

JGuesser

Sviluppo Applicazione



Dipartimento di Ingegneria e Scienza dell'Informazione

Gruppo: T24

Membri: Lorenzo D'Ambrosio, Andrea Goldoni, Marco Murru

Indice

1 User Flows	3
2 Application Implementation and Documentation	4
2.1 Project Structure	4
2.2 Project Dependencies	5
2.3 Project Data or DB	5
2.4 Project APIs	6
2.4.1 Resources Extraction from the Class Diagram	6
2.4.2 Resources Models	7
2.5 Sviluppo API	8
2.5.1 Registrazione	8
2.5.2 Autenticazione	9
2.5.3 Mostra classifica	9
2.5.4 Ricerca dati classifica di uno specifico giocatore	10
2.5.5 Ottieni i dati di uno specifico utente	10
2.5.6 Ottieni le credenziali di un utente data la sua e-mail	11
2.5.7 Ottieni le statistiche di uno specifico utente	11
2.5.8 Invia un e-mail tramite JGuesser	12
2.5.9 Impostazione di una nuova e-mail	12
2.5.10 Impostazione di una nuova password	13
2.5.11 Upgrade dell'utente a premium	13
2.5.12 Impostazione daily challenge giocata	14
2.5.13 Aggiornamento punteggio training	15
2.5.14 Aggiornamento punteggio sfida giornaliera	16
2.5.15 Generazione quiz	16
2.5.16 Richiesta sfida giornaliera	16
3 API Documentation	17
4 Front-End Implementation	18
5 GitHub Repository And Deployment Info	24
6 Testing	25

Scopo del documento

Questo documento riporta tutte le informazioni necessarie per lo sviluppo dell'applicazione JGuesser.

All'interno di questo documento saranno presentati gli user flow dell'utente guest e dell'utente loggato-premium. Il documento poi contiene l'implementazione e la documentazione delle API sviluppate, con i relativi casi di test. Infine un capitolo è dedicato al front-end dell'applicazione e un altro a come è stato gestito il github, con relativo deployment dell'applicazione.

Abbiamo deciso di non sviluppare determinate parti dell'applicazione (multiplayer, quiz tipo 4 e accesso con google/recaptcha) per non allungare esponenzialmente i tempi di sviluppo per funzionalità non caratterizzanti dell'applicazione.

1 User Flows

Si è deciso per semplicità di dividere l'user flow in due, uno per l'utente guest e l'altro per l'utente loggato/premium.

