

Ingegneria del Software

Dipartimento di Ingegneria e Scienza dell'Informazione

Gruppo G-51: Giuliano Campagnolo, Elisa Beltrame, Mattia Marini



Mergify - merge your events

Deliverable 4

Analisi della struttura del codice



Indice

1 Scopo del documento	3
2 User flow	3
3 Application implementation and documentation	4
3.1 Project Structure	5
3.2 Project Dependencies	7
3.3 Project Data or DB	7
4 Project APIs	10
4.1 Estrazione delle risorse del Class Diagram	10
4.2 Diagramma delle risorse	12
4.3 Sviluppo API	16
4.4 Gestione delle collezioni	17
4.5 Middleware	25
4.6 Utility per Gestione Password, Debug e Funzioni Ausiliarie	35
4.7 API per la gestione degli utenti	38
4.8 API per la gestione dei gruppi	42
4.9 API per la gestione delle relazioni tra utenti e gruppi	45
4.10 API per la gestione del calendario	46
4.11 API per la gestione dello stato del sistema	48
5 API documentation	50
6 FrontEnd implementation	53
6.1 Login	54
6.2 Signup	54
6.3 My cal	55
6.4 My groups	56
6.5 Info utente	57

7 Testing	58
8 Github repository e informazioni sul deployment	61

1 Scopo del documento

Il presente documento fornisce un'ampia panoramica sullo sviluppo di una componente significativa dell'applicazione web Mergify.

Nel primo capitolo, viene presentato lo user flow, rappresentato attraverso un diagramma in cui vengono descritte tutte le azioni eseguibili nulla parte implementata di Mergify. Questo include una descrizione dettagliata delle richieste front-end su ciascuna pagina e delle possibili risposte.

A seguire, viene fornita un'analisi della struttura del codice realizzato, comprendente le dipendenze installate, i modelli sviluppati e le API implementate.

Un'attenzione particolare è dedicata alla descrizione delle API implementate, attraverso l'uso del diagramma delle risorse e del diagramma di estrazione, che identificano gli elementi estratti a partire dal diagramma delle classi.

Nel quarto capitolo è dettagliato il processo di documentazione delle API. Successivamente, vengono fornite brevi descrizioni delle pagine implementate e del repository di GitHub, accompagnate da istruzioni per il deployment.

In seguito, viene descritta la parte front-end dell'applicazione, con un'analisi di ciascuna pagina implementata.

Infine, vengono presentati diversi casi di test sviluppati per verificare il corretto funzionamento delle API, completando così l'analisi e lo sviluppo dell'applicazione web Mergify.

2 User flow

In questa sezione viene illustrato lo user flow dell'applicazione, che descrive in dettaglio le funzionalità disponibili nell'implementazione presentata in questo documento.

Di seguito è fornita una legenda che illustra i diversi componenti utilizzati nello user-flow.



Figura 1: Legenda dello user flow

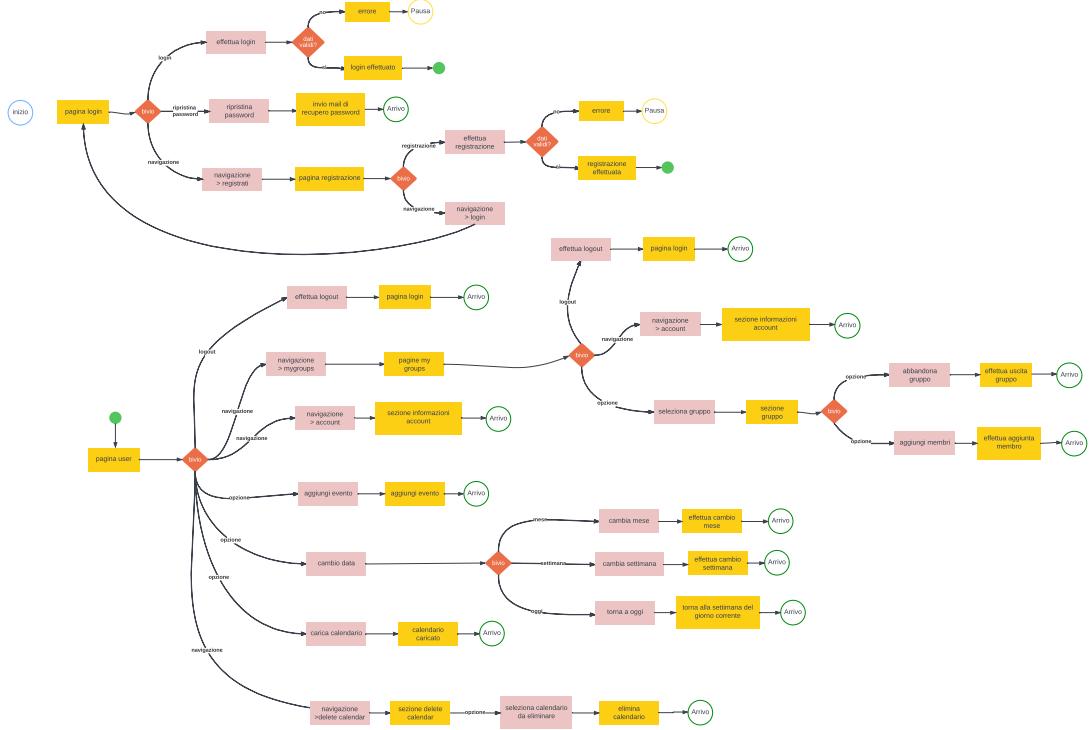


Figura 2: User flow

3 Application implementation and documentation

L'applicazione Mergify è stata realizzata impiegando NodeJS per il front-end, mentre per il back-end è stato adottato MongoDB per la memorizzazione dei dati. Come evidenziato nello user flow, sono state implementate tutte le funzionalità relative alla registrazione, al login, alla creazione e gestione degli eventi, al caricamento di un calendario, nonché alla navigazione tra il gruppo e il profilo personale.

Il codice è diviso in tre repository:

- frontend: contiene il front-end dell'utente;
- admin-dashboard: contiene il front-end di un utente admin (serve come testing per determinate API);
- backend: il back-end dell'applicazione.

3.1 Project Structure

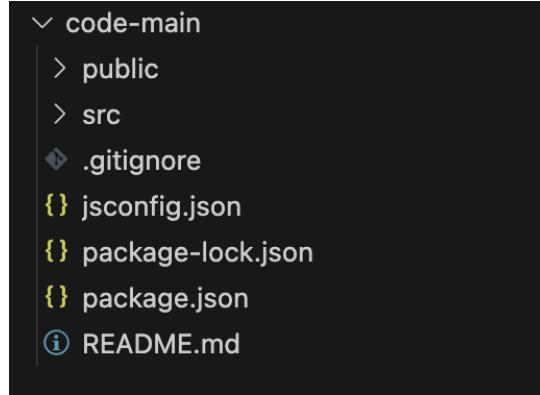


Figura 3: Struttura del progetto

è composta dai seguenti file e cartelle:

- Cartella Public;
- Cartella src;
- File .env per le variabili d'ambiente.

