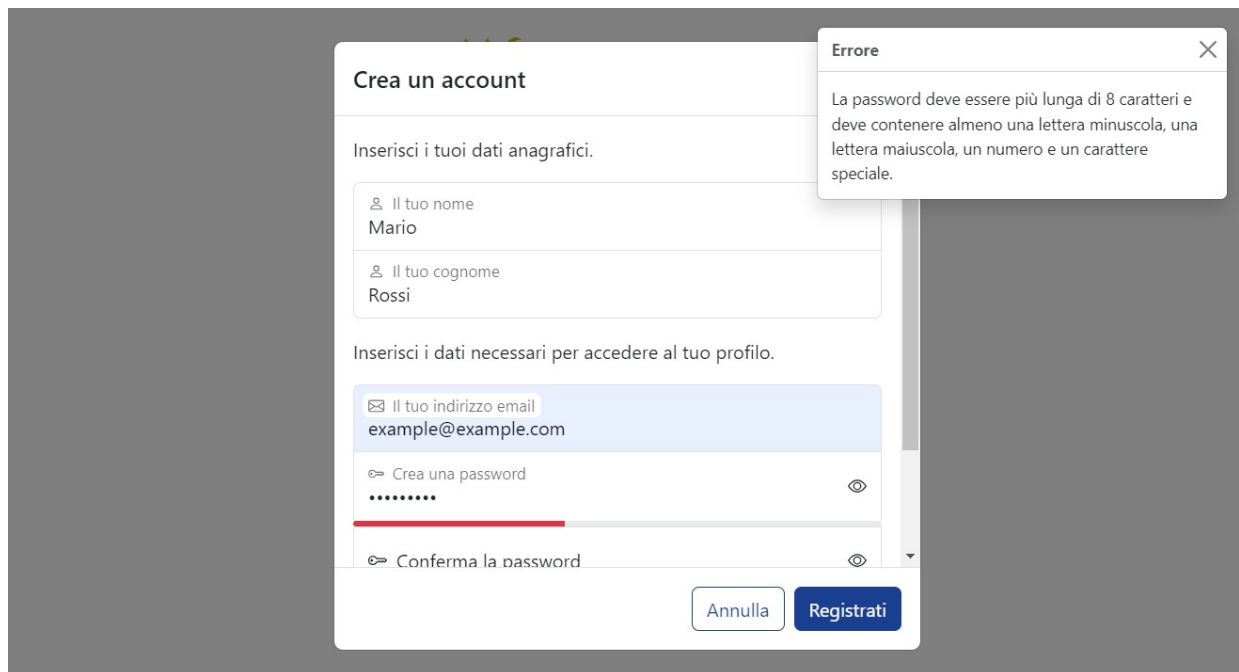
A seguito di un accesso andato a buon fine il client scarica la *Protected Symmetric Key*, e, tramite un procedimento analogo a quello della registrazione, una volta inseriti indirizzo email e *Master Password*, decifra la *Symmetric Key* tramite la *Stretched Master Key* (nuovamente calcolata) con cui è possibile decifrare la cassaforte.

La procedura di cambio *Master Password*, anch'essa effettuata in locale, prevede una nuova cifratura della stessa *Symmetric Key* (essendo cambiato il payload iniziale) e la condivisione al server della nuova *Protected Symmetric Key*.

2. Interfacce

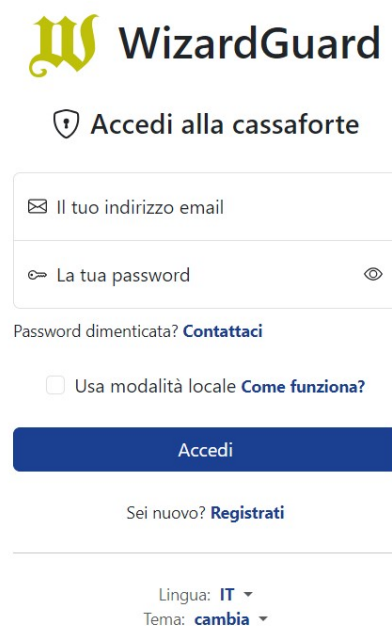
WizardGuard è usufruibile in tema chiaro/scuro e multilingua (attualmente italiano/inglese).

2.1 Form di registrazione




The screenshot shows the 'Crea un account' (Create an account) form. It has two sections: 'Inserisci i tuoi dati anagrafici.' (Enter your personal data) and 'Inserisci i dati necessari per accedere al tuo profilo.' (Enter the data needed to access your profile). The first section contains fields for 'Il tuo nome' (Your name) with the value 'Mario' and 'Il tuo cognome' (Your surname) with the value 'Rossi'. The second section contains fields for 'Il tuo indirizzo email' (Your email address) with the value 'example@example.com', 'Crea una password' (Create a password) with masked characters '.....', and 'Conferma la password' (Confirm the password). At the bottom right are 'Annulla' (Cancel) and 'Registrati' (Register) buttons. An 'Errore' (Error) modal is displayed, stating: 'La password deve essere più lunga di 8 caratteri e deve contenere almeno una lettera minuscola, una lettera maiuscola, un numero e un carattere speciale.' (The password must be longer than 8 characters and must contain at least one lowercase letter, one uppercase letter, a number, and a special character).


2.2 Form di accesso




The screenshot shows the WizardGuard login interface. At the top is the WizardGuard logo. Below it is the heading 'Accedi alla cassaforte' (Log in to the safe) with a shield icon. The login form consists of two input fields: 'Il tuo indirizzo email' (Your email address) and 'La tua password' (Your password). Below the password field is a link 'Password dimenticata? Contattaci' (Forgot password? Contact us). There is a checkbox for 'Usa modalità locale Come funziona?' (Use local mode How it works?). A large blue 'Accedi' (Log in) button is positioned below the form. At the bottom, there is a link 'Sei nuovo? Registrati' (Are you new? Register). At the very bottom, there are links for 'Lingua: IT' (Language: IT) and 'Tema: cambia' (Theme: change).

