



UNIVERSITÀ
DI TRENTO

Dipartimento di ingegneria e scienza dell'informazione

Progetto:

Vocable

Titolo del documento:

Sviluppo della webapp

Document info:

Document info

Doc. Name	D4-Vocable_Sviluppo	Doc. Number	D4
Description	Il documento contiene una descrizione dello sviluppo della webapp Vocable		

INDICE

INDICE.....	1
Scopo del documento.....	3
APPLICATION IMPLEMENTATION AND DOCUMENTATION.....	5
Struttura del Progetto.....	6
Dipendenze del Progetto.....	9
Modelli nel database.....	10
Modello utente.....	10
Modello utenteStats.....	10
Project APIs.....	12
Diagramma di estrazione delle risorse.....	12
Diagramma delle risorse.....	14
Endpoint per il modello Utente:.....	14
Endpoint per il modello Utentestats:.....	14
Rappresentazione completa degli endpoint:.....	15
Sviluppo API.....	16
API per la gestione del modello Utente.....	16
Modulo utenteController.....	16
createUtenteControllerFn.....	16
loginUtenteControllerFn.....	16
forgotPasswordControllerFn.....	17
resetPasswordControllerFn.....	17
meUtenteControllerFn.....	17
logoutUtenteControllerFn.....	17
Modulo utenteServices.....	17
createUtenteDBService.....	18
loginUtenteDBService.....	18
generateResetToken.....	18
resetPassword.....	18
logoutUtente.....	18
findUserByEmail.....	18
findNicknameByEmail.....	19
API per la gestione del modello Utentestats.....	20
Modulo utentestatsController.....	20
createUtentestatsControllerFn.....	20
statGetterControllerFn.....	20
updateStatsControllerFn.....	20

Modulo utentestatsServices.....	20
createUtentestatsDBService.....	20
findStatsByEmail.....	21
updateUtentestatsDBService.....	21
API documentation.....	22
FrontEnd implementation.....	25
Schermata 'Home'.....	25
Schermata 'Gameplay'.....	26
Schermata 'About Us'.....	28
Schermata 'Accedi'.....	28
Schermata 'Forgot Password'.....	29
Schermata 'Reset Password'.....	30
Schermata 'Register'.....	31
Schermata 'Statistiche'.....	31
Testing.....	33
Risultati del testing.....	35
GitHub repository e informazioni sul deployment.....	36
Eseguire il server sulla propria macchina.....	36

Scopo del documento

Lo scopo del seguente documento è quello di descrivere le informazioni necessarie per lo sviluppo parziale dell'applicazione web *Vocabile*.

Nel primo capitolo è presentato l'user flow, un diagramma che descrive tutte le azioni che si possono effettuare nella parte implementata dell'applicazione. In questo diagramma troviamo quindi le richieste effettuabili a front-end e le varie risposte possibili.

Successivamente rappresentiamo lo sviluppo effettivo dell'applicazione, dunque riportiamo il codice, descrivendo le dipendenze installate, i modelli adottati e le varie API implementate.

Nel terzo capitolo vengono descritte in modo dettagliato le API implementate tramite il diagramma delle risorse e il diagramma di estrazione delle risorse, nel quale evidenzieremo le risorse estratte dal diagramma delle classi presente nel documento *D3-G48*.

Nel quarto capitolo, tramite *Swagger*, viene spiegato come è stata effettuata la documentazione delle API.

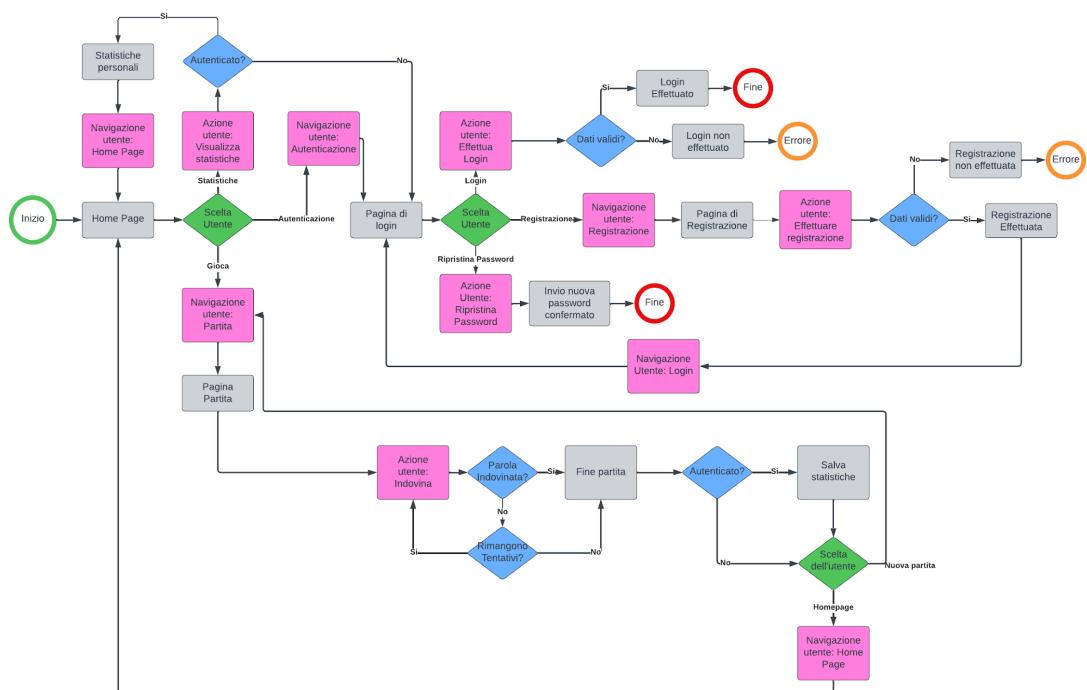
Infine, saranno presenti delle istruzioni su come effettuare il deployment del codice presente su github.

User Flow

Di seguito è riportato l'user-flow dell'applicazione, preceduto da una breve didascalia dei componenti che abbiamo utilizzato.



La scelta dell'app rappresenta degli stati dell'applicazione, che permettono all'utente di compiere o non compiere determinate azioni: l'utente non autenticato, ad esempio, non potrà salvare le proprie statistiche o visualizzarle, a differenza dell'utente autenticato.



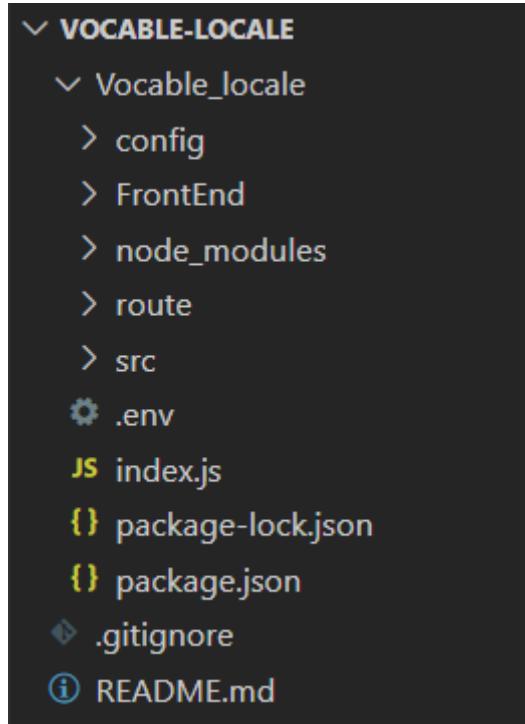
APPLICATION IMPLEMENTATION AND DOCUMENTATION

L'applicazione *Vocabile* è stata sviluppata mediante l'utilizzo del framework *VueJS*, per la parte del front-end, mentre per il back-end, sono stati utilizzati *NodeJS* ed *Express*. Per quanto riguarda la memorizzazione dei dati, abbiamo deciso di utilizzare un servizio offerto da *MongoDB*, ovvero *MongoDB Atlas*.

Come evidenziato dall'User Flow presentato nella pagina precedente, abbiamo implementato un'ampia gamma di funzionalità per garantire un'esperienza utente fluida e sicura. Tra queste, figurano la registrazione e il login, fondamentali per l'accesso alla piattaforma, accompagnate da un sistema completo per il ripristino della password, che assicura la continuità dell'accesso anche in caso di smarrimento delle credenziali. Per migliorare ulteriormente l'esperienza utente, abbiamo integrato la gestione avanzata della sessione, che consente di mantenere attiva la sessione di lavoro in modo sicuro e di effettuare il logout in qualsiasi momento. Sul fronte delle funzionalità legate al gioco, abbiamo sviluppato un sistema personalizzato di statistiche per ogni utente, che non solo tiene traccia delle performance di gioco, ma aggiorna dinamicamente i dati dopo ogni partita, fornendo un feedback immediato e dettagliato tramite l'uso di un grafico. Inoltre, abbiamo implementato i meccanismi di gioco, assicurando un'esperienza di gameplay coinvolgente e tecnicamente solida.

Struttura del Progetto

L'immagine sottostante illustra la struttura del progetto, fornendo una panoramica della sua organizzazione interna:

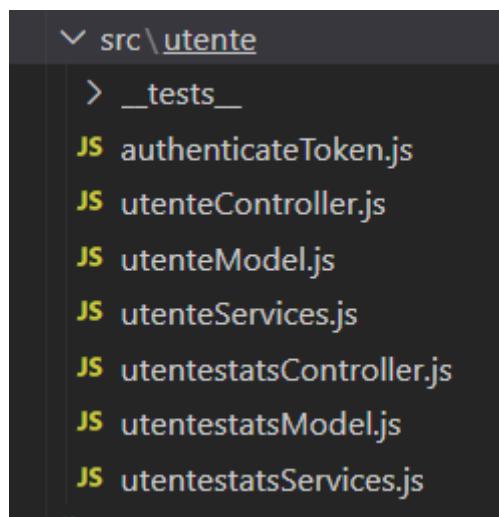


- **/config:** cartella dov'è contenuto il file setup di *Swagger* (è presente esclusivamente in locale);
- **/FrontEnd:** cartella dove sono presenti i file che gestiscono la logica di interazione con l'utente e quella di gioco;
- **/node_modules:** cartella dove sono scaricate le dipendenze presenti nel sistema;
- **/src:** cartella dov'è definita tutta la logica di interazione con l'utente a livello di back-end;
- **/route:** cartella dove è presente un file nel quale sono definiti gli endpoint delle API;
- **.env:** file dove sono presenti le variabili d'ambiente, utilizzate per una maggiore sicurezza e comodità;
- **.gitignore:** file che permette di non rendere pubblico su GitHub alcuni file (fra cui il file .env);
- **index.js:** file essenziale che serve per l'attivazione del server e per la connessione al database di *MongoDB*;
- **package-lock.json:** file automaticamente generato, blocca le versioni esatte delle dipendenze installate in un progetto *NodeJS*:

- **package.json**: file che rappresenta la configurazione generale del progetto (come il nome del progetto, versione, ecc.);
- **package-loc.json**: file che memorizza alcuni metadati sulle mutazioni relative al file package.json o alla directory node_modules;
- **swagger.json**: file dove sono descritte in modo approfondito le API implementate nel sistema con opportuni esempi.

All'interno della cartella **/src** menzionata precedentemente, troviamo una sottocartella denominata **/utente**. Questa, contiene al suo interno la cartella **/_tests_**, dove sono definiti i test cases utilizzati per il testing, e altri file, che sono essenziali per il funzionamento dell'applicazione.

La struttura è la seguente:



- **Files col suffisso *Controller*:**

sono le API che il client utilizza per interagire con l'applicazione.