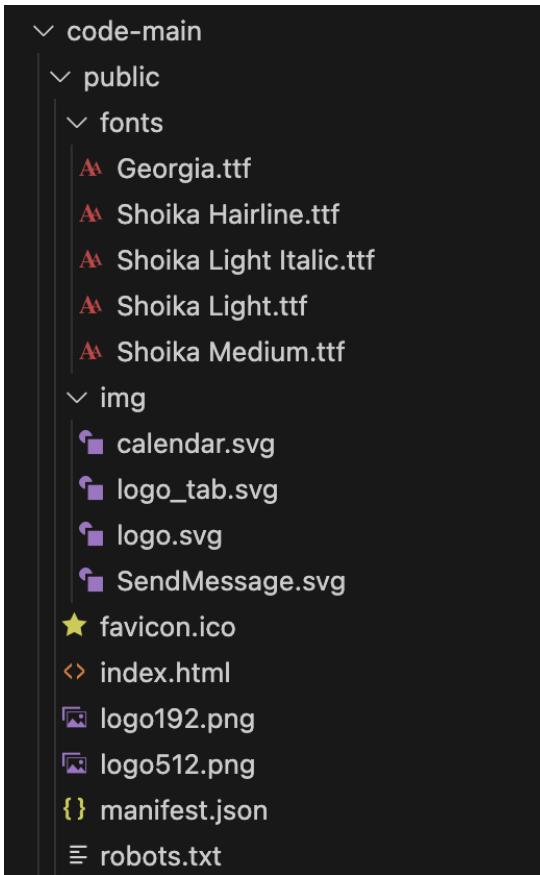


Figura 4: Cartella Public

La cartella public è formata dai seguenti file e cartelle:

- Cartella fonts, include i font usati nel sito;
- Cartella img, raccoglie le immagini usate in svg;
- File index.html, il file HTML che definisce la struttura del sito;
- File favicon.ico, manifest.json, logo192.png, logo512.png, in cui sono racchiuse altre immagini e icone utili.

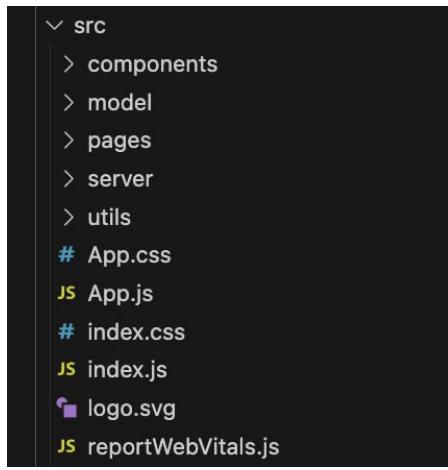


Figura 5: Cartella src

Il restante è implementato nella cartella src, che contiene le seguenti cartelle e file:

- Cartella components, che ospita vari componenti riutilizzabili nel codice;
- Cartella model, in cui sono racchiusi i modelli utilizzati nel codice;
- Cartella pages, contenente le diverse pagine dell'applicazione;
- Cartella server, che gestisce API;
- Cartella utils, in cui sono raccolti gli strumenti e le funzioni utili;
- Files App.js e App.css, le quali gestiscono le funzionalità e la presentazione delle pagine;
- Files Index.js e Index.css, che costituiscono il punto d'ingresso principale dell'applicazione.

3.2 Project Dependencies

I moduli Node utilizzati e aggiunti al file *package.json*, nel campo *dependencies*, sono:

- cors: modulo che consente alla web app di supportare il Cross-Origin Resource Sharing protocol (CORS).
- dotenv: permette di l'uso delle variabili d'ambiente definite nel file ‘.env’.
- express: framework che fornisce numerose funzionalità per la creazione e la gestione delle API.
- jsonwebtoken: modulo per la creazione e la gestione di un token d'accesso.
- mongoose: fornisce le funzioni necessarie per interagire con MongoDB.
- multer: gestisce il body delle richieste nelle API.
- nodemailer: utilizzato per l'invio di email agli utenti.
- swagger-ui-express: strumento per documentare e testare la API progettato.
- jest: modulo usato per il testing delle API e delle funzioni nel back-end.
- supertest: modulo per effettuare chiamate alle API in fase di testing.

```
"dependencies": {  
    "bcryptjs": "^2.4.3",  
    "cookie-parser": "^1.4.6",  
    "cors": "^2.8.5",  
    "dotenv": "^16.4.5",  
    "express": "^4.19.2",  
    "jsonwebtoken": "^9.0.2",  
    "mongodb": "^6.8.0",  
    "supports-color": "^9.4.0",  
    "swagger-jsdoc": "^6.2.8",  
    "swagger-ui-express": "^5.0.1"  
},
```

Figura 6: Dipendenze del progetto

3.3 Project Data or DB

Per la gestione dei dati utili all'applicazione si è definito:

- User;
- Group;

- Calendar;
- Event.

3.3.1 Modello *User*

Per memorizzare i dati degli utenti della nostra web app è stato creato il modello ‘User’:

```
export default class User {
  constructor(initializer) {

    this.name = "Default name"
    this.surname = "Default surname"
    this.emailAddress = "Default mail"
    this.password = "Default password"
    this.calendar = new Calendar()
    this.creationDate = new Date()
    this.groups = {};

    if (typeof initializer == "object") {
      for (let key in initializer) {
        if (this[key] != undefined)
          this[key] = initializer[key]
      }
    }
  }

  toJSON() {
    return {
      name: this.name,
      surname: this.surname,
      emailAddress: this.emailAddress,
      creationDate: this.creationDate.toISOString(), // Format date as string
      groups: {} // Add additional fields if needed
    }
  }
}
```

Figura 7: Modello User

3.3.2 Modello *Group*

Il modello ‘Group’ è stato utilizzato per memorizzare i dati relativi ai diversi gruppi:

```

export default class Group {

  constructor(name, leaderID, description = "No description", users = {}, date = new Date()) {
    this.name = name
    this.leaderID = leaderID
    this.description = description
    this.creationDate = date
    this.users = users
    const cal = new Calendar()
    cal.events = { default: [] }
    this.calendar = cal
  }

  toJSON() {
    return {
      name: this.name,
      leaderID: this.leaderID,
      description: this.description,
      creationDate: this.creationDate.toISOString(), // Convert date to ISO string
      users: this.users, // Ensure users is an array or appropriate type
    };
  }
}

```

Figura 8: Modello Group

3.3.3 Modello *Event*

Relativamente agli eventi, è stato definito il modello ‘Event’:

```

class Event {

  constructor(startDate, endDate, description = "No desc") {
    this.startDate = new Date(startDate)
    this.endDate = new Date(endDate)
    this.description = description
  }
}

```

Figura 9: Modello Event

3.3.4 Modello *Calendar*

Infine per i calendari si è servito del modello Calendar:

```

export default class Calendar {

    name = "Unnamed cal collection"
    events = {
        default: [
            new Event(2023, 11, 1, 0, 0, 1, 0, "Evento 1"),
            new Event(2023, 11, 1, 2, 0, 3, 0, "Evento 1"),
            new Event(2023, 11, 1, 4, 0, 7, 0, "Evento 1"),
            new Event(2023, 11, 1, 9, 0, 10, 0, "Riunione"),
            new Event(2023, 11, 2, 4, 0, 5, 0, "Compleanno"),
            new Event(2023, 11, 5, 8, 0, 10, 0, "Boh"),
            new Event(2023, 11, 6, 5, 0, 10, 0, "2"),
            new Event(2023, 11, 7, 2, 30, 8, 30, "desc"),
        ],
    };
}

constructor(collectionName) {
    if (collectionName)
        this.name = collectionName
}

```

Figura 10: Modello Calendar

4 Project APIs

Questo capitolo analizza le API sviluppate e utilizzate durante la scrittura del codice.

4.1 Estrazione delle risorse del Class Diagram

- **Groups:** Memorizza i gruppi;
- **Users:** Memorizza gli utenti.

La risorsa User contiene i seguenti attributi: name, surname, emailAddress, password, calendar, creationDate, groups.

La risorsa implementa le seguenti API:

- getUserId;
- deleteUser;
- updateUserProperty;
- createUser;
- loginUser;
- logout.