2.3 Cassaforte bloccata

 **La cassaforte è bloccata**

 La tua password




Sblocca


Password dimenticata? [Esci](#)

Lingua: **IT** ▼
Tema: **cambia** ▼


2.4 Cassaforte



Cassaforte Generatore Sessioni

IT ▼ Ciao, Mario 

Filtra la cassaforte




Tutto

🔑 Password










💳 Carte di pagamento

📄 Note sicure

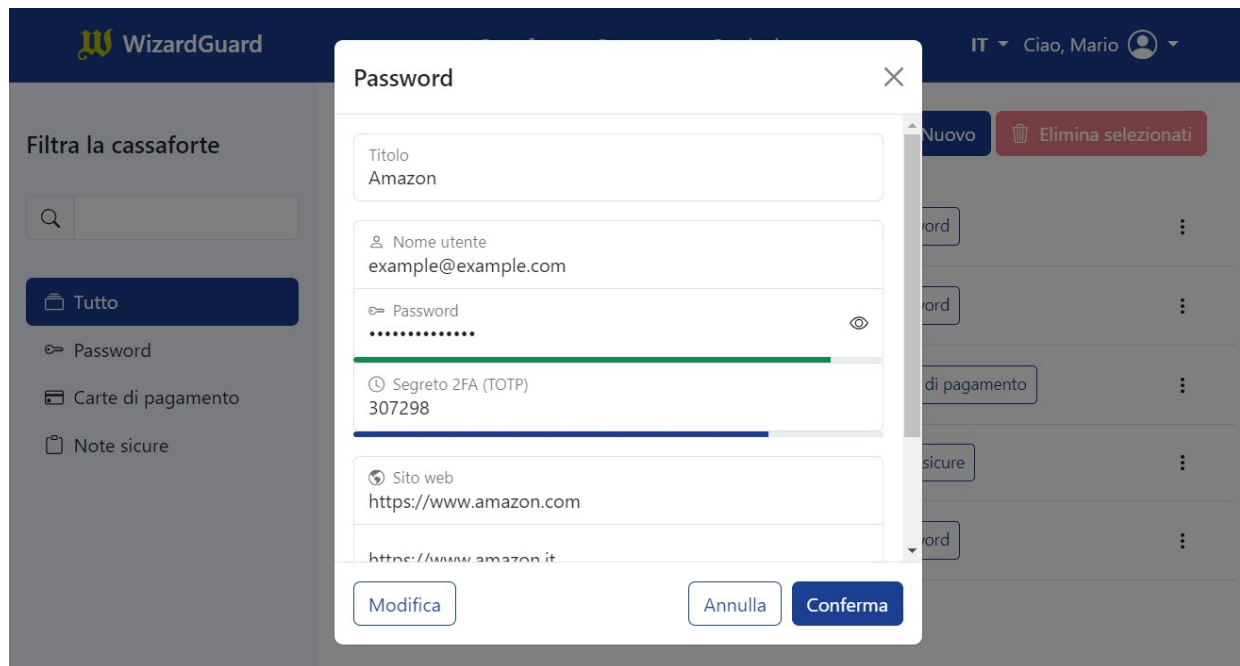
 **La tua cassaforte**

+ Nuovo

🗑 Elimina selezionati

<input type="checkbox"/>	 Amazon example@example.com	<div> Password</div>	⋮
<input type="checkbox"/>	 PayPal example@example.com	<div> Password</div>	⋮
<input type="checkbox"/>	 Banca Nazionale Italiana 5325 4212	<div> Carte di pagamento</div>	⋮
<input type="checkbox"/>	Lista della spesa Premi per leggere	<div> Note sicure</div>	⋮
<input type="checkbox"/>	 Google Mario Rossi	<div> Password</div>	⋮