- **utenteController.js**: gestisce le richieste HTTP riguardanti l'entità *utente*, collegando il client alla logica di business. Riceve le richieste, le valida, e invoca le funzioni appropriate nei servizi;
- **utentestatsController.js**: gestisce le richieste HTTP relative alle statistiche degli utenti. Riceve richieste dal client, le valida e invoca le funzioni nei servizi per ottenere o elaborare le statistiche.

- **Files col suffisso *Model*:**

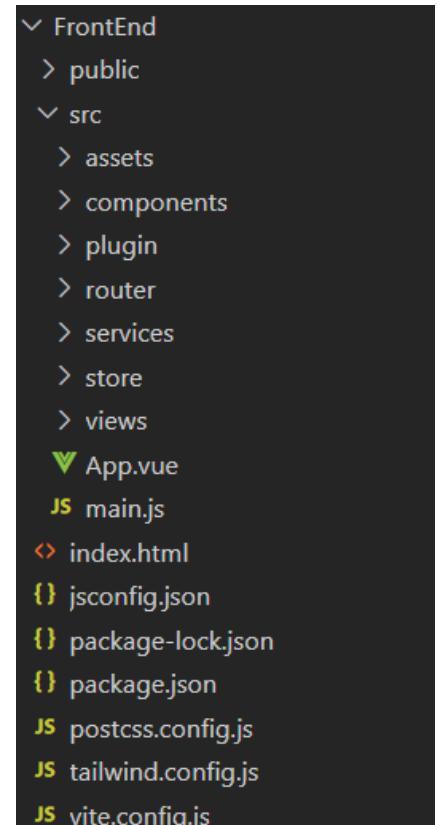
contengono le definizioni di tutti i modelli che verranno salvati nel database;

- **utenteModel.js**: contiene la definizione del modello utente quello usato per salvare i dati relativi al sistema di autenticazione;

- [utentestatsModel.js](#): contiene la definizione del modello utentestats, quello usato per salvare le statistiche;
- **Files col suffisso Services:**
sono le API interne che i controller utilizzano per eseguire la logica di business e manipolare i dati.
 - [utenteServices.js](#): contiene la logica di business per l'entità *utente*. Esegue operazioni come la creazione, lettura, aggiornamento o cancellazione degli utenti, e restituisce i risultati al controller;
 - [utentestatsServices.js](#): contiene la logica di business per calcolare, aggregare o recuperare le statistiche degli utenti. Esegue operazioni come l'analisi dei dati e restituisce i risultati al controller.

Passiamo ora all'analisi della cartella **/FrontEnd**, la quale include vari file di necessari per l'utilizzo del framework VueJS, oltre ad altre sottocartelle dedicate alla configurazione completa dell'ambiente di sviluppo.

- **/public**: cartella che contiene files statici serviti direttamente al cliente;
- **/src**: cartella che contiene tutti i file di VueJS che servono per la visualizzazione del front-end;
- **/src/assets**: cartella che contiene immagini utilizzate nell'applicazione;
- **/src/components**: cartella contenente componenti riutilizzabili e logiche modulari, progettati per essere impiegati in diverse pagine e sezioni dell'applicazione;
- **/src/plugin**: contiene i plugin utilizzati nel frontend;
- **/src/router**: cartella che contiene gli endpoint delle varie schermate;
- **/src/services**: cartella contenente alcune funzioni di servizio utilizzate nel front-end (es. controllare che l'email sia formattata in modo corretto);
- **/src/store**: cartella contenente la logica per l'autenticazione e sessione dell'utente;
- **/src/views**: cartella contenenti le viste di tutte le schermate implementate che l'utente potrà visualizzare;



Dipendenze del Progetto

I moduli Node inclusi nel file `package.json`, sotto la sezione `dependencies`, sono:

- **vue, vuex, vite**: permettono di far funzionare l'applicazione VueJS. Vue costruisce l'interfaccia, Vuex gestisce lo stato globale e Vite ottimizza il processo di sviluppo e build;
- **util**: fornisce funzioni e strumenti di supporto utili per lo sviluppo in NodeJS, come la formattazione di stringhe e la promozione di errori;
- **dotenv**: modulo che permette di usufruire delle variabili d'ambiente definite nel file `.env`;
- **cors**: modulo che permette all'applicazione di supportare il Cross-Origin Resource Sharing protocol;
- **express**: è un framework per Node.js che semplifica la creazione di server web e la gestione delle richieste HTTP;
- **express-session**: gestisce le sessioni utente in un'applicazione Express, permettendo di memorizzare e gestire dati specifici per ciascun utente tra le richieste HTTP;
- **jsonwebtoken**: fornisce funzionalità per creare e verificare JSON Web Tokens, utilizzati per l'autenticazione e l'autorizzazione tra client e server;
- **axios**: è una libreria per effettuare richieste HTTP, sia client-side che server-side, con supporto per promesse e una sintassi semplice;
- **body-parser**: analizza il corpo delle richieste HTTP e lo rende disponibile sotto forma di oggetti JavaScript, facilitando l'accesso ai dati inviati tramite POST;
- **mongoose**: è una libreria che crea una connessione tra MongoDB e l'ambiente runtime JavaScript Node.js;
- **simple-encryptor**: fornisce funzionalità di crittografia e decrittografia;
- **punycode**: gestisce la codifica e la decodifica di nomi di dominio internazionali (IDN) in formato Punycode, consentendo di utilizzare caratteri non ASCII nei nomi di dominio;
- **emailjs/browser**: permette di inviare email direttamente dal browser utilizzando un servizio di email basato su JavaScript, senza necessità di configurare un server email;
- **jest**: modulo usato per il testing delle API e delle funzioni nel back-end;
- **supertest**: modulo usato per chiamare le API in fase di testing.

Inoltre, nel campo scripts, abbiamo personalizzato l'avvio dell'applicazione utilizzando lo script `"start": "concurrently \"npm run dev:frontend\" \"npm run dev:backend\""` per lanciare simultaneamente il frontend e il backend, rispetto a una configurazione standard che avvierebbe solo il server principale. Inoltre, abbiamo specificato `"build": "cd FrontEnd && npm install && npm run build"` per gestire la costruzione del frontend nella sua directory dedicata. Gli script `"dev:frontend"` e `"dev:backend"` sono stati definiti per avviare separatamente il server di sviluppo del frontend e del backend.

Modelli nel database

Per gestire ed immagazzinare dati utili all'applicazione sono in seguito definiti due modelli di dati ispirati dalle classi identificate nel D3. Ognuno di essi è contenuto in una collezione specifica interna al database della web-app.

Modello utente

Definisce la struttura dei documenti nella collezione **utente**, collezione fondamentale per la logica del mantenimento della sessione e necessario per la creazione delle statistiche. È formato dai seguenti campi:

- **email**: un campo di tipo stringa, unico e obbligatorio, che assicura l'unicità dell'indirizzo email di ciascun utente, fungendo da identificatore;
- **password**: un campo di tipo stringa, anch'esso obbligatorio, progettato per memorizzare in modo sicuro la password dell'utente, garantendo la protezione dei dati sensibili (spiegato qui sotto il procedimento).
- **nickname**: un campo di tipo stringa che consente all'utente di scegliere un soprannome unico e personale.

```
var utenteSchema = new Schema({  
  email: {  
    type: String,  
    required: true,  
    unique: true  
  },  
  password: {  
    type: String,  
    required: true  
  },  
  nickname: {  
    type: String,  
    required: true  
  }  
},  
{ collection: 'utente' });
```

Per quanto riguarda la sicurezza dei dati sensibili, in questo caso la **password**, durante la creazione dell'utente nel backend, questo campo viene criptato tramite *simple-encryptor* e una chiave. Nel database in MongoDB, un esempio di password risulta tale, garantendo la sicurezza per gli utenti:

```
_id: ObjectId('66cf5c905b5776cc656005e7')  
email : "nits@yahoo.com"  
nickname : "Nits"  
password : "9ee02b68159f316cf9a409b311c316bac81eb6dd3e4cba95bdc7f51a8be4113b568b10..."  
__v : 0
```

Modello utenteStats

Il modello **utentestats** consente il salvataggio delle statistiche di gioco visibili a ogni utente registrato e le abbina tramite e-mail ai rispettivi giocatori.

Tutti i dati del modello, salvo la mail, sono aggiornati in seguito a ogni partita svolta da un utente registrato da apposite funzioni di gestione statistiche.

Viene creata un'istanza di questo modello, ogni volta che ne viene creata una per il modello **utente**, collegando così i due modelli tramite il campo **email**.

Tutti i campi qui descritti sono obbligatori.

- **email:** campo di tipo stringa, obbligatorio, che memorizza l'indirizzo email dell'utente, fungendo da identificatore unico per ciascun record;
- **totalgames:** campo di tipo *Number* che rappresenta il totale delle partite giocate dall'utente, fornendo una panoramica completa dell'attività di gioco;
- **gameswon:** campo di tipo *Number* che tiene traccia del numero di vittorie ottenute dall'utente;
- **gameslost:** campo di tipo *Number* che tiene traccia del numero di partite perse dall'utente;
- **won1:** campo di tipo *Number*, tiene conto delle partite vinte in un tentativo;
- **won2:** campo di tipo *Number*, tiene conto delle partite vinte in due tentativi;
- **won3:** campo di tipo *Number*, tiene conto delle partite vinte in tre tentativi;
- **won4:** campo di tipo *Number*, tiene conto delle partite vinte in quattro tentativi;
- **won5:** campo di tipo *Number*, tiene conto delle partite vinte in cinque tentativi;
- **won6:** campo di tipo *Number*, tiene conto delle partite vinte in sei tentativi;

```
var utentestatsSchema = new Schema({
  email: {
    type: String,
    required: true,
    unique: true
  },
  totalgames: {
    type: Number,
    required: true
  },
  gameswon: {
    type: Number,
    required: true
  },
  gameslost: {
    type: Number,
    required: true
  },
  won1: {
    type: Number,
    required: true
  },
  won2: {
    type: Number,
    required: true
  },
  won3: {
    type: Number,
    required: true
  },
  won4: {
    type: Number,
    required: true
  },
  won5: {
    type: Number,
    required: true
  },
  won6: {
    type: Number,
    required: true
  }
},
{ collection: 'utentestats' });
```

Project APIs

In questa parte del documento sono descritte le varie API implementate nella web-app *Vocabile*.

Tale descrizione avverrà tramite diagramma di estrazione delle risorse e diagramma delle risorse.

Entrambi presentano dati tratti dal diagramma delle classi del documento D3-G48, ma hanno maggiore specificità rispetto a esso e sono finalizzati all'illustrazione delle funzionalità implementate nel programma.

Diagramma di estrazione delle risorse

Nel seguente diagramma sono illustrati i meccanismi di estrazione delle principali risorse del sistema. Le risorse individuate grazie al diagramma delle classi sono: l'**utente** e le rispettive **utenteStats**.

Entrambe le risorse sono corredate da metodi specifici, semplici ma assolutamente essenziali al funzionamento dell'applicazione.

La risorsa utente dispone di metodi API relativi alla registrazione e all'autenticazione:

- *createUtente* per la registrazione (**POST**)
- *loginUtente* per il login (**POST**)
- *logoutUtente* per il logout (**POST**)
- *meUtente* per il passaggio delle informazioni dell'utente al sistema dopo il login (**GET**)

Oltre alle API sopracitate sono implementati endpoint per la modifica della password di un account:

- *forgotPassword* per richiedere un link di reset (**POST**)
- *resetPassword* per scegliere una nuova password che sostituisca la precedente (**POST**)

Per quanto riguarda le statistiche dell'utente, per la loro creazione, lettura e scrittura sono fornite le API:

- *createUtentestats* per creare le statistiche di un utente nel momento della registrazione (**POST**)
- *statsGetter* per leggerle e renderle disponibili alla visualizzazione da parte dell'utente (**GET**)
- *updateUtentestats* per aggiornare le statistiche dopo ogni partita (**POST**)

Accanto a ogni API è inoltre specificato l'ambito in cui è più rilevante, Frontend o Backend.