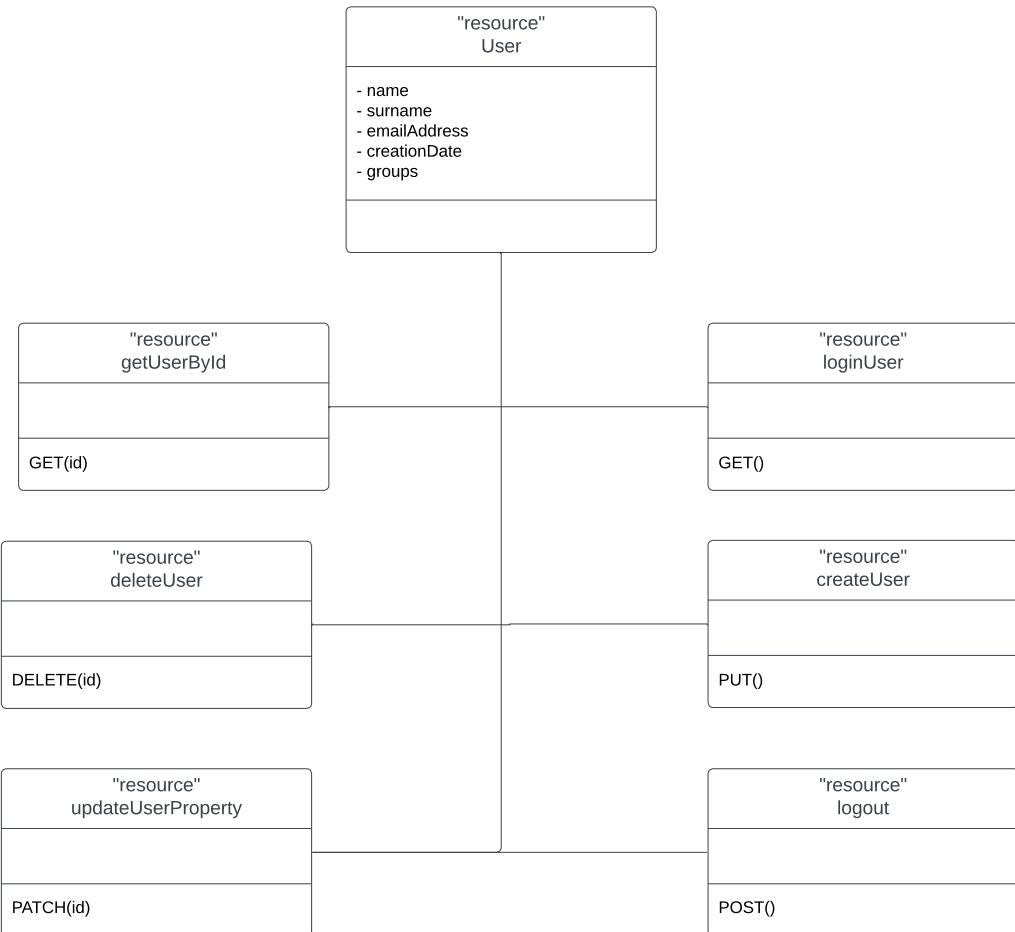


Figura 11: Diagramma delle risorse User

La risorsa Groups contiene i seguenti attributi: name, leaderID, description, creationDate, users, calendar.

La risorsa Groups implementa le seguenti API:

- getGroupbyId;
- deleteGroup;
- updateGroupProperty;
- createGroup.

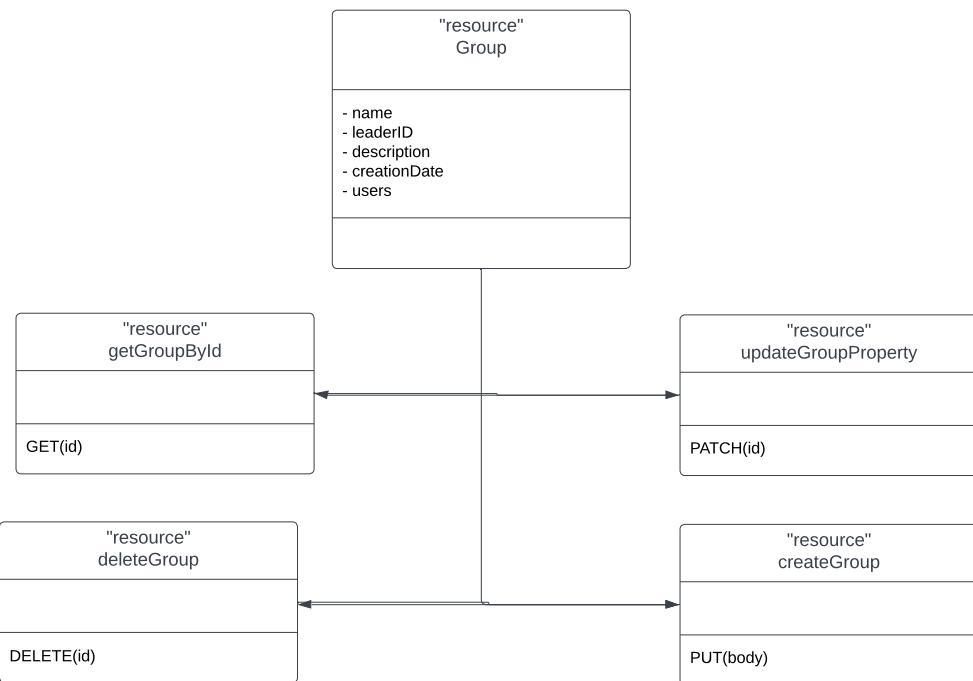


Figura 12: Diagramma delle risorse Groups

4.2 Diagramma delle risorse

Il Diagramma delle risorse illustrato rappresenta una specifica del processo di Resources Extraction. Per ottimizzare chiarezza e modularità, il diagramma è suddiviso in cinque sezioni. Oltre alle API principali del progetto, sono incluse alcune API aggiuntive necessarie per garantire il corretto funzionamento delle API principali.

Ogni diagramma illustra gli input e gli output delle diverse API. Poiché le API possono generare vari tipi di errori, è stato adottato un formato di output standard che include anche il codice dell'errore. Analogamente, per le API che restituiscono esclusivamente un codice di conferma, l'output che riflette tale funzionalità.