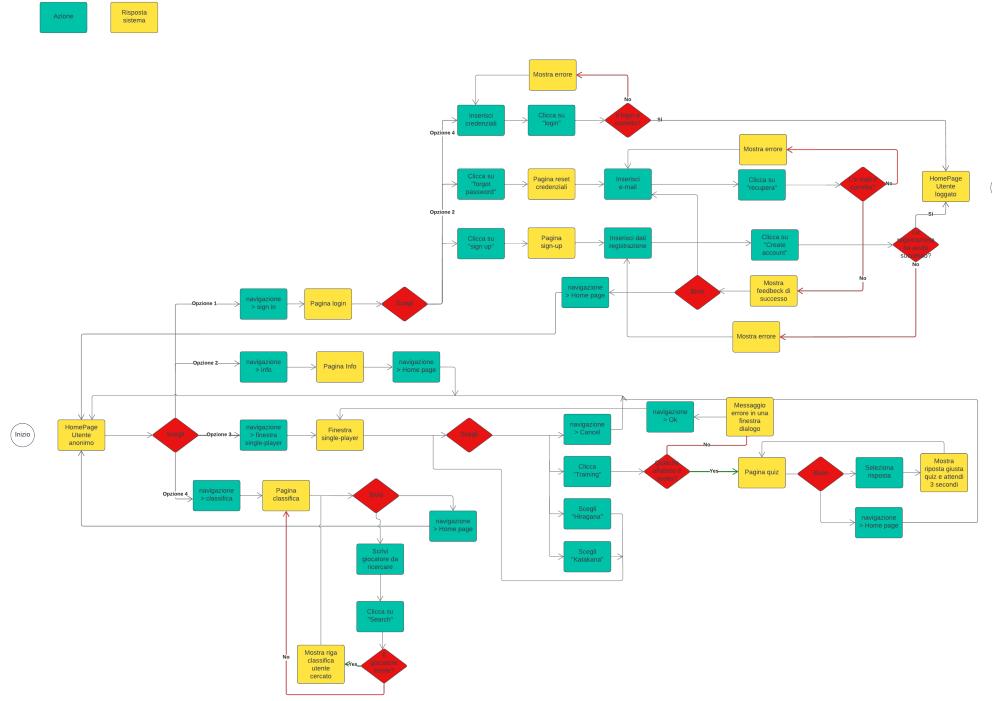


Figura 1: User Flow Guest

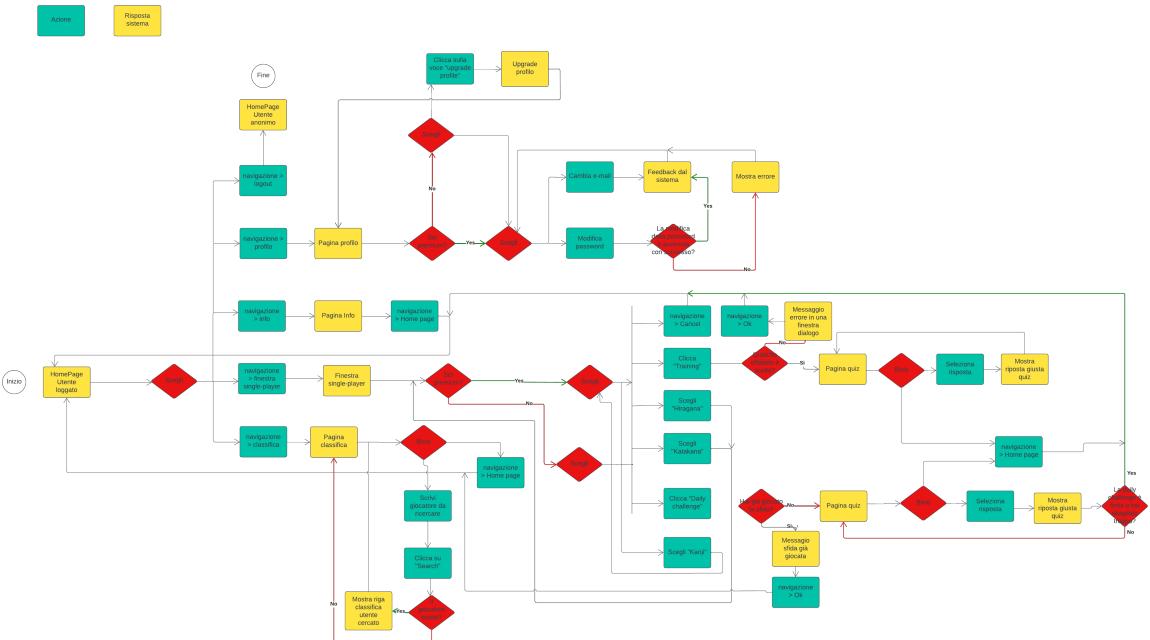


Figura 2: User Flow Loggato-Premium

2 Application Implementation and Documentation

In questa sezione sono esposte la struttura del progetto, le dipendenze software del progetto, le strutture dati implementate per il DB, le API implementate con diversi diagrammi.

2.1 Project Structure

Per il progetto abbiamo volute dividere lo sviluppo del front-end e del back-end in due cartelle separate in modo da dividerci i compiti in maniera più facile e appropriata. Nel Back-end abbiamo le seguenti cartelle:

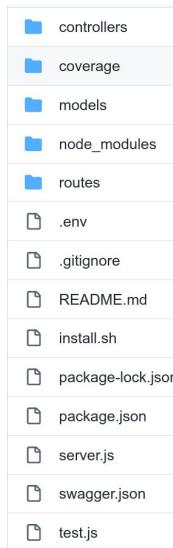


Figura 3: Struttura Back-end

Nella cartella **models**, abbiamo definito tutte le strutture dati che il database deve essere in grado di manipolare.

Nella cartella **routes** abbiamo definito tutti gli end-point delle API che abbiamo sviluppato.

Nella cartella **controllers** abbiamo inserito la definizione di questi end-point e quindi il vero e proprio codice, che viene eseguito all'invocazione delle API.

La cartella **coverage** è una cartella generata automaticamente in fase di testing dell'applicazione.

2.2 Project Dependencies

Nel Front-end abbiamo le seguenti cartelle:

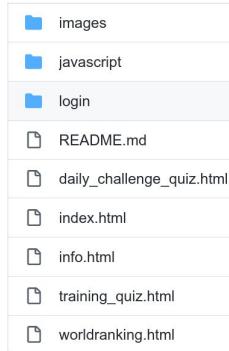


Figura 4: Struttura Front-end

Nella cartella **images**abbiamo tutte le immagini utilizzate nel Front-end.

Nella cartella **javascript** sono contenuti tutti gli script javascript delle diverse pagine.

Nella cartella **login** sono presenti le pagine HTML per il login.

Nella root della cartella Front-end sono contenute invece tutte le altre pagine HTML.

2.2 Project Dependencies

I seguenti moduli Node sono stati utilizzati e aggiunti al file 'package.json':

- **cors**: necessario per connettere Back-end con Front-end ed effettuare le chiamate alle API con successo.
- **dotenv**: necessario per utilizzare le variabili d'ambiente.
- **express**: necessario per setupprire un server.
- **jest-fetch-mock**: necessario per effettuare il testing finale dell'applicazione.
- **mongoose-express-api**: necessario per connettersi al database MongoDB.
- **nodemailer**: necessario per inviare e-mail.
- **swagger-ui-express**: necessario per produrre la documentazione con swagger.

2.3 Project Data or DB

Per la gestione dei dati utili all'applicazione sono state definite quattro strutture dati: quiz, sfida giornaliera, simbolo, e utente.

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
quizzes	0	0B	0B	4KB	1	4KB	4KB
sfidagiornalieras	1	789B	789B	36KB	2	72KB	36KB
simbolos	258	22.67KB	90B	44KB	2	80KB	40KB
users	43	9.27KB	221B	36KB	2	72KB	36KB

Figura 5: Collezione dati usati nell'applicazione

2.4 Project APIs

Definizione degli schemi delle diverse strutture dati:

```
const userSchema = new mongoose.Schema({
  username: {type:String, required:true, unique:true, dropDups:true},
  email: {type:String, required:true, unique:true, dropDups: true},
  password: {type:String, required:true},
  nation: {type:String, required:true},
  isPremium: {type:Boolean, default:false},
  hasPlayedDailyChallenge: {type:Boolean, default:false},
  statisticheUtente: {sfideGiornalieraGiocate:{type:Number, default:0}, sfideGiornalieraVinte:{type:Number, default:0},
  punteggioTraining:{type:Number, default:0}}
});
```

Figura 6: Schema definito per l’utente

```
const simboloSchema = new mongoose.Schema({
  carattereGiapponese: {type:String, required:true, unique:true, dropDups:true},
  valore: {type:String, required:true},
  alfabeto: {type:String, required:true}
});
```

Figura 7: Schema definito per il simbolo

```
const quizSchema = new mongoose.Schema({
  tipo: {type: Number},
  domanda: {type: String},
  alfabeto: {type: String},
  opzione: [{type: String}],
  soluzione: {type: String}
});
```

Figura 8: Schema definito per il quiz

```
const sfidaGiornalieraSchema = new mongoose.Schema({
  tipoDiSfida: {type:Number, required:true},
  data: {type:Date, required:true, unique:true, default: Date.now()},
  listaDiQuiz: [{}]
});
```

Figura 9: Schema definito per la sfida giornaliera

2.4 Project APIs

2.4.1 Resources Extraction from the Class Diagram

Nel diagramma delle classi possiamo notare diverse ”risorse” (tipi di dati con relativi metodi nativi del nostro programma). A diverse di queste risorse sono associate delle API che svolgono determinati ruoli all’interno del progetto. In questo diagramma vengono espresse prima le risorse (come Utente, Quiz o Sfida Giornaliera) e sotto di loro collegate tramite frecce le rispettive API.