È infine importante notare che alcuni degli endpoint (in particolare *meUtente*, *logoutUtente*, *statsGetter* e *updateUtentestats*) sono protetti da un middleware di autenticazione “*authenticateToken*” basato su json web token e sono quindi accessibili soltanto attraverso account che hanno effettuato con successo il login.

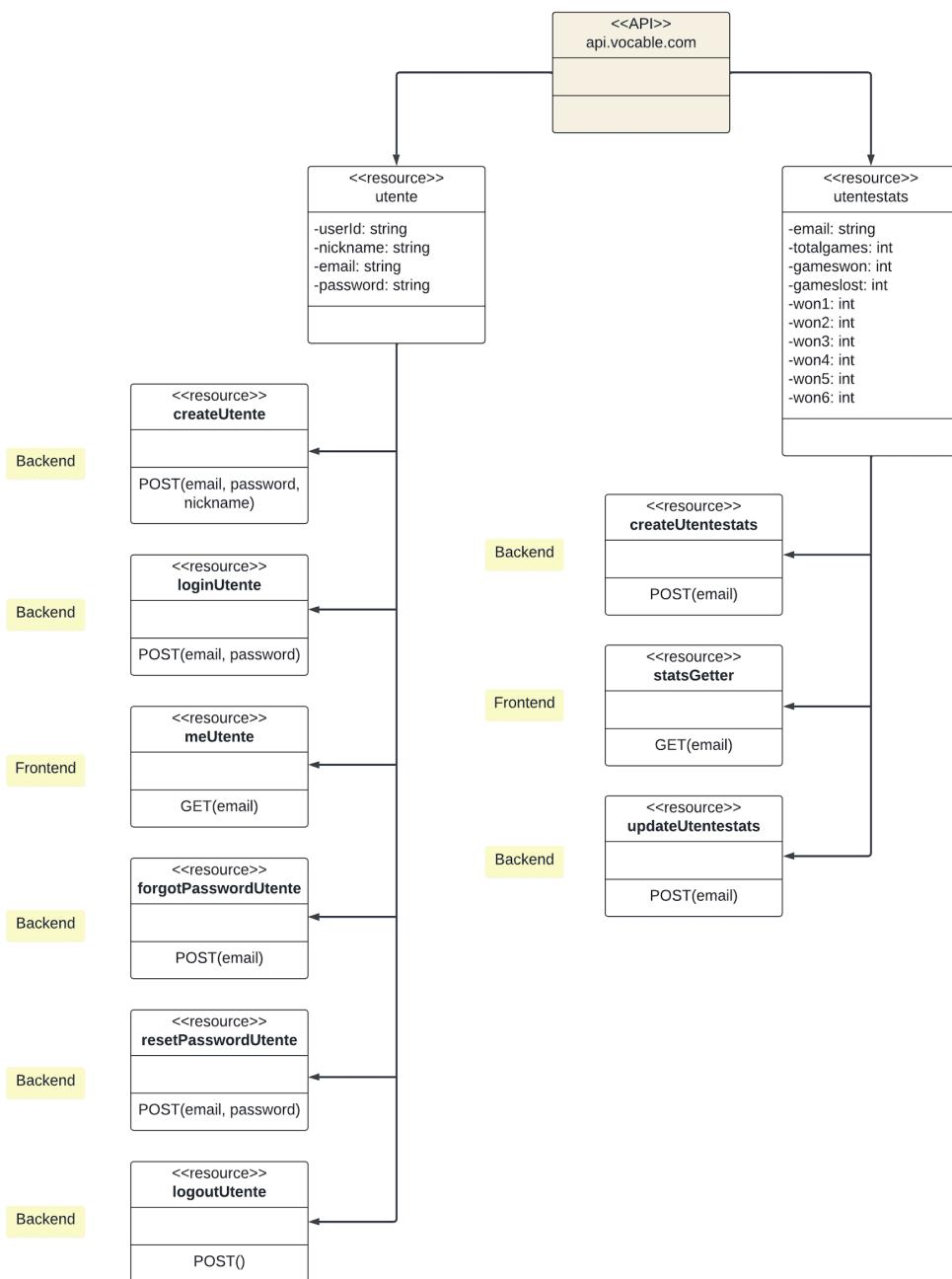
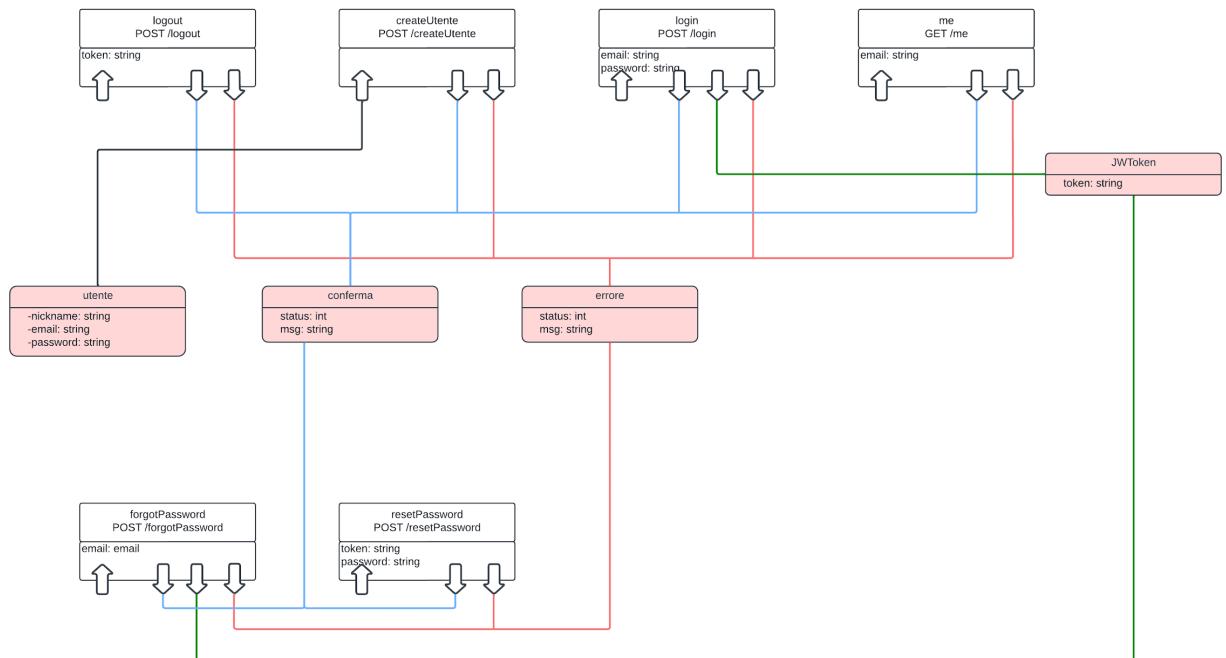


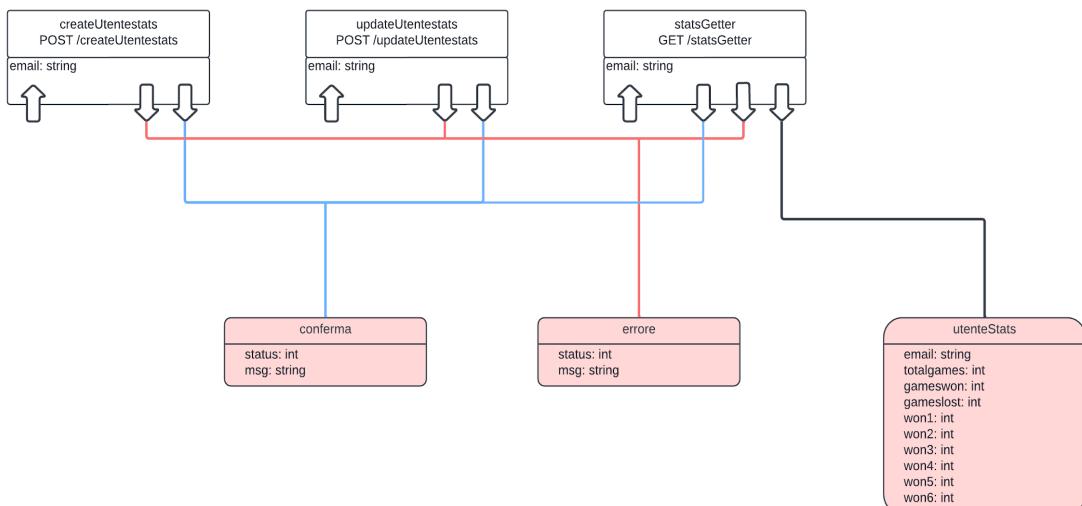
Diagramma delle risorse

Il seguente diagramma illustra in maggiore dettaglio le API presentate nel diagramma di estrazione delle risorse. Ognuno degli endpoint delle API implementate nell'applicazione è progettato per ritornare, oltre ai propri output specifici, messaggi di errore o di conferma di operazione riuscita.

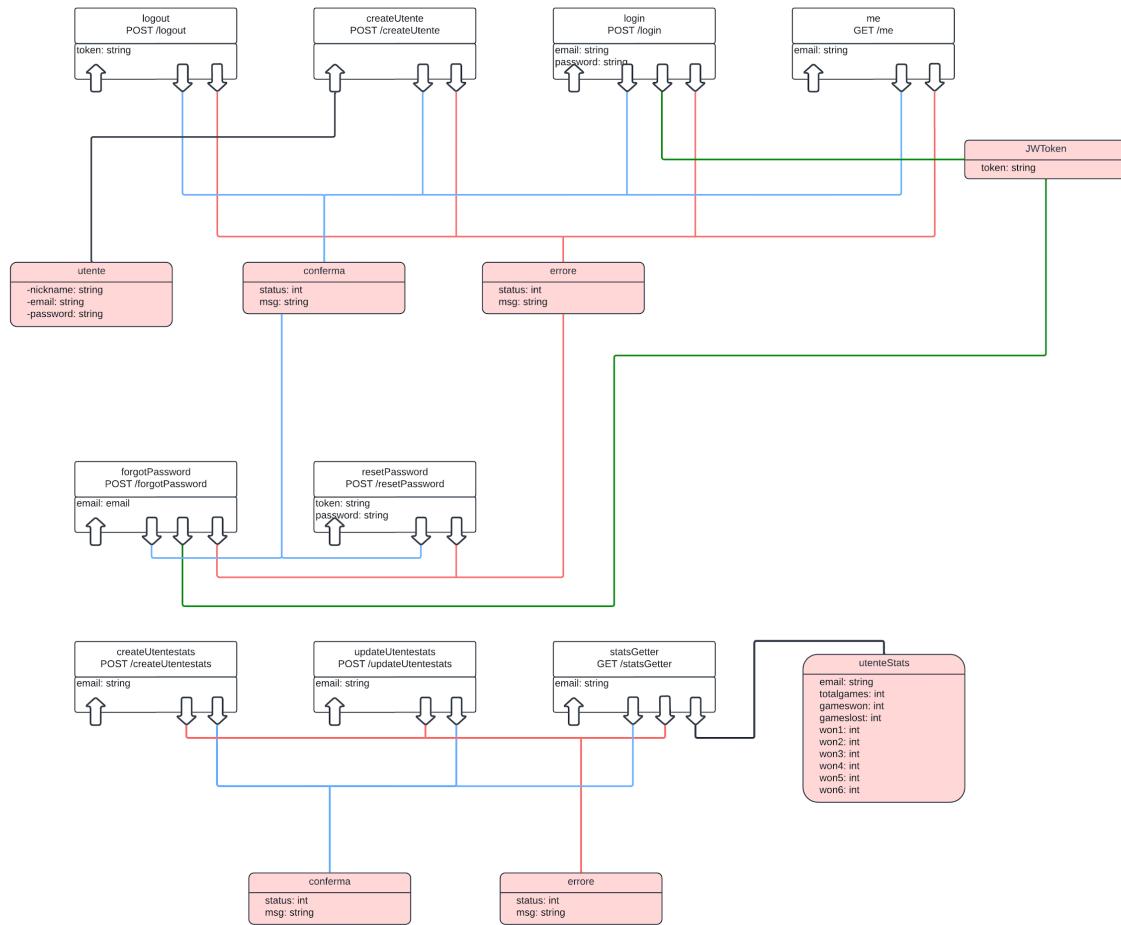
Endpoint per il modello Utente:



Endpoint per il modello Utentestats:



Rappresentazione completa degli endpoint:



Sviluppo API

Nel seguito, illustreremo il funzionamento delle diverse API sviluppate all'interno del progetto, esaminando le specifiche funzioni di MongoDB invocate al loro interno e le funzioni interne necessarie per il loro corretto funzionamento. La decisione di separare in modo modulare i file in controller e servizi nasce dall'esigenza di mantenere una chiara separazione delle responsabilità: i controller gestiscono le richieste HTTP e orchestrano il flusso delle operazioni, mentre i servizi encapsulano la logica di business e l'interazione con il database.