4.2.1 Gestione User

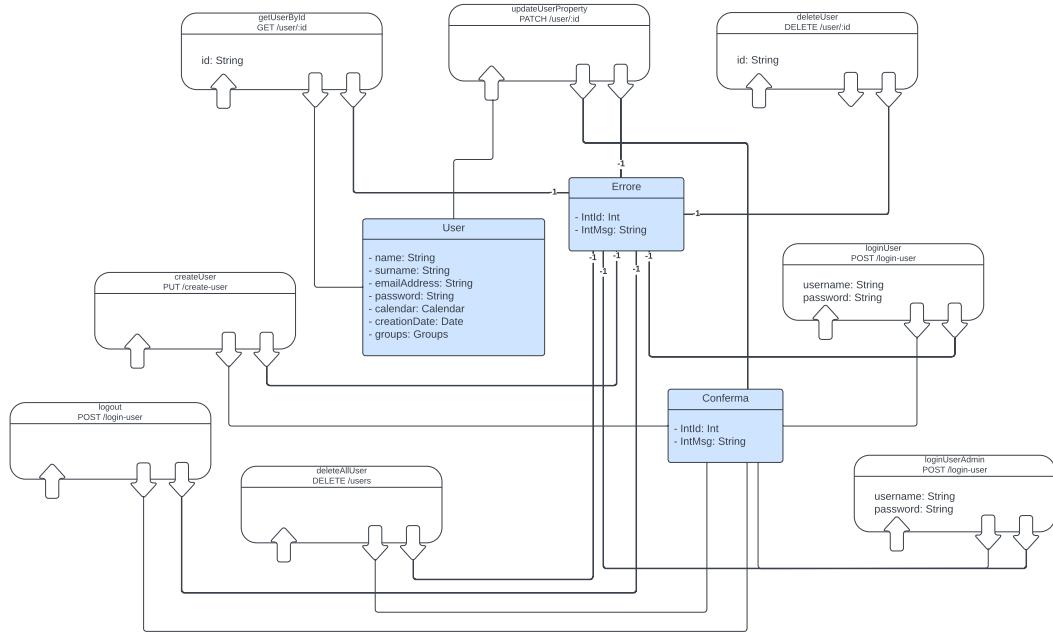


Figura 13: Gestione User

4.2.2 Gestione Group

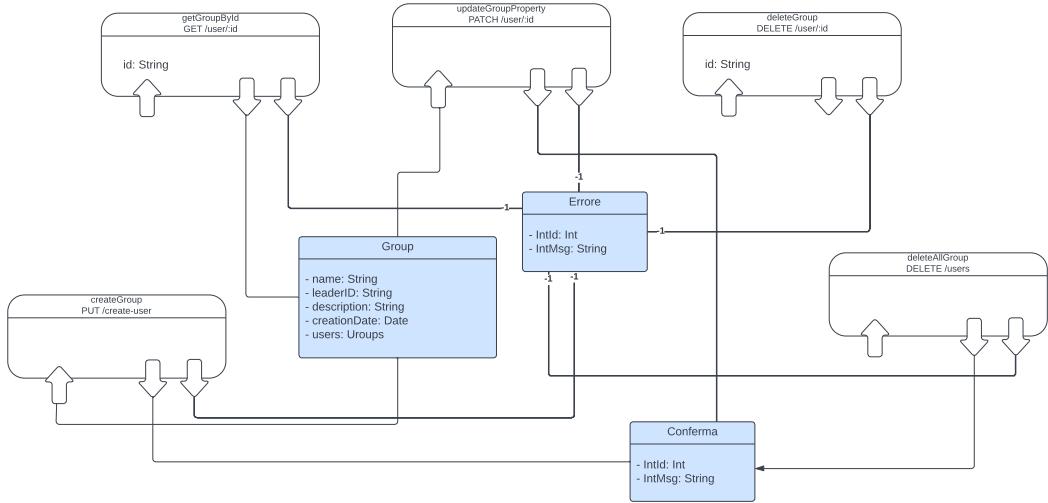


Figura 14: Gestione Group

4.2.3 Gestione Calendario

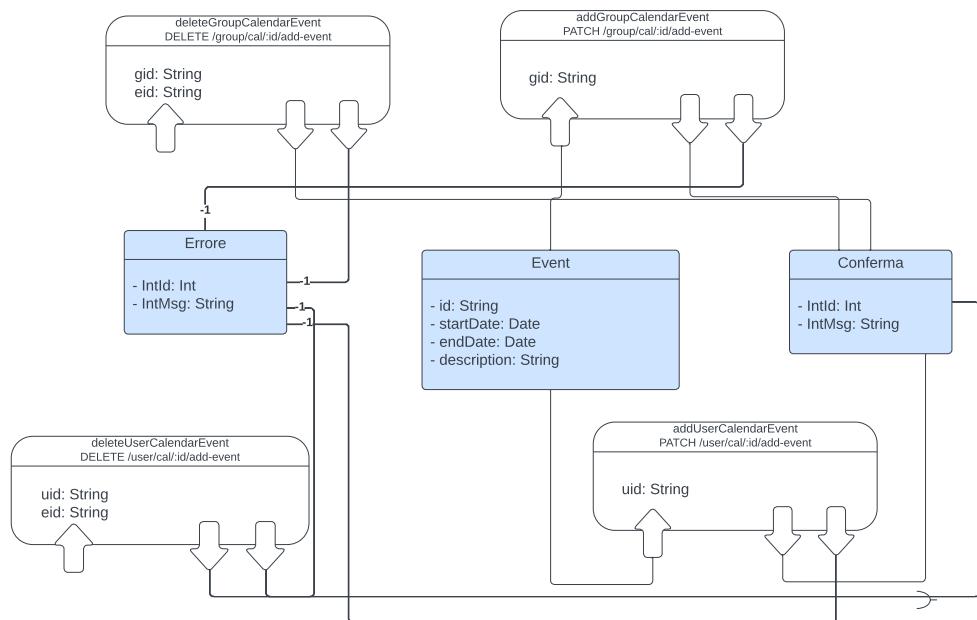


Figura 15: Gestione Calendario

4.2.4 Gestione Link

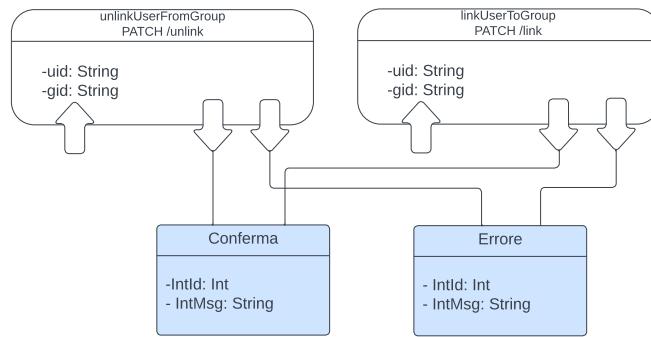


Figura 16: Gestione Link

4.2.5 Gestione Sistema

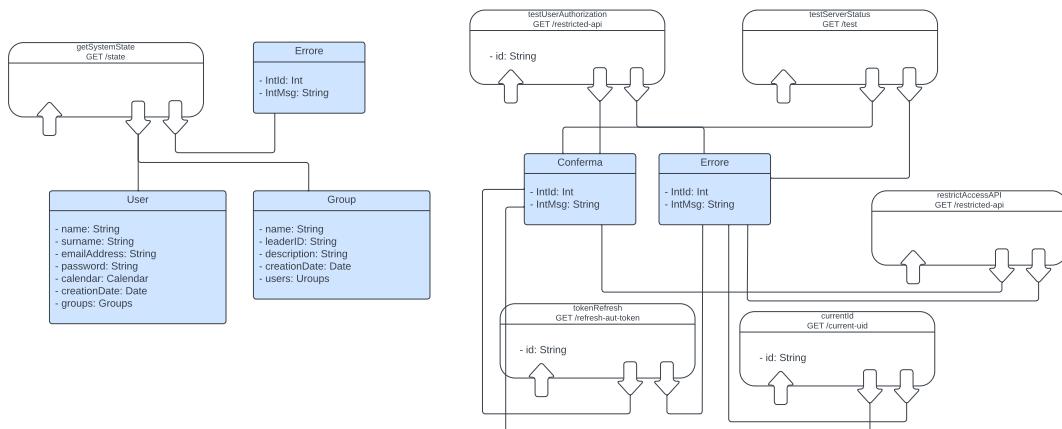


Figura 17: Gestione Sistema

4.3 Sviluppo API

In questa sezione viene descritta le API implementate e il loro funzionamento.

Per una comprensione completa, è opportuno iniziare con la presentazione delle funzioni ausiliarie, organizzate come segue:

- Gestione delle collezioni: per la gestione delle collection;
- Middleware: per la gestione delle autenticazioni all'interno dell'API;
- Utility: per la gestione delle password, il debug e altre funzioni ausiliarie;

In aggiunta è presente un file ‘returnCodes.mjs’ che contiene tutti i messaggi di ritorno, inclusi errori e conferme.

Successivamente, verranno presentate le API implementate.

4.4 Gestione delle collezioni

Le API fanno uso di specifiche funzioni per estrarre i dati dalle collezioni, le quali sono definite nel file ‘Db.mjs’. Le funzioni incluse sono le seguenti:

- **connect()**: Stabilisce la connessione al database MongoDB utilizzando MongoClient. Inizializza le collezioni per gruppi, utenti, amministratori e token di accesso.

```
static async connect() {  
    this.client = new MongoClient(process.env.URI);  
    await this.client.connect().catch((error) => console.log(error))  
    this.db = this.client.db(process.env.DB_NAME);  
    this.groupsCollection = this.db.collection('Groups');  
    this.usersCollection = this.db.collection('Users');  
    this.adminsCollection = this.db.collection('Admins');  
    this.accessTokenCollection = this.db.collection('AdminAccessTokens');  
}
```

Figura 18: Funzione connect()

- **getUserById(uid)**: ricerca un utente dal database MongoDB cercandolo per ID. Se l'ID non è valido, ritorna ‘false’.
- **getUserByMail(email)**: trova un utente dal database MongoDB cercandolo per indirizzo email.
- **getAdmin(username)**: recupera un amministratore cercandolo per nome utente.
- **getGroupById(gid)**: estrae un gruppo dal database MongoDB cercandolo per ID. Se l'ID non è valido, ritorna ‘null’.