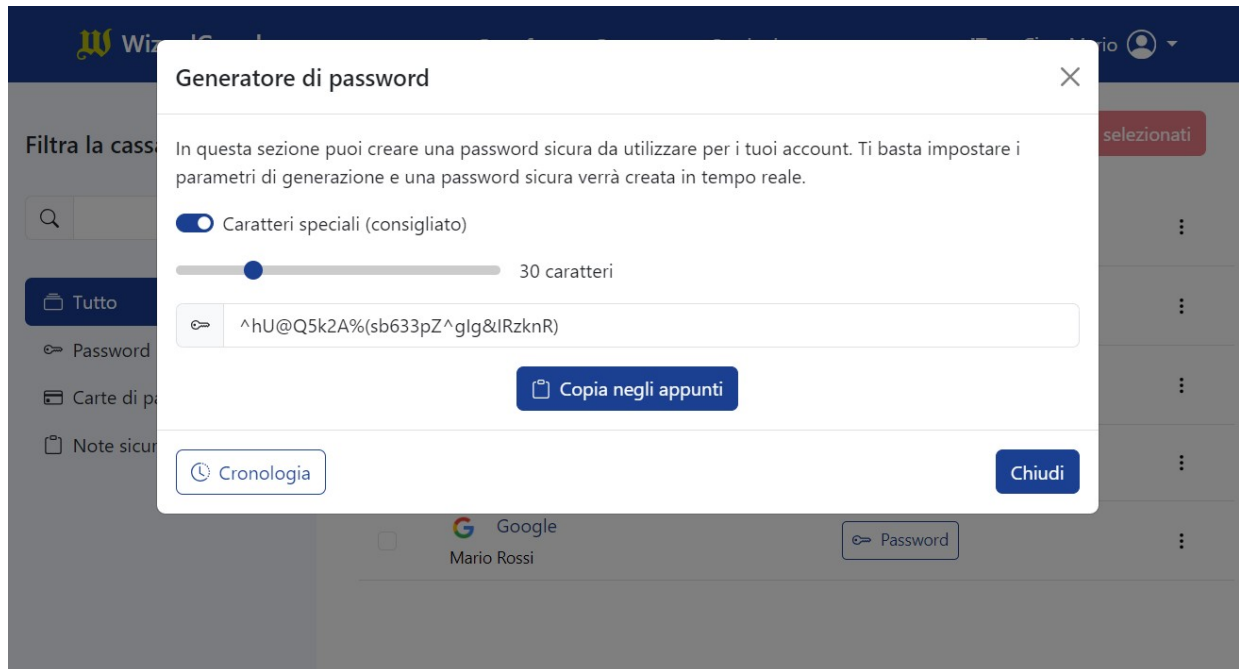
2.5 Elemento della cassaforte



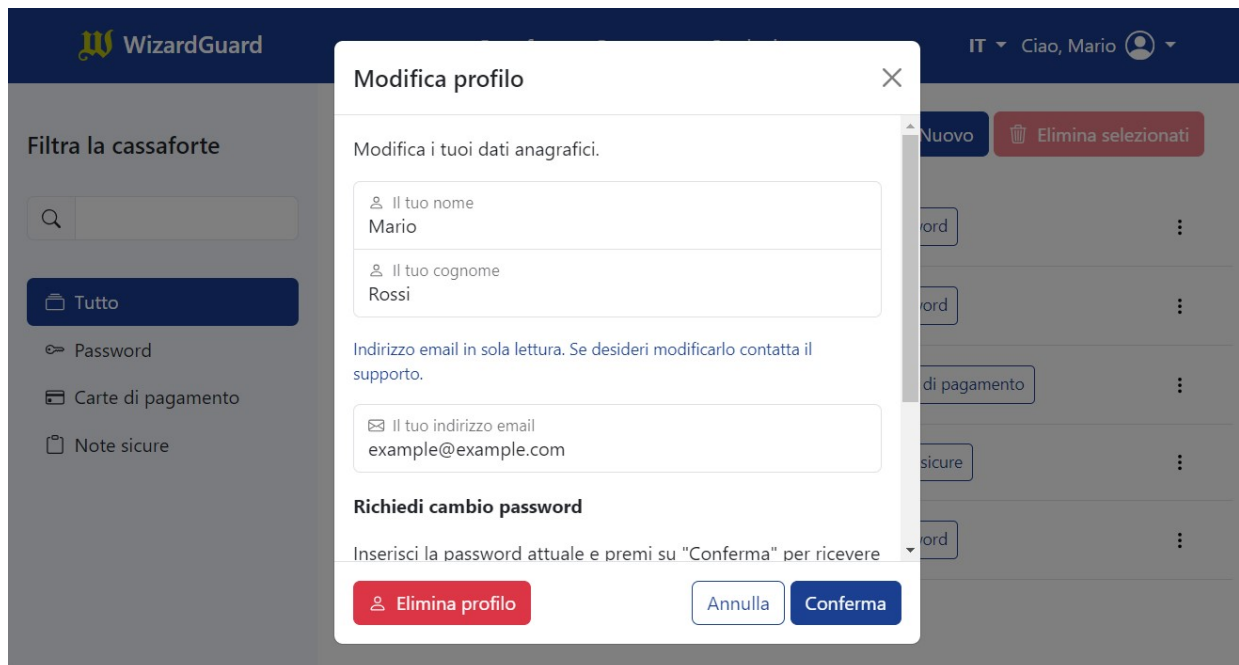
2.6 Sessioni



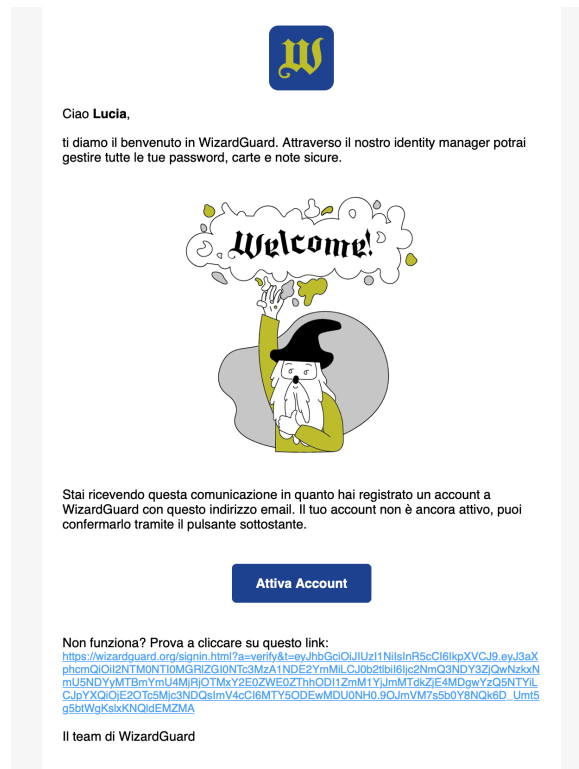
2.7 Generatore di password



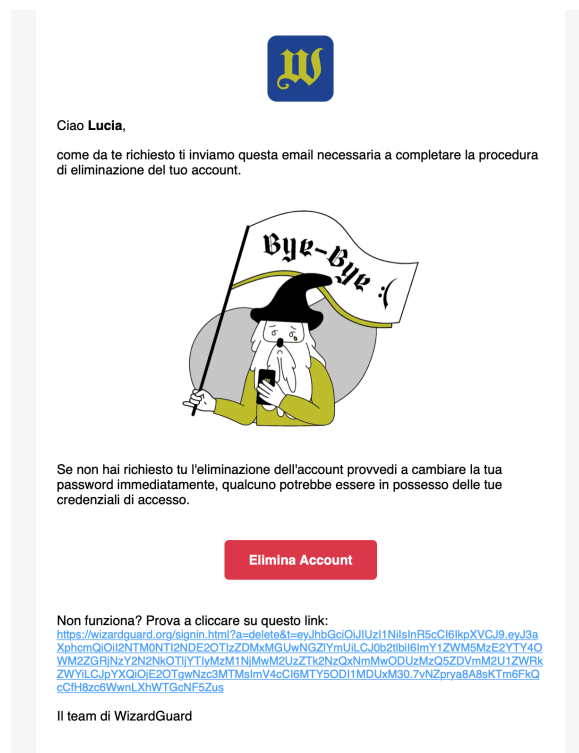
2.8 Profilo utente



2.9 Template email di benvenuto



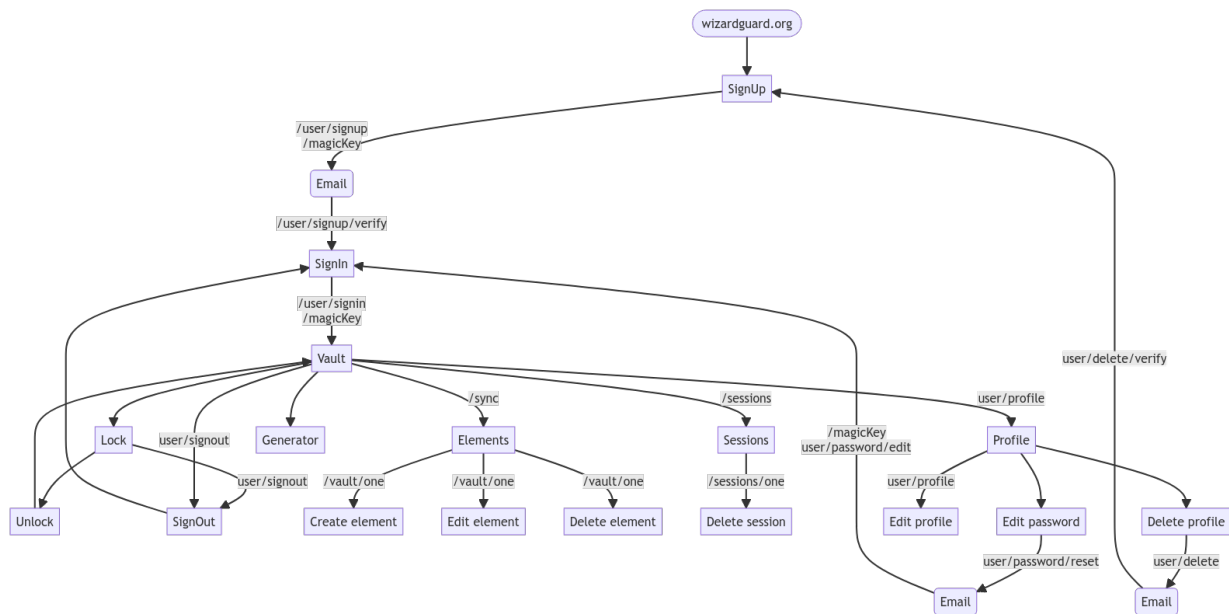
2.10 Template email di eliminazione utente



3. Architettura

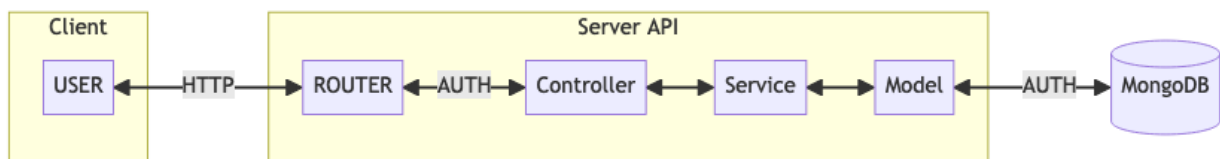
3.1 Diagramma dell'ordine gerarchico delle risorse

Si riportano le funzionalità di frontend accessibili dall'utente e le loro transazioni con le relative chiamate API a backend.



3.2 Diagramma di flusso

3.2.1 Backend



Le richieste al server api arrivano tramite protocollo HTTP Sicuro e vengono smistate dal Router che richiama la funzione dedicata.

Tutti gli endpoint forniscono dati tramite JSON nel body della richiesta. In base alla chiamata effettuata vengono fatti i controlli preliminari previsti (autenticazione e/o validazione dati inviati).

- Il **Controller** gestisce la richiesta e si occupa di comporre la risposta;
- Al **Service** vengono demandate le elaborazioni riferite alla chiamata invocata;
- Il **Model** gestisce le relazioni dirette con la base di dati.