Non riporteremo qui tutto il codice in quanto molto lungo per ognuna di esse.
Il codice si può visionare nella [Repository](#) (`/src/utente`).

API per la gestione del modello Utente

Modulo utenteController

Le API di `utenteController` gestiscono diverse operazioni legate agli utenti, come la creazione di nuovi utenti, l'autenticazione, la gestione delle password, e la visualizzazione e disconnessione dell'account. Di seguito sono descritti le API presenti:

`createUtenteControllerFn`

L'API `createUtenteControllerFn` ha lo scopo di creare una nuova istanza del modello Utente e salvarla nel database fornito da *MongoDB*. Riceve in input un body con tutti i dati dell'utente necessari per la memorizzazione e li invia alla funzione `createUtenteDBService`. Attende la risposta da questa funzione e, in base al risultato, fornisce un messaggio di errore se l'utente non è stato creato oppure un messaggio di successo se la creazione è andata a buon fine.

`loginUtenteControllerFn`

L'API `loginUtenteControllerFn` ha lo scopo di autenticare un utente che vuole accedere al proprio account, consentendogli di usufruire dei servizi dipendenti dall'autenticazione. Questa API riceve come body i dati necessari per il login e li invia alla funzione `loginUtenteDBService`, che esegue l'operazione di login vera e propria. Dopo aver ricevuto la risposta, invia un messaggio di errore se il login non è andato a buon fine, oppure un messaggio di successo se il login è avvenuto correttamente.

forgotPasswordControllerFn

L'API `forgotPasswordControllerFn` è progettata per gestire le richieste di reset della password. Riceve l'email dell'utente come input e verifica che sia stata fornita correttamente. Successivamente, genera un token di reset tramite la funzione `generateResetToken` di `utenteService` e lo invia all'utente via email. Se il token viene generato con successo, viene restituito un messaggio di successo; altrimenti, viene restituito un messaggio di errore.

resetPasswordControllerFn

L'API `resetPasswordControllerFn` è responsabile del reset della password dell'utente. Riceve un token e una nuova password come input, e invia questi dati alla funzione `resetPassword` di `utenteService`, che esegue il reset della password. Se l'operazione è completata con successo, restituisce un messaggio di conferma, altrimenti restituisce un messaggio di errore.

meUtenteControllerFn

L'API `meUtenteControllerFn` consente di ottenere i dettagli dell'utente autenticato. Utilizzando l'email memorizzata nel token, cerca l'utente nel database e restituisce le informazioni come email, nickname e ID. Se l'utente non viene trovato, restituisce un messaggio di errore.

logoutUtenteControllerFn

L'API `logoutUtenteControllerFn` gestisce la disconnessione dell'utente. Invia un messaggio di successo quando il logout è avvenuto correttamente. In caso di errore, restituisce un messaggio di errore.

Modulo utenteServices

Nel nostro sistema, oltre alle API esposte direttamente ai client, abbiamo un modulo chiamato `utenteServices` che gioca un ruolo cruciale nella gestione delle operazioni legate agli utenti. `utenteServices` non contiene API direttamente accessibili dall'esterno, ma include una serie di **funzioni interne** che implementano la logica di business necessaria per il funzionamento delle nostre API.

Queste funzioni sono utilizzate dai controller per eseguire operazioni come la creazione di nuovi utenti, l'autenticazione, la gestione delle password e altre attività critiche. In pratica, `utenteServices` è il "motore" che alimenta le API relative agli utenti, offrendo una separazione chiara tra la logica di business e la gestione delle richieste HTTP.

Queste sono le funzioni principali di `utenteServices`:

createUtenteDBService

La funzione `createUtenteDBService` gestisce la creazione di un nuovo utente. Verifica prima se l'email fornita esiste già nel database. Se l'email è già in uso, restituisce un errore. In caso contrario, crea una nuova istanza del modello `utenteModel1`, cifra la password dell'utente utilizzando `simple-encryptor` e salva l'utente nel database. Alla fine, restituisce un messaggio di successo o un messaggio di errore a seconda del risultato dell'operazione.

loginUtenteDBService

La funzione `loginUtenteDBService` gestisce il login di un utente. Cerca l'utente nel database utilizzando l'email fornita. Se l'utente esiste, decifra la password salvata nel database e la confronta con la password fornita dall'utente. Se le password corrispondono, genera un token *JWT* con un payload contenente l'email e l'ID dell'utente e lo restituisce insieme a un messaggio di successo. Se le password non corrispondono o se l'utente non esiste, restituisce un messaggio di errore.

generateResetToken

La funzione `generateResetToken` gestisce la generazione di un token di reset della password. Cerca l'utente nel database utilizzando l'email fornita. Se l'utente viene trovato, genera un token *JWT* che scade in un'ora e lo restituisce insieme a un messaggio di successo. Se l'email non è associata ad alcun account, restituisce un messaggio di errore.

resetPassword

La funzione `resetPassword` gestisce il reset della password dell'utente. Decodifica il token *JWT* ricevuto per verificare la sua validità e, se valido, trova l'utente associato all'email nel payload del token. Quindi, cifra la nuova password fornita dall'utente e aggiorna la password nel database. Restituisce un messaggio di successo se l'operazione è completata correttamente, altrimenti restituisce un messaggio di errore.

logoutUtente

La funzione `logoutUtente` gestisce la disconnessione dell'utente. La funzione non esegue nessuna funzione particolarmente complessa, manda semplicemente un messaggio di successo al client, segnalando che il logout è avvenuto correttamente.

findUserByEmail

La funzione `findUserByEmail` cerca un utente nel database utilizzando l'email fornita. Restituisce l'intero documento dell'utente se trovato o un errore se l'operazione fallisce.

findNicknameByEmail

La funzione `findNicknameByEmail` cerca un utente nel database utilizzando l'email fornita, ma restituisce solo il campo `nickname`. Se l'utente non viene trovato, restituisce `null` o un altro valore predefinito.

API per la gestione del modello Utentestats

Modulo utentestatsController

Il modulo `utentestatsController` contiene una serie di API che gestiscono le operazioni relative alle statistiche degli utenti all'interno della nostra applicazione. Queste API permettono di creare, recuperare e aggiornare le statistiche degli utenti, interfacciandosi con il modulo `utentestatsServices` che implementa la logica di business.

Le API incluse in questo modulo sono:

`createUtentestatsControllerFn`

L'API `createUtentestatsControllerFn` ha lo scopo di creare una nuova istanza `utentestatsModel`. Riceve in input un body contenente i dati delle statistiche e li invia alla funzione `createUtentestatsDBService`, che si occupa di salvare queste informazioni nel database. In base al risultato dell'operazione, l'API restituisce un messaggio di successo o un messaggio di errore.

`statGetterControllerFn`

L'API `statGetterControllerFn` è progettata per recuperare le statistiche di un utente specifico. Utilizza l'email dell'utente autenticato, estratta dal token, per chiamare la funzione `findStatsByEmail` del modulo `utentestatsServices`, che restituisce le statistiche dell'utente dal database. Successivamente, richiama `findNicknameByEmail` dal modulo `utenteServices` per ottenere il nickname associato, utilizzato poi nella pagina `user-stats` del frontend per permettere un'esperienza più personalizzata dell'applicazione all'utente. Se entrambe le informazioni vengono trovate con successo, l'API le restituisce al client in formato JSON; altrimenti, fornisce un messaggio di errore.

`updateStatsControllerFn`

L'API `updateUtentestatsControllerFn` aggiorna le statistiche di gioco di un utente nel database. Manda i dati ricevuti nel body richiesto alla funzione interna `updateUtentestatsDBService` e se l'aggiornamento delle statistiche per quel determinato utente è andato a buon fine, l'API restituisce un messaggio di successo, altrimenti restituisce un messaggio di fallimento.

Modulo utentestatsServices

`createUtentestatsDBService`

La funzione `createUtentestatsDBService` si occupa di creare una nuova voce di statistiche per un utente, ovvero, una nuova istanza del modello `utentestatsModel`. Quando viene chiamata, controlla innanzitutto se un utente con l'email specificata esiste già nel sistema. Se un utente con quell'email esiste già, la funzione restituisce un messaggio di errore, segnalando che l'email è già in uso. In caso contrario, crea un nuovo oggetto

`utentestatsModel` e lo popola con i dati ricevuti. Infine, salva i dati nel database e restituisce una notifica di successo o fallimento.

findStatsByEmail

La funzione `findStatsByEmail` è progettata per recuperare le statistiche di un utente specifico, basandosi sulla sua email. Utilizza il metodo `findOne` di MongoDB per cercare un documento corrispondente nel database. Se trova le statistiche dell'utente, le restituisce; in caso contrario, restituisce un errore.

updateUtentestatsDBService

La funzione `updateUtentestatsDBService` gestisce l'aggiornamento delle statistiche di un utente dopo una partita. Riceve il risultato della partita, il numero di tentativi effettuati e la mail dell'utente come input. Utilizza `findStatsByEmail` per recuperare le statistiche attuali dell'utente. Se le statistiche vengono trovate, aggiorna i campi pertinenti, ovvero il numero totale di partite giocate, le partite vinte e perse, e il numero di vittorie per ciascun numero di tentativi. Dopo aver apportato le modifiche, salva il documento aggiornato nel database e restituisce un messaggio di successo. In caso di errore durante il processo, restituisce un messaggio di errore.