- **getGroupByName(name):** restituisce un gruppo dal database MongoDB cercandolo per nome.

```
static async connect() {
  this.client = new MongoClient(process.env.URI);
  await this.client.connect().catch((error) => console.log(error))
  this.db = this.client.db(process.env.DB_NAME);
  this.groupsCollection = this.db.collection('Groups');
  this.usersCollection = this.db.collection('Users');
  this.adminsCollection = this.db.collection('Admins');
  this.accessTokenCollection = this.db.collection('AdminAccessTokens');
}
```

Figura 19: Funzione getGroupByName()

- **deleteUser(uid):** elimina un utente dal database MongoDB cercandolo per ID. Se l'ID non è valido o l'utente non esiste, ritorna 'null'. Inoltre, aggiorna i gruppi a cui l'utente appartiene.

```
static async deleteUser(uid) {
  let objectId = null
  try { objectId = ObjectId.createFromHexString(uid) }
  catch (error) { return null }

  const user = await this.getUserById(uid)
  if (!user)
    return null

  const promises = []
  for (let gid in user.groups) {
    let goid
    try { goid = ObjectId.createFromHexString(gid) }
    catch (error) { dprint(`Found invalid group id: ${gid}`, { color: "YELLOW" }) }

    promises.push(this.groupsCollection.updateOne(
      { _id: goid },
      { $unset: { [`users.${uid}`]: "" } }
    ))
  }

  await Promise.all(promises)

  return this.usersCollection.deleteOne({ _id: objectId });
}
```

Figura 20: Funzione deleteUser()

- **deleteGroup(gid):** elimina un gruppo dal database MongoDB cercandolo per ID. Se l'ID non è valido o il gruppo non esiste, ritorna ‘null’. Inoltre aggiorna gli utenti che appartengono al gruppo.

```

static async deleteGroup(gid) {
  let objectId = null
  try { objectId = ObjectId.createFromHexString(gid) }
  catch (error) { return null }

  const group = await this.getGroupById(gid)
  if (!group)
    return null

  const promises = []
  for (let uid in group.users) {
    let uoid
    try { uoid = ObjectId.createFromHexString(uid) }
    catch (error) { dprint(`Found invalid user id: ${uid}`, { color: "YELLOW" }) }

    promises.push(this.usersCollection.updateOne(
      { _id: uoid },
      { $unset: { [`groups.${gid}`]: "" } }
    ))
  }

  await Promise.all(promises)

  console.log(gid)
  return this.groupsCollection.deleteOne({ _id: objectId });
}

```

Figura 21: Funzione deleteGroup()

- **patchUser(uid, obj):** aggiorna le informazioni di un utente specifico nel database MongoDB. Ritorna ‘true’ se l’aggiornamento ha avuto successo, altrimenti ‘false’.
- **patchGroup(gid, obj):** aggiorna le informazioni di un gruppo specifico nel database MongoDB. Ritorna ‘true’ se l’aggiornamento ha avuto successo, altrimenti ‘false’.

```

static async patchUser(uid, obj) {

    let objectId = null
    try { objectId = ObjectId.createFromHexString(uid) }
    catch (error) { return false }

    const rv = await this.usersCollection.updateOne(
        { _id: objectId },
        { $set: obj }
    )

    return rv.modifiedCount != 0
}

static async patchGroup(gid, obj) {

    let objectId = null
    try { objectId = ObjectId.createFromHexString(gid) }
    catch (error) { return false }

    const rv = await this.groupsCollection.updateOne(
        { _id: objectId },
        { $set: obj }
    )

    return rv.modifiedCount != 0
}

```

Figura 22: Funzione patchGroup()

- **addEventToUserCal(uid, collectionName, event):** aggiunge un evento al calendario di un utente specifico. Ritorna ‘true’ se l’operazione ha avuto successo, altrimenti ‘false’.
- **addEventToGroupCal(gid, collectionName, event):** aggiunge un evento al calendario di un gruppo specifico. Ritorna ‘true’ se l’operazione ha avuto successo, altrimenti ‘false’.

```

static async addEventToUserCal(uid, collectionName, event) {
    let objectId = null
    try { objectId = ObjectId.createFromHexString(uid) }
    catch (error) { return false }

    const rv = await this.usersCollection.updateOne(
        { _id: objectId },
        { $push: { [`calendar.events.${collectionName}`]: (new Event(event.startDate, event.endDate)).toJSON() } }
    )

    return rv.modifiedCount != 0
}

static async addEventToGroupCal(gid, collectionName, event) {
    let objectId = null
    try { objectId = ObjectId.createFromHexString(gid) }
    catch (error) { return false }

    const rv = await this.groupsCollection.updateOne(
        { _id: objectId },
        { $push: { [`calendar.events.${collectionName}`]: event } }
    )

    return rv.modifiedCount != 0
}

```

Figura 23: Funzione addEventToGroupCal()

- **removeEventFromUserCal(uid, collectionName, index):** rimuove un evento dal calendario di un utente specifico in base all’indice dell’evento. Ritorna ‘true’ se l’operazione ha avuto successo, altrimenti ‘false’.
- **removeEventFromGroupCal(gid, collectionName, index):** rimuove un evento dal calendario di un gruppo specifico in base all’indice dell’evento. Ritorna ‘true’ se l’operazione ha avuto successo, altrimenti ‘false’.

```

static async removeEventFromUserCal(uid, collectionName, index) {

    let objectId = null
    try { objectId = ObjectId.createFromHexString(uid) }
    catch (error) { return false }

    const rv = await this.usersCollection.updateOne(
        { _id: objectId },
        { $unset: { [`calendar.events.${collectionName}.${index}`]: 1 } }
    )

    return rv.modifiedCount != 0
}

static async removeEventFromGroupCal(gid, collectionName, index) {

    let objectId = null
    try { objectId = ObjectId.createFromHexString(gid) }
    catch (error) { return false }

    const rv = await this.groupsCollection.updateOne(
        { _id: objectId },
        { $unset: { [`calendar.events.${collectionName}.${index}`]: 1 } }
    )

    return rv.modifiedCount != 0
}

```

Figura 24: Funzione removeEventFromGroupCal()

- **removeLink(userId, groupId):** rimuove il collegamento tra un utente e un gruppo, aggiornando entrambe le collezioni. Ritorna ‘true’ se l’operazione ha avuto successo, altrimenti ‘false’.

```

static async removeLink(userId, groupId) {
    let uoid = null
    let goid = null
    try {
        uoid = ObjectId.createFromHexString(userId)
        goid = ObjectId.createFromHexString(groupId)
    }
    catch (error) { return false }

    await this.usersCollection.updateOne(
        { _id: uoid },
        { $unset: { [`groups.${goid}`]: "" } }
    );

    await this.groupsCollection.updateOne(
        { _id: goid },
        { $unset: { [`users.${uoid}`]: "" } }
    );
}

return true
}

```

Figura 25: Funzione removeLink()

- **addLink(userId, groupId)**: aggiunge un collegamento tra un utente e un gruppo, aggiornando entrambe le collezioni. Ritorna ‘true’ se l’operazione ha avuto successo, altrimenti ‘false’.

```

static async addLink(userId, groupId) {
    let uoid = null
    let goid = null
    try {
        uoid = ObjectId.createFromHexString(userId)
        goid = ObjectId.createFromHexString(groupId)
    }
    catch (error) { return false }

    await this.usersCollection.updateOne(
        { _id: uoid },
        { $set: { [`groups.${goid}`]: { joinedAt: new Date() } } }
    );

    await this.groupsCollection.updateOne(
        { _id: goid },
        { $set: { [`users.${uoid}`]: { joinedAt: new Date() } } }
    );
}

return true
}

```

Figura 26: Funzione addLink()