Ogni API è caratterizzata dal tipo (POST / PATCH / GET), da degli eventuali parametri di cui ha bisogno per funzionare e infine da un riferimento Front-end o Back-end, a seconda del ruolo che svolge l’API, se principalmente di front-end o di back-end.

Nel nostro progetto la risorsa con più API è chiaramente Utente, infatti abbiamo bisogno di API quali login, signup, nuova e-mail per tutte quelle funzionalità che riguardano il profilo dell’utente. Abbiamo in aggiunta anche delle API strettamente legate al progetto stesso come la visualizzazione della classifica o delle proprie statistiche, o il passaggio a premium per poter usare il training Kanji.

2.4 Project APIs

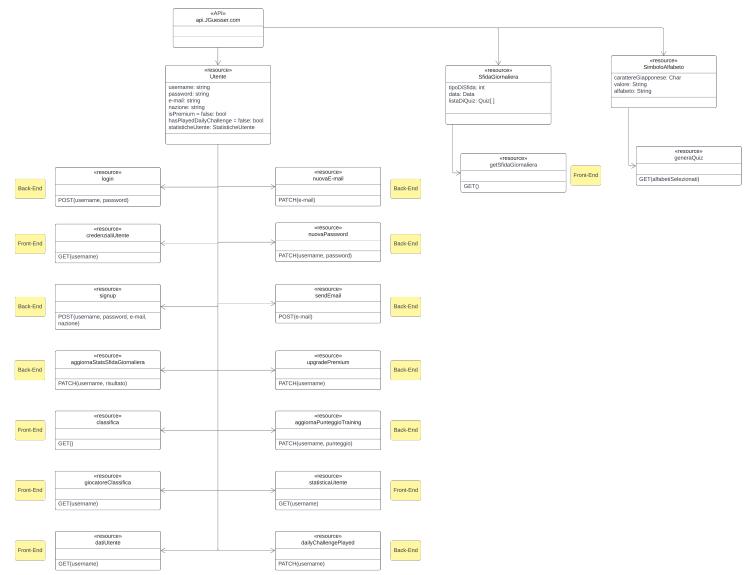


Figura 10: Diagramma "Resources Extraction from Class Diagram"

2.4.2 Resources Models

Nel diagramma delle risorse sono spigate nel dettaglio tutte le API del progetto, dal percorso interno a "routes" al nome dell'API e ai singoli casi di input e output. Abbiamo rispettato le convenzioni "HTML" per i messaggi di stato individuando nel nostro progetto diversi stati possibili per diverse API. Quelli più comuni sono il 404 NOT FOUND, 200 OK, 304 NOT MODIFIED e 500 INTERNAL SERVER ERROR. La maggioranza delle API non ha bisogno di un body di input ma solamente di parametri passati nell'URI. Un caso paticolare nel nostro progetto è il logout visto che non è stato necessario implementare un'API apposita grazie al tipo di implementazione dell'autenticazione e della registrazione (tramite localStorage).

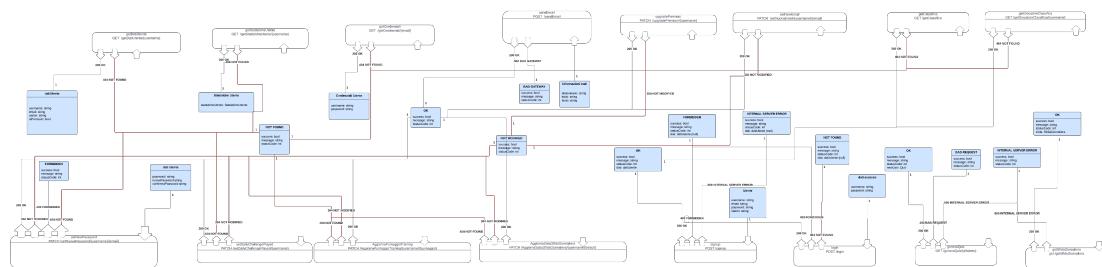


Figura 11: Diagramma delle Risorse

2.5 Sviluppo API

2.5 Sviluppo API

In questa sezione sono descritte le varie API che abbiamo sviluppato.

2.5.1 Registrazione

```
const signUp = (req, res) => {
    //Viene verificato che sia presente tale user, filtrando per username
    user.findOne({ username: req.body.username }, (err, data) => {
        //Non ci sono utenti con quell'username, è possibile inserire
        if (!data) {
            //Viene creato un nuovo username
            //DA MODIFICARE I PARAMETRI IN BASE ALLO SCHEMA DI modello
            const newUser = new user({
                username: req.body.username,
                email: req.body.email,
                password: req.body.password,
                nation: req.body.nation
            });

            // Viene salvato l'user nel database
            newUser.save((err, data) => {
                if (err) return res.json({
                    success: false,
                    statusCode: 500,
                    message: 'Error in saving the user',
                    dati: null
                });
                return res.json({
                    success: true,
                    statusCode: 200,
                    message: 'User successfully inserted',
                    dati: data
                });
            });
        } else {
            //Se c'è l'username è già presente
            return res.json({
                success: true,
                statusCode: 403,
                message: 'User already in use',
                dati: null
            });
        }
    });
};
```

Figura 12: Implementazione API signUp

Questa API di tipo POST permette all’utente di registrarsi. I parametri richiesti per registrare un utente sono:

- username;
- e-mail;
- password;
- nation.

Questa API restituisce un codice:

- 200: la registrazione è avvenuta con successo e vengono restituiti anche i dati di registrazione;
- 403: l’utente non può essere registrato perché esiste già un account con quel username;
- 500: errore generico, che significa che ci è stato un errore lato server nel salvare i dati nel database.