API documentation

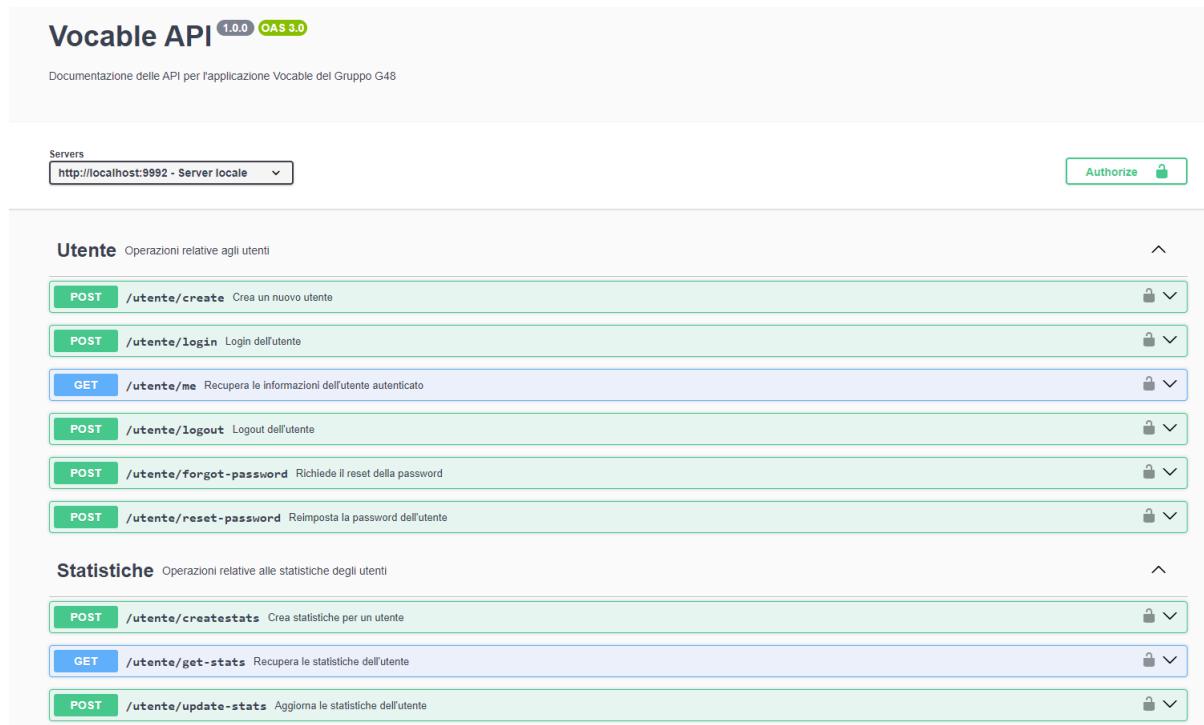
Le API descritte in precedenza e fornite dalla nostra applicazione Vocabile sono state documentate utilizzando il modulo *Swagger UI Express* di *NodeJS*. Questo strumento non solo genera automaticamente una documentazione dettagliata delle API direttamente dal codice sorgente, ma consente anche di testare le API in modo interattivo attraverso l'interfaccia di *Swagger*.

L'endpoint da invocare per raggiungere la documentazione è:

<http://localhost:9992/api-docs/>

Nota: Questo endpoint è operativo solo quando il server viene eseguito localmente.

All'apertura del link per visualizzare la documentazione *Swagger*, la pagina iniziale visualizzata sarà la seguente:



The screenshot shows the Vocabile API documentation interface. At the top, it displays "Vocabile API 1.0.0 OAS 3.0" and "Documentation delle API per l'applicazione Vocabile del Gruppo G48". Below this, there's a "Servers" dropdown set to "http://localhost:9992 - Server locale" and an "Authorize" button. The main content area is organized into sections: "Utente" (User) and "Statistiche" (Statistics). The "Utente" section contains seven API endpoints: POST /utente/create (Create a new user), POST /utente/login (Login user), GET /utente/me (Get user information), POST /utente/logout (Logout user), POST /utente/forgot-password (Forgot password), POST /utente/reset-password (Reset password). The "Statistiche" section contains three API endpoints: POST /utente/create-stats (Create user statistics), GET /utente/get-stats (Get user statistics), and POST /utente/update-stats (Update user statistics). Each endpoint row includes a lock icon and a dropdown arrow.

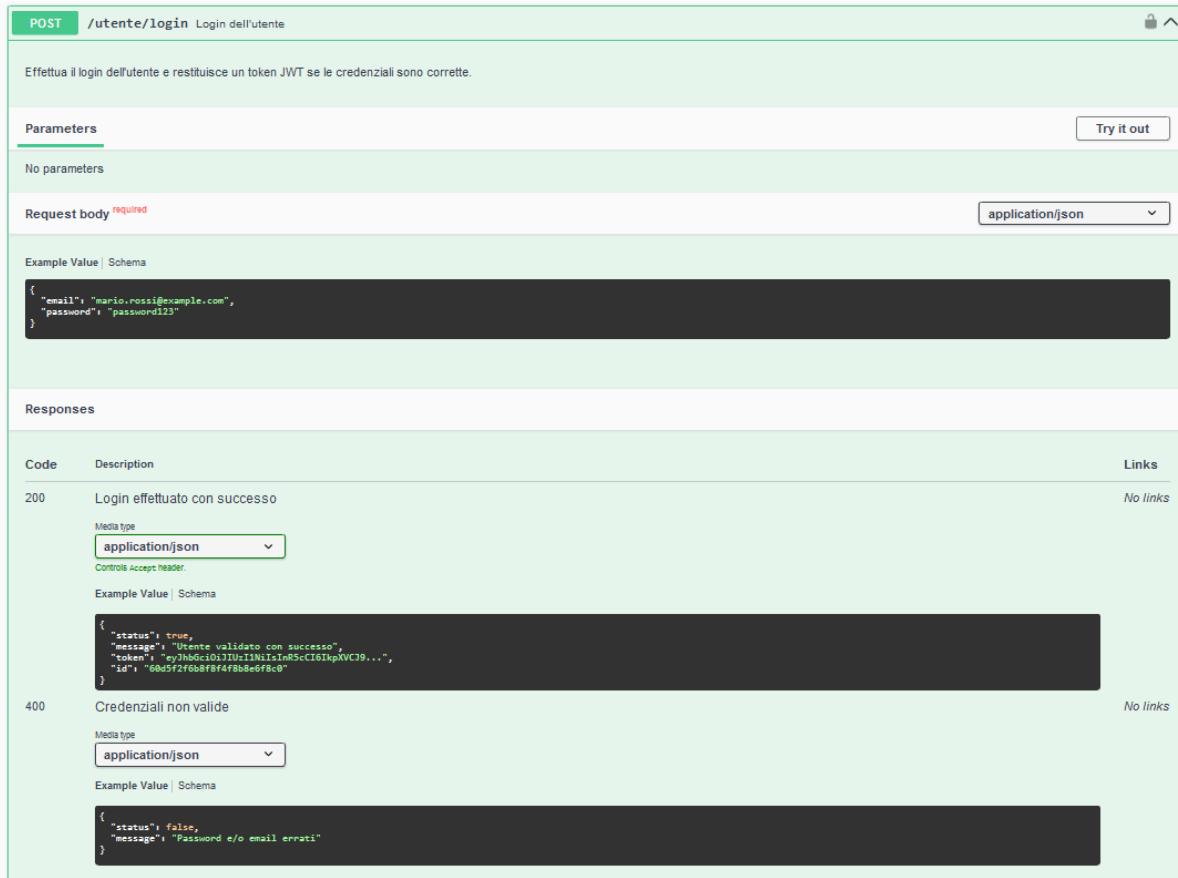
Come si può osservare, abbiamo organizzato le API in due sezioni distinte, *Utente* e *Statistiche*, per migliorare la chiarezza e la navigabilità della documentazione. Inoltre, abbiamo impiegato due metodi distinti: **POST** e **GET**.

Il metodo **POST** è utilizzato per inviare dati al sistema, come per esempio, nella creazione di nuove istanze di un utente. D'altra parte, il metodo **GET** è impiegato per recuperare dati dal sistema, ad esempio, per ottenere informazioni dettagliate su un utente.

Ogni API è dettagliatamente documentata con una descrizione concisa delle sue funzionalità. Per ciascuna API, sono specificati i tipi di dati richiesti come input,

accompagnati da valori di esempio, e le varie tipologie di risposte che possono essere restituite, complete di informazioni dettagliate.

Il seguente esempio, mostra l'API `/utente/login`:



Code	Description	Links
200	Login effettuato con successo	No links
400	Credenziali non valide	No links

Per testare le API che richiedono l'autenticazione, è necessario inserire un token, ottenuto semplicemente dopo il login fatto nella pagina stessa, utilizzando il pulsante **Authorize** situato in alto a destra nella pagina di documentazione delle API. Le API che necessitano di un token sono indicate nella loro descrizione come richiedenti un utente autenticato.

Spieghiamo dunque come fare tramite un esempio:

Dopo aver effettuato la richiesta di login tramite l'API `/utente/login`, utilizzando in questo caso i parametri di test predefiniti forniti da Swagger, il server restituirà una risposta simile alla seguente:

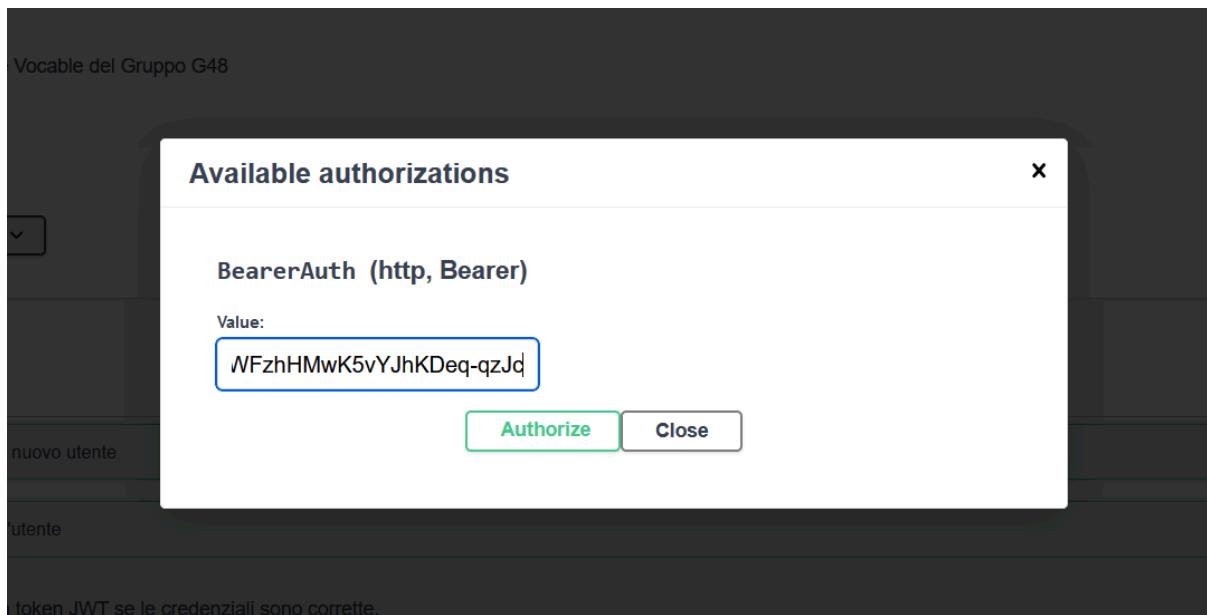
Request URL
`http://localhost:9992/utente/login`

Server response

Code	Details
200	Response body <pre>{ "status": true, "message": "Utente validato con successo", "token": "yJhbGciOiJIUzI1NiTmRscC1G3kpwVC39.ejJlbWFpbC1GIm1hcm1vLnJvc3NpQGV4YWh1vbGUyZ9tIiivaiQ1O1I2IwQ90wWZDNIhMzVhZD12NjdnYmI1NDUSLCjYXQiOjE3MjUyRTU3MzYsImV4cC16HTcyHTTzNzflNn0.JTkscZ93tvpVh8hv20expwFchMwK5vJhKDeq-qzJc", "id": "66d49c4d3a3ad2667fb545" }</pre> Response headers <pre>access-control-allow-origin: http://localhost:5173 connection: keep-alive content-length: 140 content-type: application/json; charset=utf-8 date: Sun, 01 Sep 2024 18:35:36 GMT etag: W/"13b-nM8ustKgHluy7q71CHz9eBomdY" keep-alive: timeout=5 vary: Origin x-powered-by: Express</pre>

[Download](#)

Andiamo poi a prendere il token del **Response body**, quello nel campo “**token**”, e andiamo ad inserirlo come valore nel campo “**Value**” che ci apparirà dopo aver premuto il pulsante **Authorize** in alto a destra della pagina.