- **createGroup(name, leader, description, users)**: crea un nuovo gruppo nel database MongoDB. Se esiste un gruppo con lo stesso nome, ritorna ‘null’; altrimenti, procede con l’inserimento del nuovo gruppo.
- **createUser(user)**: crea un nuovo utente nel database MongoDB. Se è presente un utente con lo stesso indirizzo email, ritorna ‘null’, altrimenti procede con l’inserimento del nuovo utente.
- **checkAccessToken(accessToken)**: verifica un token di accesso per determinare il livello di privilegio associato. Restituisce ‘-1’ se il formato del token è errato, ‘null’ se il token non esiste, oppure il livello di privilegio se il token è valido.
- **resetUsersCollection()**: elimina tutti gli utenti dalla collezione Users e ritorna il numero di documenti eliminati.
- **resetGroupsCollection()**: elimina tutti i gruppi dalla collezione Groups e ritorna il numero di documenti eliminati.

```

    static async createGroup(name, leader, description, users) {
      const newGroup = new Group(name, leader, description, users);
      const existingGroup = await this.groupsCollection.findOne({ "name": name })
      if (existingGroup == null)
        return this.groupsCollection.insertOne(newGroup)
      else
        return null
    }

    static async createUser(user) {
      const newUser = new User(user);
      const existingUser = await this.getUserByMail(user.emailAddress)
      if (existingUser == null)
        return this.usersCollection.insertOne(newUser)
      else
        return null
    }

    static async checkAccessToken(accessToken) {
      if (typeof accessToken !== 'string' || accessToken.length != 24)
        return -1
      const rv = await this.accessTokenCollection.findOne({ _id: new ObjectId(accessToken) })
      if (rv == null) return null
      return rv.privilage
    }

    static async resetUsersCollection() {
      return (await this.usersCollection.deleteMany({})).deletedCount
    }

    static async resetGroupsCollection() {
      return (await this.groupsCollection.deleteMany({})).deletedCount
    }
}

```

Figura 27: Funzione resetGroupsCollection()

4.5 Middleware

Di seguito sono descritte le funzioni utilizzate all'interno dell'API per verificare le autorizzazioni e gestire la sicurezza nelle operazioni relative a utenti e gruppi, inclusi l'aggiunta, la rimozione e la modifica di eventi. Queste funzioni utilizzano token JWT per autenticare le richieste e controllare i privilegi degli utenti.

Tali funzioni sono situate alla fine del file “middleware.mjs”:

- **checkUserRemoveEventPriv(req, res, next):** verifica se l'utente possiede i privilegi necessari per rimuovere un evento dal proprio calendario e, successivamente, chiama il middleware successivo. Include anche un controllo dell'autenticazione.

```

// 2404 4402 4403 4404 4405
export function checkUserRemoveEventPriv(req, res, next) {

    const wrapper = async () => {
        if (!(req.body.index))
            return sendCustomRes(res, 4400)

        const user = await DB.getUserById(req.params.id)

        if (req.body.index < 0 || req.body.index >= user.calendar.events.length)
            return sendCustomRes(res, 4405)

        if (!user)
            return sendCustomRes(res, 1404)

        if (req.body.calendar && !group.calendar[req.body.calendar])
            return sendCustomRes(res, 4403)

        if (req.role == "admin" || user.uid == req.uid)
            return next()
        else
            return sendCustomRes(res, 4401)
    }

    checkAut(req, res, wrapper)
}

// 2404 4402 4403 4404 4405
export function checkGroupRemoveEventPriv(req, res, next) {

    const wrapper = async () => {

        if (!(req.body.index))
            return sendCustomRes(res, 4404)

        const group = await DB.getGroupById(req.params.id)

        if (req.body.index < 0 || req.body.index >= group.calendar.events.length)
            return sendCustomRes(res, 4405)

        if (!group)
            return sendCustomRes(res, 2404)

        if (req.body.calendar && !group.calendar[req.body.calendar])
            return sendCustomRes(res, 4403)

        if (req.role == "admin" || group.users[req.uid] != undefined)
            return next()
        else
            return sendCustomRes(res, 4402)
    }

    checkAut(req, res, wrapper)
}

```

Figura 28: Funzione checkUserRemoveEventPriv()

- **checkGroupRemoveEventPriv(req, res, next):** accerta se l'utente ha i privilegi per rimuovere un evento dal calendario di un gruppo, verificando sia l'autenticazione sia la validità dei dati forniti.

```

// 2404 4402 4403 4404 4405
export function checkUserRemoveEventPriv(req, res, next) {

  const wrapper = async () => {
    if (!(req.body.index))
      return sendCustomRes(res, 4400)

    const user = await DB.getUserById(req.params.id)

    if (req.body.index < 0 || req.body.index >= user.calendar.events.length)
      return sendCustomRes(res, 4405)

    if (!user)
      return sendCustomRes(res, 1404)

    if (req.body.calendar && !group.calendar[req.body.calendar])
      return sendCustomRes(res, 4403)

    if (req.role == "admin" || user.uid == req.uid)
      return next()
    else
      return sendCustomRes(res, 4401)
  }

  checkAut(req, res, wrapper)
}

// 2404 4402 4403 4404 4405
export function checkGroupRemoveEventPriv(req, res, next) {

  const wrapper = async () => {
    if (!(req.body.index))
      return sendCustomRes(res, 4404)

    const group = await DB.getGroupById(req.params.id)

    if (req.body.index < 0 || req.body.index >= group.calendar.events.length)
      return sendCustomRes(res, 4405)

    if (!group)
      return sendCustomRes(res, 2404)

    if (req.body.calendar && !group.calendar[req.body.calendar])
      return sendCustomRes(res, 4403)

    if (req.role == "admin" || group.users[req.uid] != undefined)
      return next()
    else
      return sendCustomRes(res, 4402)
  }

  checkAut(req, res, wrapper)
}

```

Figura 29: Funzione checkGroupRemoveEventPriv()

- **checkUserAddEventPriv(req, res, next)**: valuta se l'utente ha i privilegi per aggiungere un evento al proprio calendario, controllando l'autenticità delle date e l'autenticazione.

```

// 4400 2404 2403 4402 4403
export function checkUserAddEventPriv(req, res, next) {

    const wrapper = async () => {
        if (!(req.body.startDate && req.body.endDate))
            return sendCustomRes(res, 4400)
        if (!(isValidDate(new Date(req.body.startDate)) && isValidDate(new Date(req.body.endDate))))
            return sendCustomRes(res, 4400)

        const user = await DB.getUserById(req.params.id)

        if (!user)
            return sendCustomRes(res, 1404)

        if (req.body.calendar && !group.calendar[req.body.calendar])
            return sendCustomRes(res, 4403)

        if (req.role == "admin" || user.uid == req.uid)
            return next()
        else
            return sendCustomRes(res, 4401)
    }

    checkAut(req, res, wrapper)
}

// 4400 2404 2403 4402 4403
export function checkGroupAddEventPriv(req, res, next) {

    const wrapper = async () => {
        if (!(req.body.startDate && req.body.endDate))
            return sendCustomRes(res, 4400)
        if (!(isValidDate(new Date(req.body.startDate)) && isValidDate(new Date(req.body.endDate))))
            return sendCustomRes(res, 4400)

        const group = await DB.getGroupById(req.params.id)

        if (!group)
            return sendCustomRes(res, 2404)

        if (req.body.calendar && !group.calendar[req.body.calendar])
            return sendCustomRes(res, 4403)

        if (req.role == "admin" || group.users[req.uid] != undefined)
            return next()
        else
            return sendCustomRes(res, 4402)
    }

    checkAut(req, res, wrapper)
}

```

Figura 30: Funzione checkUserAddEventPriv()