2.5 Sviluppo API

2.5.2 Autenticazione

```
const authentication = async function (req, res) {
    /* CREDO che res.json equivalga a return (anche se a me sembrava si dovesse usare res.end)
    //Per prima cosa si cerca se l'utente esiste, verificando che l'username sia corretto
    let data = await user.findOne({ username: req.body.username }).exec();

    // Nel caso l'utente non venga trovato
    if (!data) {
        return res.json({
            success: false,
            statusCode: 404,
            message: 'User not found',
            dati: null
        });
    }
    else{
        // Nel caso venga trovato un utente, si verifica che la password sia corretta
        if (data.password != req.body.password) {
            return res.json({
                success: true,
                statusCode: 403,
                message: 'Password does not match',
                dati: null
            });
        }
        else{
            // Restituisce i dati dell'utente e un token di autenticazione
            return res.json({
                id: data._id,
                username: data.username,
                isPremium: data.isPremium,
                hasPlayedDailyChallenge: data.hasPlayedDailyChallenge,
                success: true,
                statusCode: 200,
                message: 'Enjoy your token!',
                //token: token, //Una stringa criptata da jwt
            });
        }
    }
};
```

Figura 13: Implementazione API login

Questa API di tipo POST permette all'utente di autenticarsi. I parametri richiesti per autenticare un utente sono:

- username;
- password.

Questa API restituisce un codice:

- 200: l'autenticazione è avvenuta con successo e vengono restituiti una serie di dati necessari per l'utente;
- 403: la password che l'utente ha inserito non corrisponde alla password memorizzata all'interno del database;
- 404: non c'è nessun utente con quello specifico username.

2.5.3 Mostra classifica

```
const getClassifica = (req, res) => {
    user.find({}, 'username nation statisticheUtente', (err, data) => {
        if (err) {
            return res.json({
                success: false,
                statusCode: 500,
                message: 'Error no data',
                dati: null
            });
        }
        return res.json({
            success: true,
            statusCode: 200,
            message: "Returned the user's ranking data.",
            Classifica: data
        });
    }).sort({ "statisticheUtente.punteggioTraining": -1 });
};
```

Figura 14: Implementazione API classifica

2.5 Sviluppo API

Questa API di tipo GET restituisce tutti i dati della classifica in ordine decrescente di punteggio training. Questa API restituisce un codice:

- 200: i dati sono stati restituiti correttamente;
- 500: c'è stato un errore lato server e non si è riuscito a restituire i dati.

2.5.4 Ricerca dati classifica di uno specifico giocatore

```
const getGiocatoreClassifica = (req, res) => {
  let username = req.params.username; //Prendi l'username per il quale filtrare, preso dall'url

  //Trova l'user con l'username voluto
  user.findOne({ username: username }, 'username nation statisticheUtente', (err, data) => {
    if (err || !data) {
      return res.json({
        success: false,
        statusCode: 404,
        message: 'User not found',
        dati: null
      });
    }
    else return res.json({
      success: true,
      statusCode: 200,
      message: "Returned the user's ranking data.",
      dati: data
    });
  });
};
```

Figura 15: Implementazione API giocatore classifica

Questa API di tipo GET restituisce tutti i dati della classifica di uno specifico giocatore. È richiesto di inserire nell'url della richiesta l'username del giocatore che si intende cercare. Questa API restituisce un codice:

- 200: l'utente è stato trovato e se ne restituiscono i dati da visualizzare;
- 404: non c'è nessun utente con quello specifico username.

2.5.5 Ottieni i dati di uno specifico utente

```
const getDatiUtente = (req, res) => {
  let username = req.params.username; //Prendi l'username per il quale filtrare, preso dall'url
  //console.log(req.params.username);
  //Trova l'user con l'username voluto
  //Per restituire anche la password, aggiungerla tra gli apici del findOne
  user.findOne({ username: username }, 'username email nation isPremium', (err, data) => {
    if (err || !data) {
      return res.json({
        success: false,
        statusCode: 404,
        message: 'User not found',
        dati: null
      });
    }
    else return res.json({
      success: true,
      statusCode: 200,
      message: "Here is the user's data",
      dati: data
    });
  });
};
```

Figura 16: Implementazione API dati utente

Questa API di tipo GET restituisce i dati di uno specifico utente. Per dati di uno specifico utente si intende:

- username;
- e-mail;
- nation;

2.5 Sviluppo API

- isPremium.

Sono in sostanza quelli che sono visualizzati nella prima parte della propria pagina profilo. Questa API restituisce un codice:

- 200: l'utente è stato trovato e se ne restituiscono i dati da visualizzare;
- 404: non c'è nessun utente con quello specifico username.

2.5.6 Ottieni le credenziali di un utente data la sua e-mail

```
const getCredenziali = (req, res) => {
  let email = req.params.email; //Prendi l'username per il quale filtro
  console.log("Cerco utente con email = " + email);

  user.findOne({ email: email }, 'username password', (err, data) => {
    if (err || !data) {
      return res.json({
        success: false,
        statusCode: 404,
        message: 'The searched e-mail does not exist.',
        dati: null
      });
    }
    else return res.json({
      success: true,
      statusCode: 200,
      message: "Here are the required credentials",
      dati: data
    });
  });
};
```

Figura 17: Implementazione API credenziali

Questa API di tipo GET restituisce le credenziali di uno specifico utente data l'e-mail associata. Questa API restituisce un codice:

- 200: l'utente è stato trovato e se ne restituiscono username e password;
- 404: non c'è nessun utente con quella specifica e-mail.

2.5.7 Ottieni le statistiche di uno specifico utente

```
const getStatisticheUtente = (req, res) => {
  let username = req.params.username; //Prendi l'username per il quale filtro
  //console.log(req.params.username);
  //Trova l'utente con l'username voluto
  user.findOne({ username: username }, 'statisticheUtente', (err, data) => {
    if (err || !data) {
      return res.json({
        success: false,
        statusCode: 404,
        message: 'User not found',
        dati: null
      });
    }
    else return res.json({
      success: true,
      statusCode: 200,
      message: "Here are the required statistics",
      dati: data
    });
  });
};
```

Figura 18: Implementazione API statistiche utente

Questa API di tipo GET restituisce le statistiche di uno specifico utente. Questa API restituisce un codice:

- 200: l'utente è stato trovato e se ne restituisco le statistiche;
- 404: non c'è nessun utente con quello specifico username.