Basterà poi semplicemente premere il pulsante **Authorized** e si potranno testare le API che richiedono un token.

FrontEnd implementation

In questa sezione, forniremo una descrizione dettagliata delle schermate del front-end implementate e delle azioni disponibili per l'interazione con esse.

Schermata 'Home'



La schermata '*Home*' è strutturata come segue: attraverso la barra di navigazione (*navbar*), gli utenti possono esplorare il sito e accedere alle diverse pagine: la pagina di gioco denominata '*Gioca*', la sezione '*About Us*' che descrive il team di sviluppo, e la pagina di accesso intitolata '*Accedi*'.

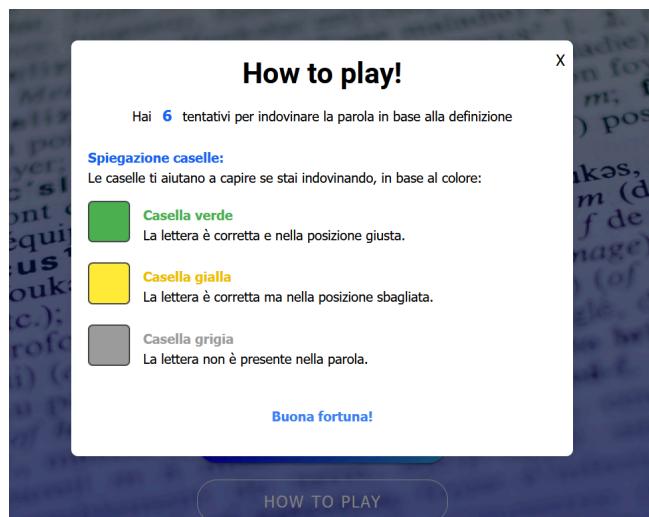
Il componente *navbar* può inoltre adattarsi in base allo stato di autenticazione dell'utente. Se l'utente è autenticato, la *navbar* presenterà le opzioni aggiuntive di '*Logout*' e delle '*Statistiche*', che sono accessibili solo agli utenti loggati.

La barra di navigazione vista da un utente autenticato è la seguente:



Ritornando alla schermata '*Home*', come si può vedere, sono presenti due pulsanti, uno che permette di giocare e l'altro, nominato "*How to Play*", che spiega le regole del gioco.

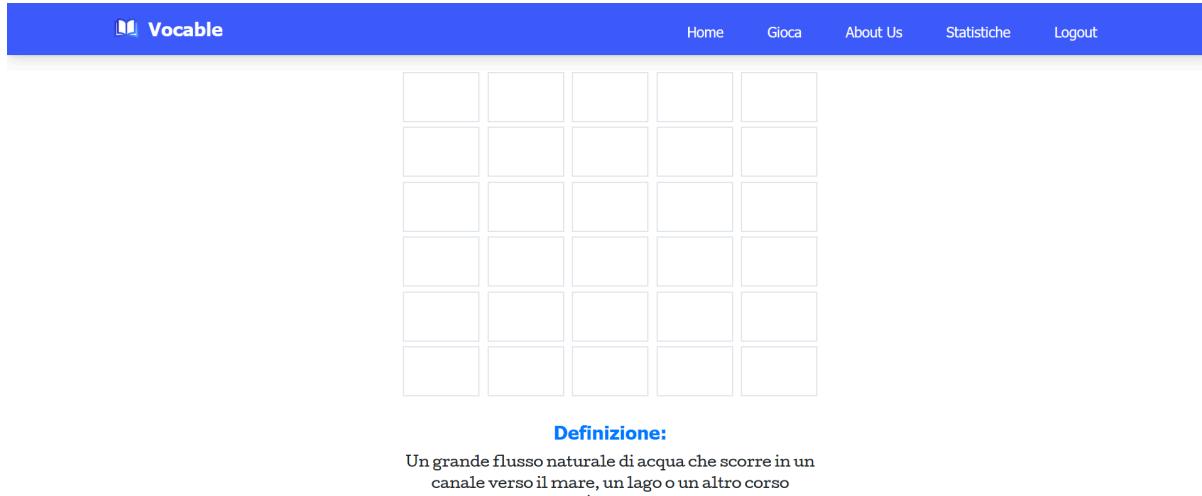
Quando si premerà il secondo, apparirà il pop-up seguente:



Questo si potrà facilmente chiudere premendo la ‘X’ in alto a destra.

Il pulsante “Gioca”, invece, reindirizzerà l’utente nella schermata di gameplay.

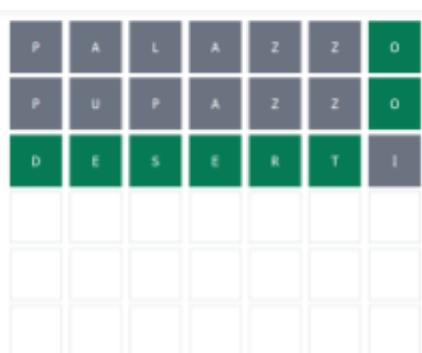
Schermata ‘Gameplay’



La schermata ‘Gameplay’ rappresenta il cuore pulsante dell’applicazione, offrendo all’utente l’opportunità di giocare e vivere l’esperienza interattiva al centro della nostra piattaforma. In essa, sono presenti le caselle della parola da indovinare, la definizione della parola e una tastiera, per permettere anche agli utenti da cellulare di giocare.

Ogni volta che l’utente invia una parola, il colore delle caselle e della tastiera si aggiorna automaticamente, seguendo le regole illustrate nella sezione “How to Play”.

Un esempio di ciò:



Esempio di tastiera

Definizione:

Un’area di paesaggio arido dove si verifica poca precipitazione e, di conseguenza, le condizioni di vita sono ostili per la vita vegetale e animale.



Esempio di tastiera

L’utente può quindi continuare a indovinare fino all’esaurimento dei tentativi disponibili. Se non riesce a vincere la partita, un messaggio sotto le caselle apparirà, rivelando la parola corretta, come illustrato di seguito:

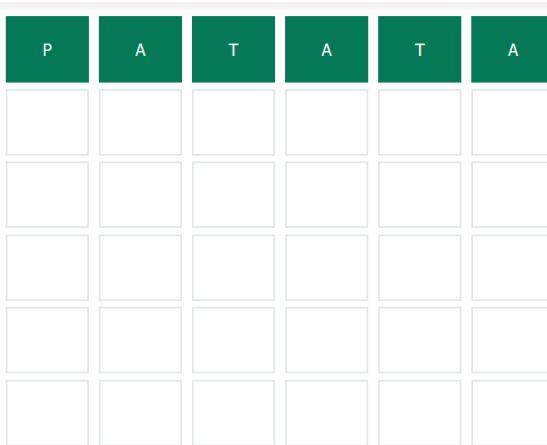
P	R	O	V	A	P	R
P	U	P	A	Z	Z	O
P	U	P	A	Z	Z	I
P	U	P	A	Z	Z	O
P	U	P	A	Z	Z	O
P	U	P	A	Z	Z	O

Peccato, hai perso!

La parola era:

inverno

Se l'utente dovesse invece vincere, verrà mostrato un messaggio di congratulazioni:



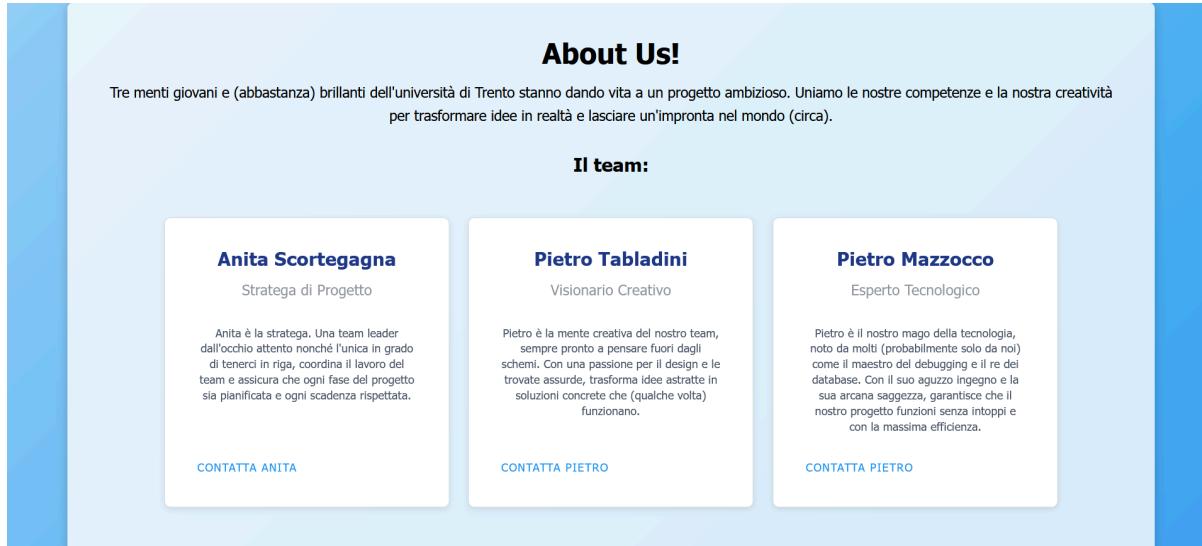
Congratulazioni! Hai trovato la soluzione!

Indipendentemente dal fatto che l'utente perda la partita esaurendo i tentativi o vinca indovinando la parola, un pulsante verrà generato in basso a destra. Questo pulsante, sempre visibile grazie alla sua posizione fissa e sopraelevata, permetterà di caricare una nuova parola.

PROSSIMA PAROLA

Da sottolineare, le statistiche si aggiornano automaticamente ogni volta che l'utente vince o esaurisce i tentativi, indipendentemente dal fatto che venga premuto il pulsante per caricare una nuova parola.