- **checkGroupAddEventPriv(req, res, next):** determina se l'utente ha i privilegi necessari per aggiungere un evento al calendario di un gruppo, verificando l'autenticità delle date e l'autenticazione.

```

// 4400 2404 2403 4402 4403
export function checkUserAddEventPriv(req, res, next) {

    const wrapper = async () => {
        if (!(req.body.startDate && req.body.endDate))
            return sendCustomRes(res, 4400)
        if (!(isValidDate(new Date(req.body.startDate)) && isValidDate(new Date(req.body.endDate))))
            return sendCustomRes(res, 4400)

        const user = await DB.getUserById(req.params.id)

        if (!user)
            return sendCustomRes(res, 1404)

        if (req.body.calendar && !group.calendar[req.body.calendar])
            return sendCustomRes(res, 4403)

        if (req.role == "admin" || user.uid == req.uid)
            return next()
        else
            return sendCustomRes(res, 4401)
    }

    checkAut(req, res, wrapper)
}

// 4400 2404 2403 4402 4403
export function checkGroupAddEventPriv(req, res, next) {

    const wrapper = async () => {
        if (!(req.body.startDate && req.body.endDate))
            return sendCustomRes(res, 4400)
        if (!(isValidDate(new Date(req.body.startDate)) && isValidDate(new Date(req.body.endDate))))
            return sendCustomRes(res, 4400)

        const group = await DB.getGroupById(req.params.id)

        if (!group)
            return sendCustomRes(res, 2404)

        if (req.body.calendar && !group.calendar[req.body.calendar])
            return sendCustomRes(res, 4403)

        if (req.role == "admin" || group.users[req.uid] != undefined)
            return next()
        else
            return sendCustomRes(res, 4402)
    }

    checkAut(req, res, wrapper)
}

```

Figura 31: Funzione checkGroupAddEventPriv()

- **checkAut(req, res, next):** middleware che verifica la presenza e l'autenticità del token di autenticazione (JWT) nell'intestazione della richiesta. Se il token è valido, estrae l'ID utente e il ruolo; altrimenti, restituisce un errore.
- **isAdmin(req, res, next):** verifica se l'utente dispone di privilegi di amministrazione.

struttore. Se sì, prosegue con il middleware successivo; altrimenti, restituisce un errore.

```
// Filter that checks the presence and authenticity of the aut token
export function checkAut(req, res, next) {
  const authHeader = req.headers['authorization']
  const token = authHeader && authHeader.split(' ')[1]
  if (token == null)
    return sendCustomRes(res, "450")

  jwt.verify(token, process.env.AUT_PRIVATE_KEY, (err, data) => {
    if (err)
      return sendCustomRes(res, "452")
    else {
      const { uid, role } = data
      if (!(uid && role))
        return sendCustomRes(res, "451")
      req.uid = uid
      req.role = role
      return next()
    }
  })
}

// Verifies if admin privileges are present.
export function isAdmin(req, res, next) {

  const wrapper = () => {
    if (req.role == "admin")
      return next()
    else
      return sendCustomRes(res, 453)
  }

  checkAut(req, res, wrapper) // ensure that checkAut is called before this middleware, so that infos about user authentication are available
}
```

Figura 32: Funzione isAdmin()

- **loginUser(req, res, next)**: middleware per il login dell'utente. Confronta le credenziali fornite (username e password) con quelle memorizzate nel database. Se le credenziali sono valide, prosegue al middleware successivo.
- **loginAdmin(req, res, next)**: middleware per il login dell'amministratore. Verifica le credenziali di accesso con quelle memorizzate nel database. Se le credenziali sono corrette, procede al middleware successivo.

```

// middleware to control if user credentials are right
export async function loginUser(req, res, next) {

    const username = req.body.username
    const password = req.body.password

    if (!(username && password))
        return sendCustomRes(res, "402")

    const user = await DB.getUserByMail(username)

    if (user && comparePasswordWithHash(password, user.password)) {
        req.uid = user._id
        next()
    }
    else
        return sendCustomRes(res, "401")

}

// middleware to control if admin credentials are right
export async function loginAdmin(req, res, next) {
    // TODO accesso a mongodb invece che dizionario

    const username = req.body.username
    const password = req.body.password
    if (!(username && password))
        return sendCustomRes(res, "402")

    const admin = await DB.getAdmin(username)

    if (admin && comparePasswordWithHash(password, admin.password)) {
        req.uid = admin._id
        next()
    }
    else
        return sendCustomRes(res, "401")

}

```

Figura 33: Funzione loginAdmin()

- **checkUserDeletionPriv(req, res, next)**: controlla se l'utente ha i privilegi necessari per eliminare un altro utente. Verifica l'autenticazione e i diritti dell'utente.
- **checkGruopDeletionPriv(req, res, next)**: verifica se l'utente ha i privilegi per eliminare un gruppo. Controlla se l'utente è il leader del gruppo o un

amministratore.

```
// middleware to check if user deletion requisites are present
export function checkUserDeletionPriv(req, res, next) {
  const wrapper = async () => {
    const uid = req.params.id;
    if (!uid)
      return sendCustomRes(res, "1400");

    const user = await DB.getUserById(uid);
    if (!user)
      return sendCustomRes(res, "1401");

    if (req.role == "admin" || req.uid == uid)
      next();
    else
      return sendCustomRes(res, "1402");
  }

  checkAut(req, res, wrapper) // ensure that checkAut is called before this middleware, so that infos about user authentication are available
}

// middleware to check if group deletion requisites are present
export function checkGroupDeletionPriv(req, res, next) {
  const wrapper = async () => {
    const gid = req.params.id;
    if (!gid)
      return sendCustomRes(res, "2400");

    const group = await DB.getGroupById(gid);
    if (!group)
      return sendCustomRes(res, "2401");

    if (req.role == "admin" || group.leaderID == req.uid)
      next();
    else
      return sendCustomRes(res, "2402");
  }

  checkAut(req, res, wrapper) // ensure that checkAut is called before this middleware, so that infos about user authentication are available
}
```

Figura 34: Funzione checkGruopDeletionPriv()

- **checkLinkPriv(req, res, next):** verifica i privilegi necessari per aggiungere un utente a un gruppo. Controlla la validità dell'utente e del gruppo e verifica l'autenticazione.
- **checkUnlinkPriv(req, res, next):** controlla i privilegi necessari per rimuovere un utente da un gruppo. Verifica che l'utente faccia parte del gruppo e controlla l'autenticazione.

```

// checks privileges needed to add user to group
export function checkLinkPriv(req, res, next) {

  const wrapper = async () => {

    const { userId, groupId } = req.body;

    if (!(userId && groupId))
      return sendCustomRes(res, "3400");
    const user = await DB.getUserById(userId);
    if (!user)
      return sendCustomRes(res, "3400");

    const group = await DB.getGroupById(groupId);
    if (!group)
      return sendCustomRes(res, "3400");

    if (req.role == "admin" || user.uid == req.uid)
      next();
    else
      return sendCustomRes(res, "3401");
  }

  checkAut(req, res, wrapper) // ensure that checkAut is called before this middleware, so that infos about user authentication are available
}

// checks privileges needed to remove user to group
export function checkUnlinkPriv(req, res, next) {

  const wrapper = async () => {

    const { userId, groupId } = req.body;

    if (!(userId && groupId))
      return sendCustomRes(res, "3400");

    const user = await DB.getUserById(userId);
    if (!user)
      return sendCustomRes(res, "3400");

    const group = await DB.getGroupById(groupId);
    if (!group)
      return sendCustomRes(res, "3400");

    if (!(group.users[user._id] && user.groups[group._id]))
      return sendCustomRes(res, "3403");

    if (req.role == "admin" || user._id == req.uid || group.leader == req.uid)
      next();
    else
      return sendCustomRes(res, "3401");
  }

  checkAut(req, res, wrapper) // ensure that checkAut is called before this middleware, so that infos about user authentication are available
}

```

Figura 35: Funzione checkUnlinkPriv()