2.5 Sviluppo API

2.5.8 Invia un e-mail tramite JGuesser

```
const sendEmail = (req, res) => {
  let mailTransporter = nodemailer.createTransport({
    service: 'gmail',
    auth: {
      user: 'japanguesser@gmail.com',
      pass: 'krroioormalitezpo'
    }
  });

  let mailDetails = {
    from: 'japanguesser@gmail.com',
    to: req.body.destinatario, //lorenzo
    subject: req.body.titolo,
    text: req.body.testo
  };

  mailTransporter.sendMail(mailDetails, function(err, data) {
    if(err) {
      return res.json({
        success: false,
        statusCode: 502,
        message: 'Error'
      });
    } else {
      res.json({
        success: true,
        statusCode: 200,
        message: 'Email sent'
      });
    }
  });
}
```

Figura 19: Implementazione API send e-mail

Questa API di tipo POST prende come parametri in input nel body:

- destinatario: e-mail di destinazione.
- titolo: oggetto dell'e-mail;
- testo: contenuto dell'e-mail.

e invia con l'indirizzo e-mail 'japanguesser@gmail.com' un e-mail con le specifiche appena descritte. Questa API restituisce un codice:

- 200: l'e-mail è stata inviata con successo;
- 502: non è stato possibile inviare l'e-mail.

2.5.9 Impostazione di una nuova e-mail

```
const setNuovaEmail = (req, res) => {
  user.findOneAndUpdate({ username: req.params.username }, { $set: {email: req.params.email} }, { new: true }, (err, newEmail) => {
    if (err) {
      return res.json({
        success: false,
        statusCode: 304,
        message: 'It was not possible to update the e-mail'
      });
    } else {
      res.json({
        success: true,
        statusCode: 200,
        message: 'Email updated.'
      });
    }
  });
}
```

Figura 20: Implementazione API nuova e-mail

Questa API di tipo PATCH prende come parametri nell'URL l'username con la nuova e-mail da settare e modifica l'e-mail dell'utente. Questa API restituisce un codice:

- 200: l'utente è stato trovato e l'e-mail è stata modificata con successo;
- 304: c'è stato un errore e non è stato possibile modificare la risorsa.

2.5 Sviluppo API

2.5.10 Impostazione di una nuova password

```
const setNuovaPassword = (req, res) => {
    user.findOne({ username: req.params.username }, 'password', (err, data) => {
        if (err || !data) {
            return res.json({
                success: false,
                statusCode: 404,
                message: 'User not found',
                dati: null
            });
        }
        else{
            if(data.password == req.body.vecchiaPassword){
                user.findOneAndUpdate({ username: req.params.username }, { $set: {password: req.params.password} }, { new: true }, (err, newPass) => {
                    if (err) {
                        return res.json({
                            success: false,
                            statusCode: 304,
                            message: 'It was not possible to update the password'
                        });
                    } else {
                        return res.json({
                            success: true,
                            statusCode: 200,
                            message: "Password updated"
                        });
                    }
                });
            }
            else{
                return res.json({
                    success: true,
                    statusCode: 403,
                    message: "The old password and the new one do not match.",
                    dati: null
                });
            }
        }
    });
};
```

Figura 21: Implementazione API nuova password

Questa API di tipo PATCH prende come parametri nell'URL l'username con la nuova password da settare e nel body la vecchia password. Quello che fa è controllare che la vecchia password corrisponda a quella memorizzata nel database e poi va a settare quella nuova. Questa API restituisce un codice:

- 200: l'utente esiste, la password vecchia corrisponde e il server è riuscito a settare la nuova password;
- 304: c'è stato un errore e non è stato possibile modificare la risorsa;
- 404: l'utente non esiste e quindi non è possibile fare alcuna modifica;
- 403: la vecchia password inserita e quella presente sul database non corrispondono.

2.5.11 Upgrade dell'utente a premium

```
const upgradePremium = (req, res) => {
    user.findOneAndUpdate({ username: req.params.username }, { $set: {isPremium: true} }, { new: true }, (err, newPass) => {
        if (err) {
            return res.json({
                success: false,
                statusCode: 304,
                message: 'It was not possible to upgrade the user'
            });
        }
        else {
            return res.json({
                success: true,
                statusCode: 200,
                message: 'The user is now a premium user'
            });
        }
    });
};
```

Figura 22: Implementazione API upgrade premium

Questa API di tipo PATCH prende come parametri nell'URL l'username dell'utente a cui verrà effettuato l'upgrade a premium. Questa API restituisce un codice:

2.5 Sviluppo API

- 200: i privilegi dell’utente sono stati aggiornati con successo;
- 304: c’è stato un errore e non è stato possibile modificare la risorsa.

2.5.12 Impostazione daily challenge giocata

```
const setDailyChallengePlayed = (req, res) => {
  user.findOne({ username: req.params.username }, 'username hasPlayedDailyChallenge', (err, data) => {
    if (err || !data) {
      return res.json({
        success: false,
        statusCode: 404,
        message: 'User not found'
      });
    }
    else{
      if(data.hasPlayedDailyChallenge)
        return res.json({
          success: true,
          statusCode: 200,
          message: "already played"
        });
      else{
        user.findOneAndUpdate({ username: req.params.username }, { $set: {hasPlayedDailyChallenge: true} }, { new: true }, (err, result) => {
          if (err) {
            return res.json({
              success: false,
              statusCode: 304,
              message: 'The status of the daily challenge could not be updated'
            });
          }
          else {
            res.json({
              success: true,
              statusCode: 200,
              message: 'User has played daily challenge'
            });
          }
        });
      }
    }
  });
}
```

Figura 23: Implementazione API impostazione challenge

Questa API di tipo PATCH prende come parametri nell’URL l’username dell’utente a cui verrà settata la daily challenge come giocata. Nel caso in cui questo l’abbia già giocata viene restituito il messaggio ’already_played’. Questa API restituisce un codice:

- 200: significa che hasPlayedDailyChallenge è settato a true o è stato trovato, già settato a ’true’;
- 304: c’è stato un errore e non è stato possibile modificare la risorsa.
- 404: l’utente per il quale si vuole eseguire questa operazione non esiste.