3.3 Descrizione delle risorse

Per la struttura delle risposte fornite dalle API di backend si è scelto di seguire le specifiche JSend[2]. Tutte le risposte rispettano la seguente struttura:

```

1 {
2     status: "success | fail | error",
3     data: {} | message: "..."
4 }

```

In particolare il campo data viene indicato solo con il campo status valorizzato come:

- **success**: contiene i dati inviati (PUT/POST) o richiesti (GET), altrimenti null;
- **fail**: contiene un array di oggetti ognuno dei quali ha come chiave l'elemento della richiesta coinvolto dal fail e come valore, un messaggio in inglese in merito.

```

1 {
2     "name.firstName": "User first name is required"
3 }

```











Nel caso status: "error", viene proposto il campo message, valorizzato con un messaggio di errore, nella lingua associata all'header della richiesta o nel caso la lingua non fosse supportata, in inglese.

Vengono di seguito elencati nel dettaglio tutti gli endpoint delle API di backend.




auth

Endpoint	Auth	Descrizione
POST auth/refreshTokens		Generazione nuovi auth token e modifica sessione



user

Endpoint	Auth	Descrizione
POST user/signup		Registrazione nuovo utente
POST user/signup/verify		Verifica e attivazione nuovo utente
POST user/signin		Accesso utente
POST user/signout		Uscita utente ed eliminazione sessione
GET user/profile		Fornisce i dati utente (nome e email)
PUT user/profile		Modifica i dati utente (nome e email)
POST user/delete		Invio email per eliminazione account
DELETE user/delete/verify		Verifica ed eliminazione utente
POST user/password/reset		Invio email per cambio password
PUT user/password/edit		Cambio password








sessions

Endpoint	Auth	Descrizione
GET sessions		Fornisce l'elenco delle sessioni (indicando quella corrente)
DELETE sessions		Elimina le sessioni (opzione elimina tutte o eccetto sessione corrente)
DELETE sessions/one		Elimina la sessione indicata

magicKey

Endpoint	Auth	Descrizione
POST magicKey		Aggiunge/modifica la magic key
GET magicKey		Fornisce la magic key

vault

Endpoint	Auth	Descrizione
GET vault		Fornisce la cassaforte utente
POST vault		Aggiunge elementi alla cassaforte utente
PUT vault		Modifica elementi della cassaforte utente
DELETE vault		Elimina la cassaforte utente
POST vault/one		Aggiunge un elemento alla cassaforte utente
PUT vault/one		Modifica un elemento della cassaforte utente
DELETE vault/one		Elimina un elemento della cassaforte utente

3.4 Autenticazione

L'accesso agli endpoint delle API di backend è gestito mediante autenticazione tramite JWT[15]. Alcuni endpoint non autenticati (*user/signin*, *user/signup*, *auth/refreshTokens*) permettono di ricevere nella risposta *Access Token* e *Refresh Token*, da utilizzare per le chiamate autenticate. I token contengono dei dati crittografati, utili per le richieste, e sono soggetti ad una scadenza (10 minuti per l'*Access Token* e 14 giorni per il *Refresh Token*).

Una volta ricevuti i token, il client li salva e utilizza l'*Access Token* (inserendolo opportunamente nell'header della richiesta) per autenticarsi. Una volta che l'*Access Token* scade, le richieste saranno rifiutate perchè non autorizzate, a questo punto tramite un endpoint specifico (*auth/refreshTokens*) sarà possibile, fornendo il *Refresh Token*, ricevere una nuova coppia di token.

Per tenere traccia delle sessioni attive, ad un nuovo accesso dell'utente viene creata una nuova sessione utente sul database, a cui è associato il *Refresh Token*. La verifica dei token, oltre che crittografica e di validità di tempo, è di consistenza vagliando la lista delle sessioni utente sul database.

3.5 Database e modelli

La struttura del database è stata modellata realizzando una singola collection riferita all'utente e alle sue attività. In particolare oltre ai dettagli dell'utente sono archiviate le sessioni personali attive e le stringhe criptate relative alla sua cassaforte. Di seguito viene proposto per esteso, il modello della collection, così come viene validato dal database.

```
1 "$jsonSchema": {
2   "bsonType": "object",
3   "required": ["user", "magicKey", "sessions", "vault"],
4   "properties": {
5     "user": {
6       "bsonType": "object",
7       "required": ["name", "email", "masterPassword", "isActive", "validationToken",
8         ↪ "isEnabled", "creationTimestamp"],
9       "properties": {
10        "name": {
```

```

10         "bsonType": "object",
11         "required": ["firstName", "lastName"],
12         "properties": {
13             "firstName": { "bsonType": "string" },
14             "lastName": { "bsonType": "string" }
15         }
16     },
17     "email": {
18         "bsonType": "string"
19     },
20     "masterPassword": {
21         "bsonType": "object",
22         "required": ["value", "salt"],
23         "properties": {
24             "value": { "bsonType": "string" },
25             "salt": { "bsonType": "string" }
26         }
27     },
28     "isActive": { "bsonType": "bool" },
29     "validationToken": { "bsonType": "string" },
30     "isEnabled": { "bsonType": "bool" },
31     "creationTimestamp": { "bsonType": "date" }
32 },
33 },
34 "magicKey": { "bsonType": "string" },
35 "sessions": {
36     "bsonType": "array",
37     "items": {
38         "bsonType": "object",
39         "required": ["id", "location", "userAgent", "refreshToken"],
40         "properties": {
41             "id": { "bsonType": "objectId" },
42             "location": {
43                 "bsonType": "string",
44                 "minLength": 2,
45                 "maxLength": 2
46             },
47             "userAgent": {
48                 "bsonType": "object",
49                 "required": ["device", "client", "type"],
50                 "properties": {
51                     "device": { "bsonType": "string" },
52                     "client": { "bsonType": "string" },
53                     "type": { "enum": ["web", "mobile", "unknown"] }
54                 }
55             },
56             "refreshToken": {
57                 "bsonType": "object",
58                 "required": ["value", "expiration"],
59                 "properties": {
60                     "value": { "bsonType": "string" },
61                     "expiration": { "bsonType": "date" }
62                 }
63             }
64         }
65     }
66 },