Schermata ‘About Us’

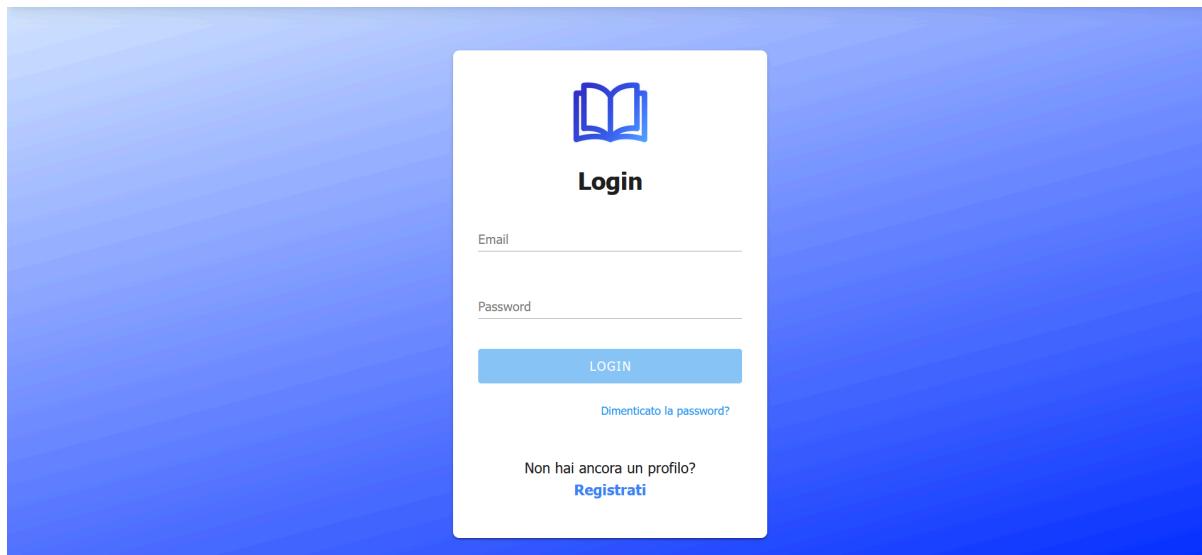


The screenshot shows the 'About Us' section of the Vocabile website. At the top, there is a brief introduction: "Tre menti giovani e (abbastanza) brillanti dell'università di Trento stanno dando vita a un progetto ambizioso. Uniamo le nostre competenze e la nostra creatività per trasformare idee in realtà e lasciare un'impronta nel mondo (circa)." Below this, the heading 'Il team:' is followed by three individual profiles:

- Anita Scortegagna** (Stratega di Progetto): Described as a team leader who coordinates work and ensures project milestones are met. A 'CONTATTA ANITA' button is present.
- Pietro Tabladini** (Visionario Creativo): Described as the creative mind of the team, always thinking outside the box. A 'CONTATTA PIETRO' button is present.
- Pietro Mazzocco** (Esperto Tecnologico): Described as the technology guru, known for his debugging skills and database knowledge. A 'CONTATTA PIETRO' button is present.

La schermata successiva, nell'ordine presente nella barra di navigazione, è la pagina ‘About Us’. Qui viene presentato il team che ha contribuito alla realizzazione di *Vocabile*. Ogni membro del team può essere contattato direttamente tramite un pulsante “Contatta” situato accanto al rispettivo nome. Cliccando su questo pulsante, si aprirà automaticamente l'applicazione di posta elettronica predefinita dell'utente (come Outlook o il client configurato nel browser) con un nuovo messaggio già indirizzato all'indirizzo email del membro selezionato.

Schermata ‘Accedi’



The screenshot shows the 'Accedi' (Login) page. It features a central login form with the following elements:

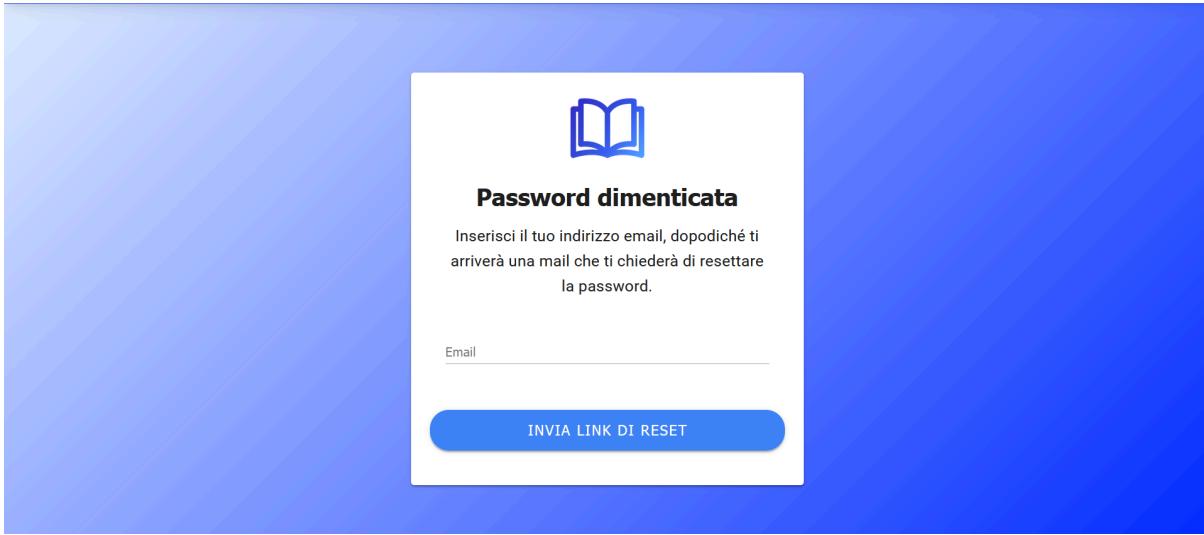
- A logo of an open book at the top.
- The word 'Login' centered below the logo.
- A 'Email' input field.
- A 'Password' input field.
- A large blue 'LOGIN' button.
- A link 'Dimenticato la password?' (Forgot password?).
- Text 'Non hai ancora un profilo?' (You don't have a profile?) followed by a 'Registrati' (Register) link.

La pagina seguente, ‘Accedi’, permette all’utente di autenticarsi e di usufruire ai servizi esclusivi dell’applicazione, tra cui le statistiche. L’operazione di *Login* andrà a buon fine se le credenziali sono corrette e già presenti nel database, in caso contrario, verrà visualizzato a video un messaggio di errore per informare l’utente del problema. L’utente è avvisato

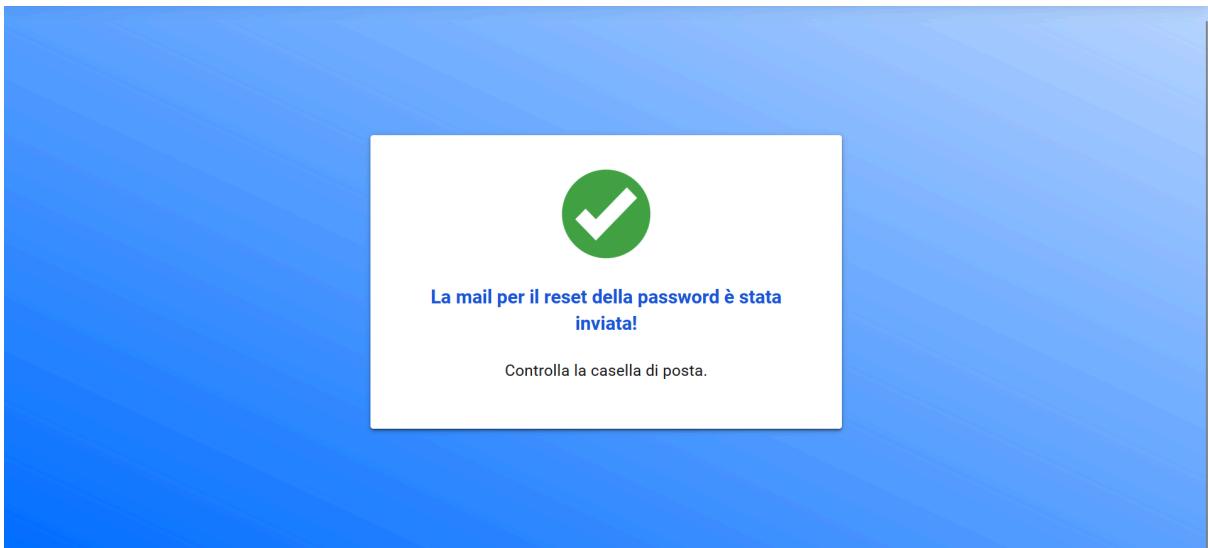
dinamicamente tramite una scritta rossa sotto i singoli campi del form dell'eventuale inadeguatezza della forma delle credenziali fornite (una password di meno di 6 caratteri o una mail del formato sbagliato).

Se l'utente dimentica la password, può semplicemente cliccare su '[Dimenticato la password?](#)', che si trova in basso a destra, sotto il pulsante di Login. Questo azionerà un reindirizzamento alla schermata '*Forgot Password*'.

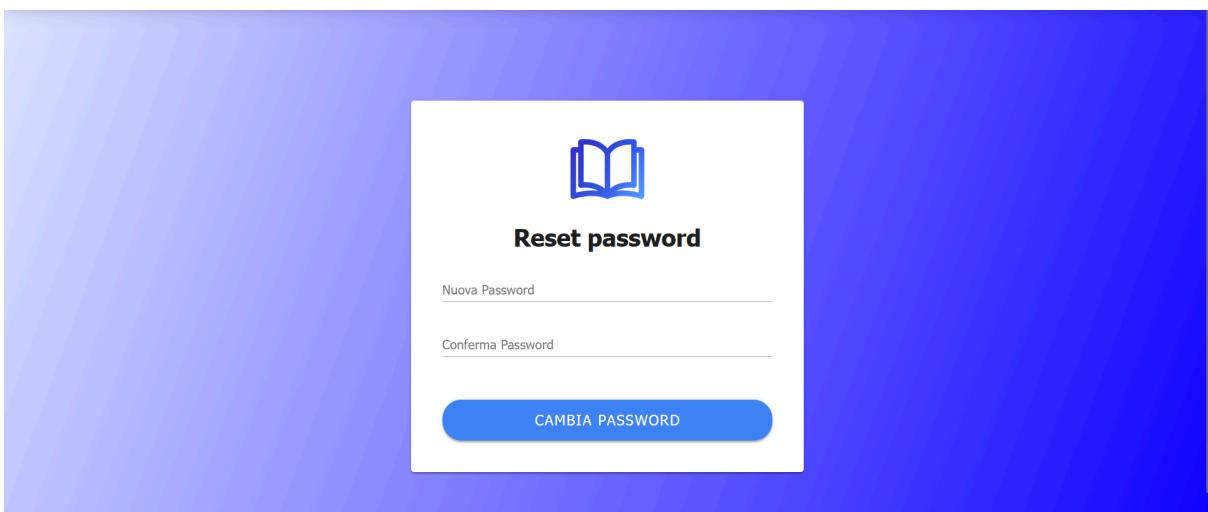
Schermata '*Forgot Password*'



La schermata '*Forgot Password*' è progettata per essere estremamente semplice. L'utente deve solo inserire l'indirizzo email per avviare il processo di recupero della password, venendo avvisato dinamicamente se la mail non è scritta nel formato giusto. Se l'email è associata a un utente presente nel database, verrà inviata una email contenente un link personalizzato. Cliccando su questo link, l'utente verrà reindirizzato alla schermata '*Reset Password*', dove potrà impostare una nuova password. Inoltre, se l'invio dell'email ha successo, verrà visualizzata una schermata chiederà all'utente di controllare la casella di posta:

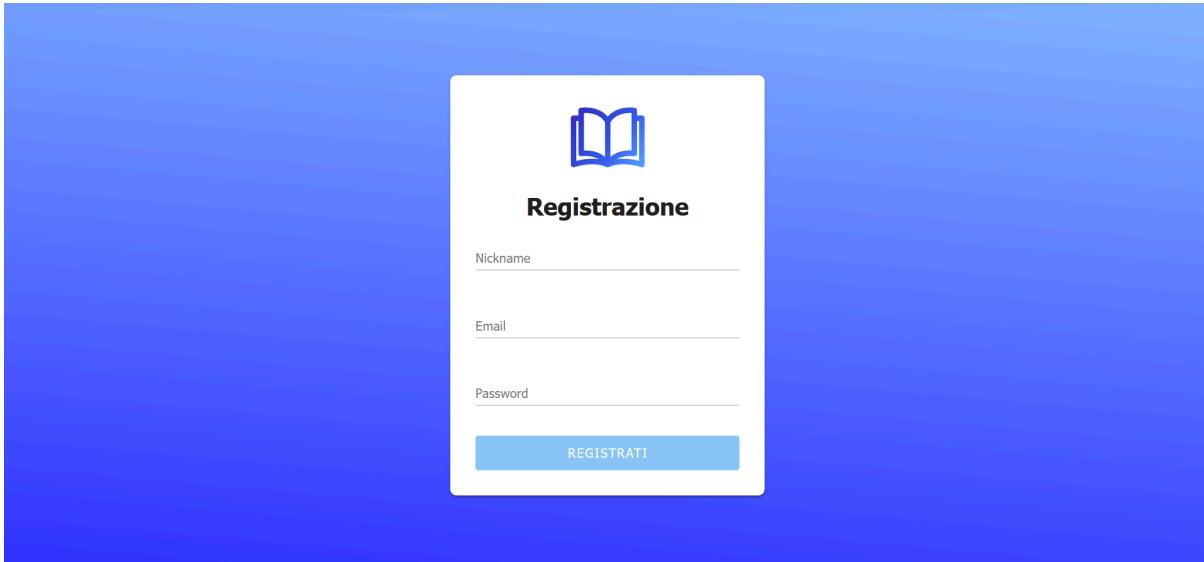


Schermata 'Reset Password'



La schermata '*Reset Password*' è stata progettata anch'essa con la massima semplicità. L'utente deve solo inserire la nuova password e confermarla, venendo avvisato dalla pagina dinamicamente se la password non corrisponde con la conferma o se è troppo corta. Una volta completato con successo il processo, l'utente riceverà una notifica tramite un pop-up che conferma l'operazione. Inoltre, sarà automaticamente reindirizzato alla schermata di login per accedere utilizzando la nuova password.