2.5 Sviluppo API

2.5.13 Aggiornamento punteggio training

```
const aggiornaPunteggioTraining = (req, res) => {
    var username = req.params.username;
    var punteggio = req.params.punteggio;
    user.findOne({ username: req.params.username }, 'statisticheUtente', (err, stats) => {
        if (err || !stats) { //se il giocatore non esiste (non viene trovato)
            return res.json({
                success: false,
                statusCode: 404,
                message: 'User not found'
            });
        } else {
            if(stats.statisticheUtente.punteggioTraining<=punteggio){
                user.findOneAndUpdate({ username: req.params.username }, { $set: {"statisticheUtente.punteggioTraining": punteggio } })
                    .if (err) { //se c'è stato un errore nel aggiornare le statistiche dell'utente
                        return res.json({
                            success: false,
                            statusCode: 304,
                            message: 'The player statistics could not be updated'
                        });
                    } else {
                        return res.json({ //statistiche aggiornate con successo
                            success: true,
                            statusCode: 200,
                            message: 'Score updated'
                        });
                    }
            } else
                return res.json({ //successo, ma non era necessario aggiornare le statistiche
                    success: true,
                    statusCode: 200,
                    message: 'Score not updated'
                });
        }
    });
}
```

Figura 24: Implementazione API aggiorna punteggio training

Questa API di tipo PATCH prende come parametri nell’URL l’username dell’utente e il nuovo punteggio raggiunto nel training. Quello che questa API fa è aggiornare il massimo punteggio raggiunto nel training del giocatore. Questa API restituisce un codice:

- 200: significa che questa operazione di controllo e in caso di aggiornamento è avvenuta con successo;
- 304: c’è stato un errore e non è stato possibile modificare la risorsa;
- 404: l’utente per il quale si vuole potenzialmente aggiornare il proprio punteggio non esiste.

2.5 Sviluppo API

2.5.14 Aggiornamento punteggio sfida giornaliera

```
const aggiornaStatsSfidaGiornaliera = (req, res) => {
    var username = req.params.username;
    var result = req.params.result;
    user.findById({ username: username }, 'statisticheUtente', {err, stats} => {
        if (err || !stats) {
            return res.json({
                success: false,
                statusCode: 404,
                message: 'User not found'
            });
        } else {
            var giocate = stats.statisticheUtente.sfideGiornaliereGiocate++;
            var vinte = stats.statisticheUtente.sfideGiornaliereVinte++;
            if(result=="sfidaVinta"){
                stats.sfideGiornaliereGiocate++;
                console.log("true");
                user.findByIdAndUpdate({ username: username }, { $set: {"statisticheUtente.sfideGiornaliereGiocate": giocate, "statisticheUtente.sfideGiornaliereVinte": vinte} }, {err} => {
                    if (err) {
                        return res.json({
                            success: false,
                            statusCode: 304,
                            message: 'The player statistics could not be updated'
                        });
                    } else {
                        return res.json({
                            success: true,
                            statusCode: 200,
                            message: 'Score updated'
                        });
                    }
                });
            } else{
                console.log("false");
                user.findByIdAndUpdate({ username: req.params.username }, { $set: {"statisticheUtente.sfideGiornaliereGiocate": giocate} }, { new: true }, {err, newPunt} => {
                    if (err) {
                        return res.json({
                            success: false,
                            statusCode: 304,
                            message: 'The player statistics could not be updated'
                        });
                    } else {
                        return res.json({
                            success: true,
                            statusCode: 200,
                            message: 'Score updated'
                        });
                    }
                });
            }
        }
    });
}
```

Figura 25: Implementazione API aggiorna sfida giornaliera

Questa API di tipo PATCH prende come parametri nell’URL l’username dell’utente e la conclusione della sfida giornaliera (‘sfidaVinta’ o ‘sfidaPersa’) e aggiorna le statistiche per la sfida giornaliera di conseguenza. Questa API restituisce un codice:

- 200: significa che le statistiche sono state aggiornate con successo;
- 304: c’è stato un errore e non è stato possibile modificare la risorsa;
- 404: l’utente per il quale si vogliono aggiornare le statistiche non esiste.

2.5.15 Generazione quiz

Questa API di tipo GET prende come parametri nell’URL un array contenente l’insieme di alfabeti presi in considerazione durante la generazione di un quiz. Questa API restituisce un codice:

- 200: quiz generato con successo;
- 500: errore lato server nel generare il quiz;
- 400: c’è stato un errore nel formato della richiesta.

2.5.16 Richiesta sfida giornaliera

Questa API di tipo GET richiede la sfida giornaliera della data attuale e nel caso non esista ne fa generare una nuova dal server. Questa API restituisce un codice:

- 200: sfida giornaliera trovata/generata con successo e restituita;
- 500: errore lato server nel generare la sfida giornaliera.

3 API Documentation

Le API fornite dall'applicazione JGuesser sono state documentate utilizzando Swagger UI Express. In questo modo la documentazione relativa alle API è direttamente disponibile a chiunque le voglia utilizzare per altri progetti. È possibile accedere alla documentazione tramite il seguente link in locale: localhost:8080/api-docs/, oppure accedendo all'URI: /api-docs/ in remoto (con dominio del server). Nel nostro caso il link è il seguente: https://JGuesser-BackEnd-Unitn.up.railway.app/api-docs/.

The screenshot shows the Swagger UI interface for the JGuesser API version 1.0.0. At the top, it displays the title "JGuesser 1.0.0" and the base URL "[Base URL: localhost:8080/]". Below this, it lists the available APIs under sections like "Guest", "User", "Simbolo", and "Sfida giornaliera". Each section contains a list of endpoints with their methods, URLs, and descriptions. For example, the "Guest" section includes endpoints for "/signUp", "/login", "/getClassifica", and "/getGiocatoreClassifica/{username}". The "User" section includes endpoints for "/getDatiUtente/{username}", "/getCredenziali/{email}", and "/getStatisticheUtente/{username}". The "Simbolo" section includes the endpoint "/generaQuiz". The "Sfida giornaliera" section includes the endpoint "/getSfidaGiornaliera". The interface uses a dark theme with blue and green highlights for different sections and endpoints.

Figura 26: Documentazione delle API di JGuesser

4 Front-End Implementation

Il FrontEnd fornisce delle schermate finalizzate alla fruizione dei quiz, altre finalizzate alla gestione degli utenti ed infine due schermate ausiliarie: una che fornisce informazioni sugli alfabeti e una che mostra l'attuale classifica in base ai punteggi accumulati dagli utenti.