```

```

67     "vault": {
68         "bsonType": "array",
69         "items": {
70             "bsonType": "object",
71             "required": ["id", "value", "updatedAt"],
72             "properties": {
73                 "id": { "bsonType": "objectId" },
74                 "value": { "bsonType": "string" },
75                 "updatedAt": { "bsonType": "string" }
76             }
77         }
78     }
79 }
80 }

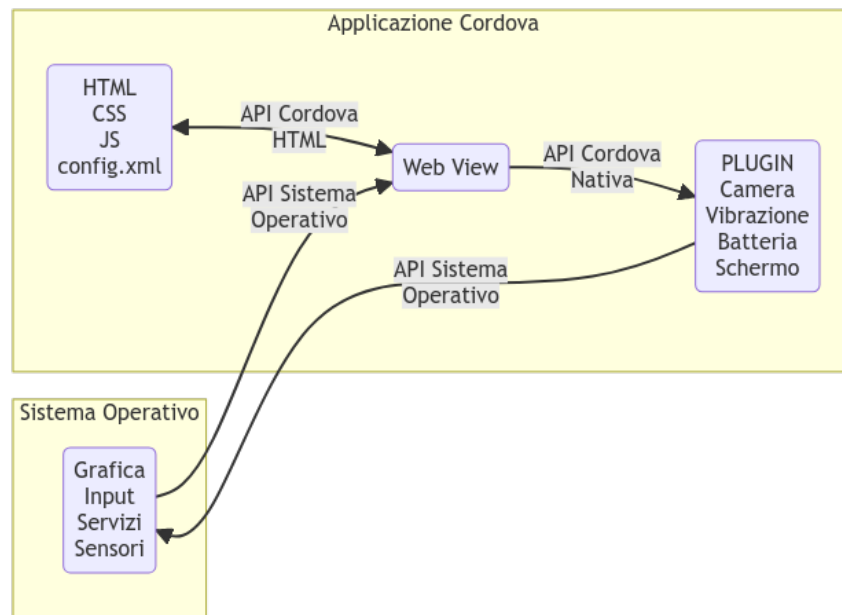
```

3.6 Diagramma di flusso app mobile

Attraverso il framework Apache Cordova è stato possibile utilizzare l'interfaccia grafica e la logica principale dell'applicazione precedentemente sviluppata, ottimizzando i tempi di sviluppo e uniformando l'usabilità per a livello utente.

Cordova permette inoltre integrazioni con il sistema operativo tramite plugin appositamente sviluppati. Tali plugin non sono stati tuttavia utilizzati in quanto non risulta necessario accedere a specifiche funzioni di sistema.

Per la specifica versione Android sono stati definiti icona e splash screen nel file config.xml, è stata inoltre creata la classe WizardGuardAutofillService.java, registrata come servizio in AndroidManifest.xml, demandata ad utilizzare l'API di sistema di AutoFill necessaria a identificare i campi password e a proporre l'apertura dell'app in tali contesti.



4. Codice

4.1 Struttura del progetto

Di seguito i riportano le strutture dei due repository git di produzione di frontend e backen.

4.1.1 Frontend

Nella cartella principale del progetto sono presenti:

- cartella **css**: contenente i fogli di stile;
- cartella **img**: contenente le immagini statiche;
- cartella **js**: contenente le logiche applicative javascript;
- **cordova.js**: file popolato da Cordova alla compilazione dell'applicazione;
- **index.html**: file di redirect alla signin;
- **signin.html**: pagina di registrazione ed accesso utente;
- **unlock.html**: pagina di blocco/sblocco della cassaforte;
- **vault.html**: pagina della cassaforte (autenticata).

4.1.2 Backend

Nella cartella principale del progetto sono presenti:

- cartella **src**: contenente tutti i file Node.js del backend;
- cartella **tests**: cartella vuota per l'implementazione futura degli unit test del backend;
- **package.json**: file di configurazione del progetto NPM, com indicate le dipendenze di pacchetti utilizzate.

Si precisa ora la struttura della cartella **src**:

- **app.js**: file principale dell'applicazione dove:
 - viene eseguita la connessione al dabatase MongoDB;
 - viene creato e configurato il server Express;
 - vengono specificati i router degli enpoint delle api;
 - viene gestita la risposta in caso di error, fail o eccezioni;
 - viene avviato il server, in ascolto sulla porta 3434 (modificata da Render a 1000).
- **config.js**: file javascript contentente alcune variabili di impostazione;
- cartella **controllers**;
- cartella **services**;
- cartella **routes**;
- cartella **data**: contenente alcuni file statici di configurazioone (codici di errore e template mail multilingua).

Nel repository di produzione del backend non è presente il file **.env**, in quanto le variabili di ambiente utilizzate (credenziali database e chiavi di cifratura token) sono indicate direttamente nella configurazione di progetto di Render.