Schermata 'Register'



La schermata 'Register' è accessibile tramite il link '*Non hai ancora un profilo? Registrati*', situato sotto il pulsante di Login. Questa scelta di design, integrata direttamente nella pagina di Login, è stata fatta per garantire un'esperienza utente più fluida e senza interruzioni. In questa sezione, l'utente deve inserire un 'nickname', un'email e una password, l'utente viene avvisato dalla pagina se il formato delle credenziali non è corretto (una password minore di 6 caratteri, nickname minore di 3 caratteri o una mail non valida). Se l'email fornita non è ancora presente nel database, verrà creato un nuovo account associato a quell'indirizzo email. In caso contrario, verrà notificato all'utente che le credenziali sono già state utilizzate.

Schermata 'Statistiche'



Nella pagina '*Statistiche*' vengono visualizzati i dati relativi alle partite giocate durante la sessione di autenticazione dell'utente. La pagina accoglie l'utente con un messaggio personalizzato che utilizza automaticamente il 'nickname' associato all'email con cui si è effettuato l'accesso.

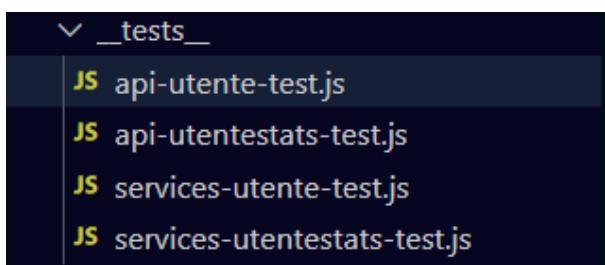
Le statistiche sono presentate in due formati: un grafico a barre e dei conteggi numerici. Il grafico a barre illustra i risultati in base ai sei tentativi massimi per indovinare la parola, mostrando il numero di vittorie ottenute in ciascuna fascia di tentativi. I conteggi numerici, invece, indicano il totale delle partite giocate, quelle vinte e quelle perse.

Testing

Il testing delle API è stato effettuato tramite le librerie ‘supertest’ di Jest e ‘mongodb-memory-server’. Queste sono rispettivamente usate per effettuare chiamate agli endpoint delle API e fornire un database fittizio in memoria su cui svolgere liberamente le operazioni di lettura e scrittura.

Per rendere il testing più accurato e completo possibile abbiamo deciso di creare funzioni di testing non solo per gli endpoint delle API di Vocabile ma anche per le funzioni di servizio su cui essi si basano.

I file di testing sono quindi 4: 2 per il testing delle API e dei servizi relativi agli utenti e 2 per quelli riguardanti le statistiche di gioco. Essi sono reperibili nella cartella ‘`__tests__`’, situata accanto alle API nella cartella backend `src/utente`.



I file di testing presentano tutti più o meno la stessa struttura. In cima al loro codice si trovano gli import necessari al loro funzionamento e funzioni `beforeAll`, `afterAll` e `afterEach` che inizializzano e terminano un’istanza locale di server MongoDB.

Una volta avviato il server ha inizio il testing effettivo, svolto tramite le funzioni che costituiscono gran parte del corpo dei file stessi. Gli endpoint delle API vengono chiamati, gli si forniscono dati esemplificativi (test@example.com, password123, etc.) e si verificano i loro comportamenti specifici in ogni situazione simulata. Dopo ogni test il database viene ripulito da dati superflui.

```
beforeAll(async () => {
  mongoServer = await MongoMemoryServer.create();
  const uri = mongoServer.getUri();
  await mongoose.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true });
});

afterEach(async () => {
  await mongoose.connection.dropDatabase();
});

afterAll(async () => {
  await mongoose.disconnect();
  await mongoServer.stop();
});
```

Riportiamo come esempio di funzione di testing quella della funzione di registrazione : 'createUtente'.

```
describe('POST /create-utente', () => {
    it('should create a user successfully', async () => {
        utenteService.createUtenteDBService.mockResolvedValue({ status: true });

        const response = await request(app)
            .post('/create-utente')
            .send({ email: 'test@example.com', password: 'password123' });

        expect(response.statusCode).toBe(200);
        expect(response.body).toEqual({
            status: true,
            message: 'Utente creato con successo'
        });
    });

    it('should return an error when creation fails', async () => {
        utenteService.createUtenteDBService.mockResolvedValue({ status: false, msg: "Errore nella creazione dell'utente" });

        const response = await request(app)
            .post('/create-utente')
            .send({ email: 'test@example.com', password: 'password123' });

        expect(response.statusCode).toBe(200);
        expect(response.body).toEqual({
            status: false,
            message: "Errore nella creazione dell'utente"
        });
    });
});
```

Ogni funzione di testing è associata a un'API (o a una serie di funzioni di servizio), attraverso supertest si effettua un mocking dei metodi necessari al corretto funzionamento dell'API e una chiamata al suo endpoint. Si confrontano quindi i risultati ottenuti dalla chiamata con quelli che ci si aspetta. Dal successo di questi confronti dipende anche il successo di ogni test.

Risultati del testing

Per effettuare i test sulla propria macchina è sufficiente scaricare la versione locale dell'app di Vocabile ed eseguire i comandi “npm i” e “npm test” (o “npm test –coverage” se si vogliono visualizzare i grafici di coverage) nella sua root repository.

Questi sono i risultati dei test presenti nei 4 file menzionati nella sezione precedente:

```
Test Suites: 4 passed, 4 total
Tests:       39 passed, 39 total
Snapshots:   0 total
Time:        3.107 s
Ran all test suites.
```

Per presentare un'analisi più approfondita dei risultati del testing è però necessario esaminare il loro “code coverage” (reperibile partendo dalla root del progetto in /coverage/lcov-report/index.html), una tabella, fornita da Jest, di dati relativi all'esecuzione del codice, degli if statement e delle funzioni.

File		Statements	Branches	Functions	Lines
authenticateToken.js		100%	12/12	100%	6/6
utenteController.js		81.66%	49/60	70%	14/20
utenteModel.js		100%	4/4	100%	0/0
utenteServices.js		80%	72/90	62.5%	20/32
utentestatsController.js		88.57%	31/35	91.66%	11/12
utentestatsModel.js		100%	4/4	100%	0/0
utentestatsServices.js		92.85%	39/42	62.5%	10/16

Come si può riscontrare dalla tabella i test non solo hanno successo ma forniscono anche un'ottima copertura del codice delle API e delle rispettive funzioni di servizio.

Le poche linee di codice che non vengono eseguite durante il testing rappresentano controlli e failsafe per errori che non avvengono in condizioni normali di esecuzione del codice.

GitHub repository e informazioni sul deployment

Tutto lo sviluppo dell'applicazione *Vocabile* è stato gestito tramite GitHub. Di seguito è riportato il link dell'organizzazione del progetto:

<https://github.com/Gruppo-G-48>

All'interno sono presenti tre repository, una per la versione online di *Vocabile*, una per la versione locale e i documenti riguardanti lo sviluppo dell'applicazione.

Versione online: <https://github.com/Gruppo-G-48/Vocabile>

Versione locale:<https://github.com/Gruppo-G-48/Vocabile-locale>

Documenti: <https://github.com/Gruppo-G-48/Deliverables>

Il deployment è stato effettuato con *Railway*. Nonostante il fatto che abbiamo utilizzato *VueJS* per il front-end, siamo riusciti a eseguire un solo deployment, anziché separare il deployment del front-end e del back-end. Abbiamo realizzato il front-end utilizzando *VueJS* insieme al framework *Vuetify*. Abbiamo inoltre impiegato gli strumenti di *Vue CLI*, eseguendo il build del progetto. Questo ci ha permesso di integrare il front-end con il back-end, consentendo un deployment unificato e senza complicazioni.

Il link della nostra applicazione su *Railway* è il seguente:

<https://vocabile-production.up.railway.app/>

Nota per la documentazione API: ricordiamo, come specificato nel capitolo corrispondente, che la documentazione delle API tramite swagger è disponibile nella versione locale (<http://localhost:9992/api-docs/>).

Eseguire il server sulla propria macchina

Nel caso in cui il link della nostra applicazione su *Railway* fosse scaduto o nel caso in cui si volesse eseguire il server locale nella propria macchina, è necessario eseguire i passaggi qui descritti.

Per prima cosa, è necessario clonare localmente il progetto dalla repository dedicata (ricordiamo, <https://github.com/Gruppo-G-48/Vocabile-locale>). Successivamente, per scaricare tutti i codici dei moduli utilizzati, sarà sufficiente eseguire il comando `npm install` (o `npm i`) nella cartella root dell'applicazione. Grazie alla modifica apportata al file `package.json`, non sarà più necessario eseguire lo stesso comando nella cartella *FrontEnd*, poiché la macchina lo farà automaticamente.

Prima di avviare l'applicazione, sarà necessario creare un file `.env`, per le variabili d'ambiente, nella cartella root dell'applicazione.

Il contenuto del file .env deve essere strettamente quello qui riportato sotto:

```
# Porta del server

PORT=9992

# Url MongoDB

MONGO_URI=mongodb://localhost:27017/utenti

# URL del client (frontend)

CLIENT_URL=http://localhost:5173

# JWT secret key

JWT_SECRET=perfavorecidia30cisiamoimpegnatitanto
```

Il parametro **PORT** si riferisce alla porta del Server, **MONGO_URI** fa riferimento al database fornito da *MongoDB*, **CLIENT_URL** è l'url del client e **JWT_SECRET** è la chiave per criptare i token *JWT*.

Dopo l'installazione delle dipendenze, sarà possibile avviare l'applicazione semplicemente digitando **npm start**. A questo punto, si potrà verificare nella console se il server è in ascolto e se la connessione a *MongoDB* è avvenuta correttamente:

```
Server in ascolto sulla porta: 9992
MongoDB Connection -- Ready state is: 1
```

Infine, verrà inoltre stampato a video il link per accedere all'applicazione (<http://localhost:5173/>).

```
VITE v5.4.2 ready in 721 ms
→ Local: http://localhost:5173/
→ Network: use --host to expose
```