Figura 27: Schermata home per utenti non loggati



Figura 28: Schermata home per utenti loggati

All'interno della home page è possibile trovare le seguenti opzioni:

- Pulsante “INFO”: da tale pulsante sarà possibile accedere ad una schermata mostrante informazioni riguardanti gli alfabeti presenti all’interno dell’applicazione. Al di sotto sarà riportato il nome dell’utente (nel caso di utente loggato) oppure ”Utente anonimo”
- Pulsante “SINGLE-PLAYER”: attraverso tale pulsante si aprirà una finestra di dialogo (figura 29 o figura 30) che permetterà all’utente di scegliere tra 2 (utente non loggato) o 3 (utente loggato) alfabeti. L’utente non loggato avrà inoltre a disposizione soltanto la modalità di “Training”, mentre l’utente loggato avrà a disposizione la “Daily challenge”, da eseguire massimo una volta al giorno.



Figura 29: Scelta modalità di gioco per utenti non loggati

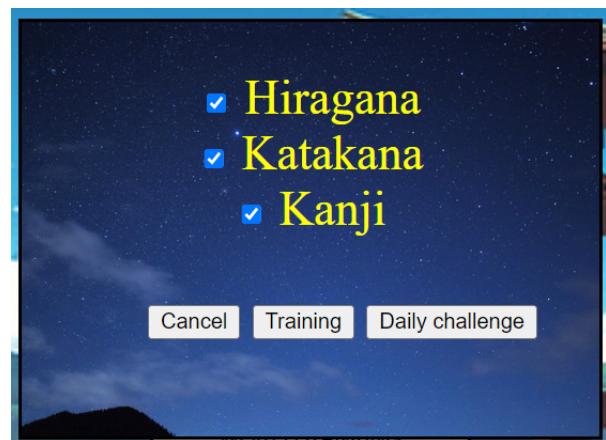


Figura 30: Scelta modalità per utenti loggati

Il pulsante “Cancel” chiuderà la finestra di dialogo, mostrando nuovamente la schermata home.

- Pulsante “WORLD RANKING”: cliccando su questo pulsante sarà possibile visualizzare la classifica degli utenti (figura 31).
- A seconda del fatto che l’utente sia loggato o meno, in alto a destra della schermata home compariranno pulsanti differenti:
 - SIGN IN: l’utente non loggato verrà reindirizzato alla schermata di login (figura 33)
 - PROFILO: l’utente loggato potrà visualizzare dati e statistiche personali (figura 36), permettendo inoltre la modifica di email e password
 - LOGOUT: l’utente loggato verrà reindirizzato alla schermata home (figura 27) e le sue credenziali verranno dimenticate



	Rank	Nationality	Username	Points
1	USA	topolino2003	1231232	
2	Italy	Dambro	1000	
3	Italy	Akai_Marco	31	
4	Spain	SE-T24	0	
5	England	Elton	0	

Figura 31: Classifica



	Rank	Nationality	Username	Points
#	Italy	Dambro	1000	

Figura 32: Classifica

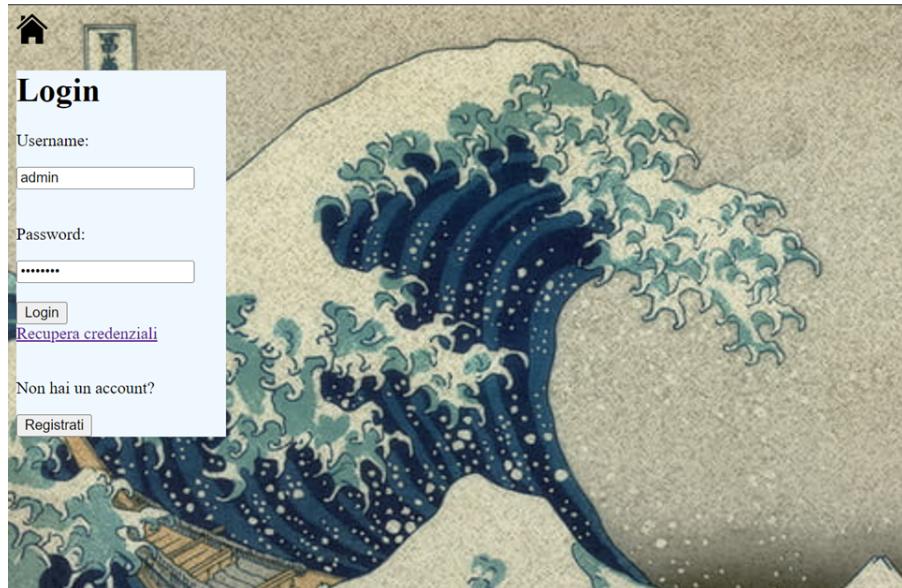


Figura 33: Schermata di login



Figura 34: Recupero credenziali



Figura 35: Schermata di registrazione

Dalla schermata di login sarà possibile, dopo aver inserito le proprie credenzia-

li e cliccato sul pulsante “LOGIN”, visualizzare la schermata home da utente loggato (figura 28). Nel caso non si ricordino le credenziali, sarà possibile, inserendo l'email associata all'account, ottenere via mail le proprie credenziali (figura 34). Dalla schermata di login sarà inoltre possibile registrarsi, cliccando sull'apposito pulsante “Registrati” e compilando i relativi campi (figura 35)



Figura 36: Schermata profilo utente

Una volta giunto alla schermata relativa al proprio profilo, l'utente potrà visualizzare i propri dati personali e le proprie statistiche, inoltre, compilando i relativi form, potrà modificare la propria email o la propria password.

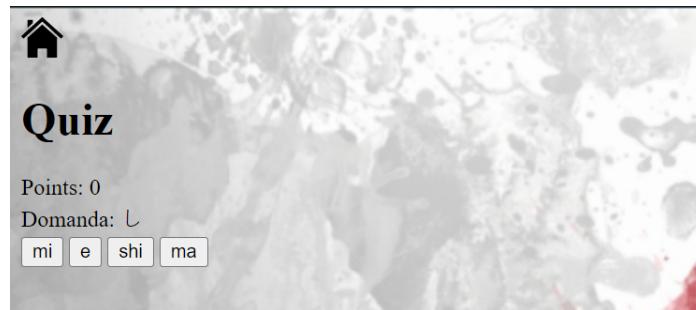


Figura 37: Schermata di quiz

Selezionando una delle due modalità di gioco (figura 29/30) l'utente visualizzerà una schermata contenente un quiz generato casualmente (in caso di training) o relativo al quiz prescelto per la sfida giornaliera (in caso di daily challenge). Una volta visualizzata la schermata, l'utente dovrà selezionare (o digitare, a seconda della tipologia di quiz) la risposta che ritiene corretta (eventualmente premendo INVIA RISPOSTA). A questo punto il sistema comunicherà all'utente se la risposta selezionata è errata (figura 38) o corretta (figura 39), in quest'ultimo caso il punteggio andrà ad incrementarsi.