4.2 Backend

Si riportano le funzioni di verifica dei token di autenticazione, in particolare la funzione `authenticateToken` viene chiamata prima dell'esecuzione del codice specifico dell'endpoint autenticato chiamato, e dell'eventuale validazione del body della richiesta. La verifica, come spiegato in precedenza, viene effettuata sia tramite il controller dei token JWT, quindi verificando la corretta decifratura dei dati contenuti nel token e la sua validità temporale, che verificando l'esistenza della sessione corrispondente tra quelle associate all'utente nel database. La funzione `verifyControlToken` riguarda invece la verifica di un token artificiale, creato sempre tramite JWT, con una chiave di cifratura dedicata, utilizzato come paramentro nel link inviato all'utente via mail, nelle operazioni di verifica utenza, cambio password ed eliminazione utenza.

```
1 const TokenController = require("../token");
2 const Exception = require("../exception");
3 const SessionModel = require("../models/session");
4 const UserModel = require("../models/user");
5 const UserController = require("../controllers/user");
6
7 exports.authenticateToken = async (req, res, next) => {
8   const authHeader = req.headers["authorization"];
9   if (typeof authHeader !== 'undefined') {
10     const token = authHeader.split(" ")[1];
11     if (token !== null) {
12       try {
13         const tokenData = TokenController.verifyAccessToken(token);
14         if (!(await SessionModel.getRefreshToken(tokenData.wizard,
15           ↪ tokenData.session)))
16           return next(new Exception(106));
17         req.auth = tokenData;
18         return next();
19       } catch (err) {
20         next(new Exception(106));
21       }
22     }
23     next(new Exception(106));
24   }
25
26 exports.verifyControlToken = async (controlToken) => {
27   const tokenData = TokenController.verifyControlToken(controlToken);
28   if (!tokenData)
29     return false;
30   const verifyFlag = await UserModel.verifyUserTokenById(tokenData.wizard,
31     ↪ tokenData.token);
32   const updateFlag = await UserController.updateUserToken(tokenData.wizard);
33   return verifyFlag && updateFlag ? tokenData.wizard : false;
34 }
```

Di seguito vengono mostrate due funzioni associate a due differenti endpoint, il primo `user/signin` (non autenticato) permette di effettuare l'accesso, inserendo i dati di autenticazione utente nel body della richiesta. La risposta restituisce il nome dell'utente e i token necessari alle successive chiamate autenticate. Nella funzione, dopo aver verificato i dati di accesso dell'utente, viene creata una nuova sessione associata al Refresh Token generato e viene effettuato un controllo sulla localizzazione dell'ip della richiesta utente e, nel caso sia riferito ad una nazione inedita per l'utente, lo stesso viene avvisato con una email.

```
1 router.post('/signin', userSignInDataValidateChain, async (req, res, next) => {
```

```

2   try {
3     if (!(fail = validationResult(req)).isEmpty())
4       return next(new Fail(fail.array()));
5     const user = await UserController.signin(req.body.email, req.body.password);
6     if (!user)
7       return next(new Exception(107));
8     if (!user.isActive)
9       return next(new Exception(115));
10    const sessionId = SessionController.generateSessionId();
11    const tokens = AuthController.generateTokens(user.wizard, req.body.email,
    ↪   sessionId);
12    await SessionController.pushSession(user.wizard, sessionId, req.ip,
    ↪   req.useragent, tokens.refreshToken);
13    const data = ResponseController.concatTwo({
14      name: user.name
15    }, tokens);
16    response.setSuccess(data);
17    const location = getLocationByIp(req.ip);
18    if (await SessionModel.isNewCountry(user.wizard, location.iso)) {
19      lang = req.acceptsLanguages(acceptedLanguages);
20      await EmailController.sendNewLocationEmail(user.name, req.body.email, lang,
    ↪   user.wizard, user.validationToken, location.ext);
21    }
22    res.status(response.getHttpCode()).json(response.getResponse());
23  } catch (error) {
24    next(error);
25  }
26  });

```

Infine si mostrano due funzioni del SessionModel, in particolare la prima permette di inserire una nuova sessione sul database, mentre la seconda elimina tutte le sessioni utente presenti (con l'eventualità di mantenere quella corrente specificandone l'id come paramentro).

```

1   const WizardModel = require('./wizard');
2
3   exports.pushSession = async (wizard, session, location, userAgent, refreshToken,
    ↪   expiration) => {
4     const sessionData = {
5       id: session,
6       location: location,
7       userAgent: userAgent,
8       refreshToken: {
9         value: refreshToken,
10        expiration: expiration
11      }
12    };
13    await WizardModel.updateOne({
14      "_id": wizard
15    }, {
16      $push: {
17        "sessions": sessionData
18      }
19    });
20  };
21
22  ...

```

```

23
24 exports.clearSessions = async (wizard, id = undefined) => {
25   if (id === undefined)
26     return await WizardModel.updateOne({
27       "_id": wizard
28     }, {
29       $set: {
30         "sessions": []
31       }
32     }) ?? false;
33   return await WizardModel.updateMany({
34     "_id": wizard
35   }, {
36     $pull: {
37       "sessions": {
38         "id": {
39           $ne: id
40         }
41       }
42     }
43   }) ?? false;
44 }

```

4.3 Frontend

Qui viene riportato il funzionamento del web worker demandato all'esecuzione delle funzioni crittografiche. L'utilizzo di un web worker si è reso necessario in quanto queste operazioni, se eseguite nel processo principale, avrebbero bloccato la grafica e le animazioni poiché necessitano di elevata quantità di risorse. Si è deciso di utilizzare un solo web worker (denominato `crypto-worker` in funzione delle sue attività) passando tra i parametri la funzione da eseguire, i parametri da utilizzare e la funzione di callback (con i relativi parametri esterni in quanto non serializzabili). Questo il codice del chiamante:

```

1 var cryptoWorker = new Worker("./js/crypto-worker.js");
2
3 cryptoWorker.addEventListener("message", function(e) {
4   var args = e.data.args;
5   const callback = args[1];
6   const callbackParameters = args[2];
7   const results = args[3];
8   const parameters = callbackParameters.concat(results);
9   window[callback](...parameters);
10 });
11
12 function startCryptoWorker(functionName, functionParameters, callback,
13   ↪ callbackParameters) {
14   cryptoWorker.postMessage({
15     "args": [functionName, functionParameters, callback, callbackParameters]
16   });
17 }

```

Si riporta un frammento significativo di `crypto-worker.js`:

```

1 importScripts("./crypto.js?v202310281736");
2

```

```

3 self.addEventListener("message", function(e) {
4
5     var args = e.data.args;
6     const functionName = args[0];
7     const functionParameters = args[1];
8     const callback = args[2];
9     const callbackParameters = args[3];
10    var results = [];
11
12    if (functionName == "generateMasterPasswordHash") {
13        results[0] = generateMasterPasswordHash(functionParameters[0],
14            ↪ functionParameters[1]);
15    } else if (functionName == "signUp") {
16        results[0] = generateMasterPasswordHash(functionParameters[0],
17            ↪ functionParameters[1]);
18        results[1] = generateProtectedSimmetricKey(functionParameters[0],
19            ↪ functionParameters[1]);
20    }
21
22    self.postMessage({
23        "args": [functionName, callback, callbackParameters, results]
24    });
25 }, false);

```

Di seguito si riporta la funzione di sincronizzazione tra gli elementi locali e quelli remoti precedentemente mostrata tramite diagramma di flusso. Attraverso le funzionalità implementate risulta possibile gestire gli elementi creati su più dispositivi, evitando sovrascritture grazie all'utilizzo di un timestamp relativo all'ultimo aggiornamento.

```

1 function syncVault(remoteElements, remoteElementsTimestamp, remoteElementsValues) {
2     var localElements = [];
3     var createElements = [];
4     var editElements = [];
5     var deleteElements = [];
6
7     // Load local elements
8     for (var i = 0; i < localStorage.length; i++) {
9         if (localStorage.key(i).startsWith("element-")) {
10             var key = localStorage.key(i);
11             key = key.replace("element-", "");
12             localElements.push(key);
13         }
14     }
15
16     // Check remote elements
17     for (var i = 0; i < remoteElements.length; i++) {
18         if (localElements.includes(remoteElements[i])) {
19             var localTimestamp = localStorage.getItem("timestamp-element-" +
20                 ↪ remoteElements[i]);
21             if (!localTimestamp || (localTimestamp < remoteElementsTimestamp[i])) {
22                 localStorage.setItem("element-" + remoteElements[i],
23                     ↪ remoteElementsValues[i]);
24                 localStorage.setItem("timestamp-element-" + remoteElements[i],
25                     ↪ remoteElementsTimestamp[i]);
26             } else if (localTimestamp && localTimestamp != remoteElementsTimestamp[i]) {
27                 if (localStorage.getItem("element-" + remoteElements[i]) == "") {

```

```

24         deleteElements.push(localElements[i]);
25     } else {
26         editElements.push(remoteElements[i]);
27     }
28 }
29 } else {
30     localStorage.setItem("element-" + remoteElements[i], remoteElementsValues[i]);
31     localStorage.setItem("timestamp-element-" + remoteElements[i],
32         ↪ remoteElementsTimestamp[i]);
33 }
34 }
35 // Check local elements
36 for (var i = 0; i < localElements.length; i++) {
37     if (!remoteElements.includes(localElements[i])) {
38         if (!localElements[i].startsWith("local-")) {
39             localStorage.removeItem("element-" + localElements[i]);
40         } else {
41             var value = localStorage.getItem("element-" + localElements[i]);
42             if (value == "") {
43                 deleteElements.push(localElements[i]);
44             } else {
45                 createElements.push(localElements[i]);
46             }
47         }
48     }
49 }
50 createElements = removeDuplicates(createElements);
51 editElements = removeDuplicates(editElements);
52 deleteElements = removeDuplicates(deleteElements);
53
54 createElementsInVault(createElements);
55 updateElementsInVault(editElements);
56 deleteElementsFromVault(deleteElements);
57 }

```

5. Conclusione

Lo sviluppo di questo progetto ci ha permesso di utilizzare alcune delle tecnologie approfondite in teoria, potendole applicare in un contesto reale. Inoltre l'opportunità di svolgere il progetto in gruppo, ci ha anche permesso di utilizzare applicativi di sviluppo condiviso del codice come Git.

Immaginando funzionalità future per il progetto, ipotizziamo la realizzazione di una estensione (per i browser principali), che permetta all'utente la gestione integrata nel browser della sua cassaforte di credenziali, sfruttando l'auto-completamento dei campi di accesso.

5.1 Ringraziamenti e crediti

Si ringrazia Francesca Castoldi per aver realizzato il logo, la palette colori e le illustrazioni per i template email e per la cassaforte vuota.

Bibliografia

- [1] Bitwarden Security Whitepaper, <https://bitwarden.com/help/bitwarden-security-white-paper/>
- [2] JSend, JSON format for application-level communication, <https://github.com/omniti-labs/jsend>
- [3] Bootstrap, <https://getbootstrap.com/docs/5.3/getting-started/introduction/>
- [4] jQuery, <https://api.jquery.com/>
- [5] NPM, <https://docs.npmjs.com/getting-started>
- [6] Browserify, <https://github.com/browserify/browserify#usage>
- [7] Node.js, <https://nodejs.org/en/docs>
- [8] Express, <https://expressjs.com/en/4x/api.html>
- [9] mongoose, <https://mongoosejs.com/docs/index.html>
- [10] MongoDB, <https://www.mongodb.com/docs/manual/>
- [11] Render, <https://render.com>
- [12] Git, <https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup>
- [13] GitHub, <https://github.com>
- [14] Zoho, <https://www.zoho.com/>
- [15] JWT: JSON Web Token, <https://jwt.io/introduction/>