Figura 38: Schermata di quiz, risposta errata



Figura 39: Schermata di quiz, risposta esatta



Figura 40: Schermata quiz, daily challenge

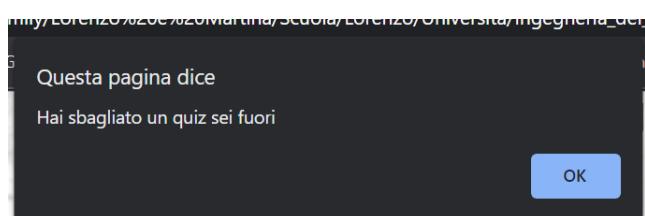


Figura 41: Schermata quiz, errore daily challenge

La schermata rimarrà invariata per 3 secondi, per poi lasciar spazio al quiz successivo. Per terminare la partita, all'utente basterà premere il pulsante home presente in alto a sinistra (figura 37) per essere reindirizzato alla schermata di home (figura 27/28). Il sistema provvederà ad aggiornare il punteggio ottenuto nel training o nella sfida giornaliera, segnando quest'ultima come completata. Sarà possibile effettuare una nuova partita nella modalità “daily challenge” a partire dalla mezzanotte dello stesso giorno. Nel caso della sfida giornaliera, quest'ultima terminerà al compimento di essa, o in seguito ad un certo quantitativo di errori (in base alla tipologia di sfida prestabilita), il tutto segnalato all'utente (figura 41)

5 GitHub Repository And Deployment Info

Per strutturare il progetto abbiamo deciso di utilizzare 3 repositories all'interno di GitHub: Derivables (contenente i vari derivables da consegnare), Front-End (contenente tutti i file di Front-End, compresi file .css, .html e .javascript, quest'ultimo contenente gli script utilizzati dalle pagine html) e Back-End (contenente tutti i file che permettono l'utilizzo, l'implementazione e la documentazione delle API, il testing e i collegamenti al database MongoDB).

Per quanto riguarda il deployment, abbiamo scelto di utilizzare railway, che offre un servizio di deployment gratuito, per un totale di 500 ore (attivo fino al 21/01/2023). Di seguito sono elencati i link per raggiungere le cartelle di Github ed il deployment dell'applicazione:

- Link Github gruppo
- Link repository Back-End
- Link repository Front-End
- Link repository Derivables
- Link swagger
- Link deployment Back-End
- Link deployment Front-End

6 Testing

Il testing delle API è stato fatto tutto all'interno di un unico file chiamato 'test.js'. È possibile trovare questo file nella root della cartella di Back-end. I test che sono stati fatti sono abbastanza banali, in quanto si è verificato che soltanto lo 'statusCode' e qualche volta il campo 'message' presente all'interno della risposta fossero corretti. Ad esempio per l'API di registrazione sono stati impostati i seguenti casi di test:

```
//signUp
test("It should successfully register the user and return a status code 200", async () => {
  expect.assertions(1);
  const input =
  {
    username: "paperinik2004",
    email: "paperinik@gmail.com",
    password: "paperino_il_più_bello",
    nation: "USA"
  };
  const response = await request(app).post("/signUp").set({Content-Type: "application/json"}).send(input);
  expect(await response.body.statusCode).toEqual(200);
});
test("It should not allow the user to register because they already exist and return a status code of 403", async () => {
  expect.assertions(1);
  const input =
  {
    username: "paperinik2004",
    email: "paperini@gmail.com",
    password: "paperino_il_più_bello",
    nation: "USA"
  };
  const response = await request(app).post("/signUp").set({Content-Type: "application/json"}).send(input);
  expect(await response.body.statusCode).toEqual(403);
});
```

Figura 42: Esempio test API registrazione

Sono state realizzate quattro test suite, una per ogni file presente all'interno della cartella controllers:

- Unlogged user API test suite: sono stati fatti tutti i test delle API che è possibile invocare se non si è loggati all'interno dell'applicazione;
- Logged user API test suite: sono stati fatti tutti i test delle API che è possibile invocare se si è loggati all'interno dell'applicazione;
- API to generate a quiz: sono stati fatti i test dell'API che viene utilizzata per generare i quiz;
- API to generate a daily challenge: sono stati fatti i test dell'API che viene utilizzata per generare la sfida giornaliera.

I risultati sono stati i seguenti:

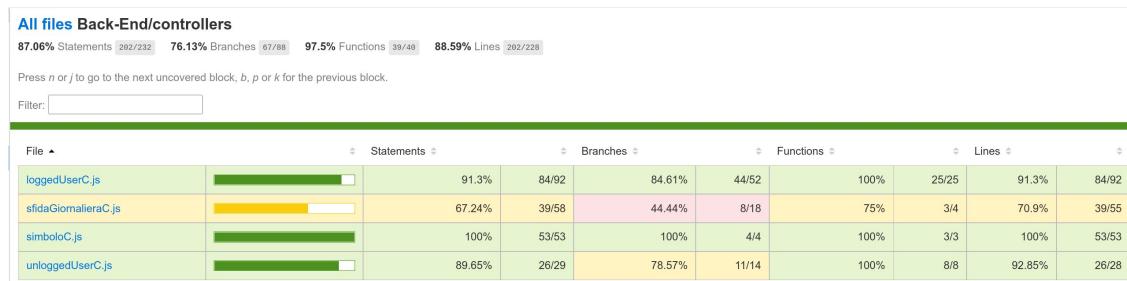


Figura 43: Rilsultato del testing

Come si può notare il file 'sfidaGiornalieraC.js' non ha avuto una gran copertura. Questo perchè ci sono due tipi di sfide giornaliere ed è possibile avere solo una sfida

giornaliera per volta. Se la sfida è già presente nel database, viene restituita quella già presente non ne viene generata una nuova ogni volta.