- **checkGroupPropertyModifiable(req, res, next):** verifica se l'utente ha i privilegi necessari per modificare le proprietà di un gruppo. Controlla l'autenticazione e verifica che le modifiche richieste siano consentite.

```

// checks privileges needed to modify group property
// 2400 2405 2404 2406
export async function checkGroupPropertyModifiable(req, res, next) {

    const wrapper = async () => {
        const groupId = req.params.id;
        const reqBody = req.body

        if (!groupId)
            return sendCustomRes(res, 2400)

        if (Object.keys(reqBody).length === 0 || typeof reqBody !== "object")
            return sendCustomRes(res, 2405)

        const group = await DB.getGroupById(groupId)
        if (!group)
            return sendCustomRes(res, 2404)

        const checkForbiddenKeys = (obj, forbidden) => {
            let forbiddenKeys = []
            for (const key in obj)
                if (obj.hasOwnProperty(key) && (group[key] === undefined || forbidden.includes(key)))
                    forbiddenKeys.push(key)
            }
            return forbiddenKeys
        }

        // check aut
        if (req.role === "admin")
            return next()
        else if (group.leaderID === req.uid) {
            const forbiddenKeys = checkForbiddenKeys(reqBody, ["_id", "creationDate", "users"])
            if (forbiddenKeys.length === 0)
                return next()
            else
                return sendCustomRes(res, 2406, forbiddenKeys)
        }
        else
            return sendCustomRes(res, 2402)
    }

    checkAut(req, res, wrapper)
}

```

Figura 36: Funzione checkGroupPropertyModifiable()

- **checkUserPropertyModifiable(req, res, next):** controlla se l'utente ha i privilegi per modificare le proprietà del proprio profilo utente. Verifica l'autenticazione e controlla che le modifiche richieste sianomesse.

```

// checks privileges needed to modify user property
// 1400 1405 1404 1406
export async function checkUserPropertyModifiable(req, res, next) {

    const wrapper = async () => {
        const userId = req.params.id;
        const reqBody = req.body

        if (!userId)
            return sendCustomRes(res, 1400)

        if (Object.keys(reqBody).length === 0 || typeof reqBody != "object")
            return sendCustomRes(res, 1405)

        const userToModify = await DB.getUserById(userId)
        if (!userToModify)
            return sendCustomRes(res, 1404)

        const checkForbiddenKeys = (obj, forbidden) => {
            let forbiddenKeys = []
            for (const key in obj) {
                if (obj.hasOwnProperty(key) && (userToModify[key] == undefined || forbidden.includes(key))) {
                    forbiddenKeys.push(key)
                }
            }
            return forbiddenKeys
        }

        // check aut
        if (req.role == "admin")
            return next()
        else if (userToModify._id == req.uid) {
            const forbiddenKeys = checkForbiddenKeys(reqBody, ["_id", "creationDate", "groups"])
            if (forbiddenKeys.length == 0)
                return next()
            else
                return sendCustomRes(res, 1406, forbiddenKeys)
        }
        else
            return sendCustomRes(res, 1407)
    }

    checkAut(req, res, wrapper)
}

```

Figura 37: Funzione checkUserPropertyModifiable()

4.6 Utility per Gestione Password, Debug e Funzioni Ausiliarie

Di seguito funzioni di utility per la gestione delle password (hashing e confronto), la stampa condizionale di messaggi colorati per il debug e alcune funzioni ausiliarie per generare amministratori fittizi, controllare variabili d'ambiente e validare date.

Tali funzioni sono contenute nel file ‘utility.mjs’:

- **hashPassword(pwd):** restituisce l'hash della password (pwd) o ‘null’ se il processo di hashing fallisce.
- **comparePasswordWithHash(pwd, hash):** restituisce ‘true’ se la password (pwd) corrisponde all’hash fornito (hash), altrimenti ‘false’.
- **dprint(string, opts):** stampa un parametro string sulla console con formattazione e colori, se ‘opts.debuglvl’ è sufficientemente alto e se è abilitato il supporto per i colori.
- **generateRandomAdmin(username, password):** crea e restituisce un oggetto amministratore con un ID univoco, username e password hashata.
- **checkEnvVariables():** verifica se tutte le variabili d’ambiente necessarie sono definite e restituisce ‘true’ se lo sono tutte, restituisce ‘false’.
- **getCookieDomainFromUrl(url):** estrae e restituisce il dominio dalla URL fornita (url), gestendo i casi di URL locali e di dominio completo.
- **isValidDate(input):** controlla se input è una data valida. Restituisce ‘true’ se lo è, altrimenti ‘false’.
- **ANSIColors:** oggetto contenente codici di colore ANSI per la formattazione del testo nella console. Non è una funzione, ma un’utilità per dprint.

```

// returns the hash of pwd or null if hashing has failed
export function hashPassword(pwd) {
  const saltRounds = 10
  const salt = bcrypt.genSaltSync(saltRounds)
  try { return bcrypt.hashSync(pwd, salt) }
  catch (err) { return null }
}

// returns true if pwd and hash match
export function comparePasswordWithHash(pwd, hash) {
  try { return bcrypt.compareSync(pwd, hash) }
  catch (err) { return null }
}

export const ANSIColors = {
  RESET: "\x1b[0m",
  BOLD: "\x1b[1m",
  ITALIC: "\x1b[3m",
  UNDERLINE: "\x1b[4m",
  BLACK: "\x1b[30m",
  RED: "\x1b[31m",
  GREEN: "\x1b[32m",
  YELLOW: "\x1b[33m",
  BLUE: "\x1b[34m",
  MAGENTA: "\x1b[35m",
  CYAN: "\x1b[36m",
  WHITE: "\x1b[37m"
}

// this functions serves as a wrapper to the default console.log().
// 1. it prints only if debug is define
//
const debugPrintLvl = 2
export function dprint(string, opts) {
  if (debugPrintLvl < (opts.debuglvl == undefined ? 1 : opts.debuglvl)) return
  if (supportsColor.stdout && opts.color && ANSIColors[opts.color]) {
    if (typeof string == "string")
      console.log(ANSIColors[opts.color] + string + ANSIColors.RESET)
    else
      console.log(string)
    process.stdout.write(ANSIColors.RESET)
  }
  else
    console.log(string)
}

export function generateRandomAdmin(username, password) {
  return { _id: new ObjectId(), username: username, password: hashPassword(password) }
}

```

Figura 38: Utility per Gestione Password, Debug e Funzioni Ausiliarie

```

export function checkEnvVariables() {
  const envVars = ["URI", "USERNAME", "PASSWORD", "PORT", "CLUSTER_NAME", "DB_NAME", "REFRESH_PRIVATE_KEY", "AUT_PRIVATE_KEY", "CURRENT_DOMAIN"]

  let rv = true
  envVars.map(x => {
    if (process.env[x] == undefined)
      rv = false
  })
  return rv
}

export function getCookieDomainFromUrl(url) {
  let cleanedUrl = url.replace(/(^w+:|^)\/\//, '')
  if (cleanedUrl.includes("localhost"))
    return cleanedUrl.split(":")[0]
  else
    return cleanedUrl
}

export function isValidDate(input) {
  if (!(input && input.getTimezoneOffset && input.setUTCFullYear))
    return false;

  var time = input.getTime();
  return time === time;
};

```

Figura 39: Ulteriori Utility

4.7 API per la gestione degli utenti

- **getUserById:** Recupera i dettagli di un utente specifico.

```

// Gets user by id
this.app.get('/user/:id', async (req, res) => {
  dprint(`GET -> user/${req.params.id}`, { color: "GREEN" })

  try {
    const userId = req.params.id;
    const user = await DB.getUserById(userId);

    if (user)
      sendCustomRes(res, "1203", user)
    else
      sendCustomRes(res, "1404")

  } catch (error) {
    sendCustomRes(res, "-1", error)
  }
});

```

Figura 40: Funzione getUserById()

- **updateUserProperty:** Aggiorna una proprietà specifica di un utente

```

this.app.patch('/user/:id', checkUserPropertyModifiable, async (req, res) => {
  dprint(`PATCH -> user/${req.params.id}`, { color: "YELLOW" })
  try {
    const groupId = req.params.id;
    const reqBody = req.body;
    if (await DB.patchUser(groupId, reqBody) == true)
      sendCustomRes(res, 1204)
    else
      sendCustomRes(res, -1, "DB error")
  } catch (error) {
    sendCustomRes(res, -1, error)
  }
});

```

Figura 41: Funzione updateUserProperty()

- **deleteUser:** Elimina un utente specifico

```

// deletes the specified user
this.app.delete('/user/:id', checkUserDeletionPriv, async (req, res) => {
  dprint(`DELETE -> user/${req.params.id}`, { color: "BLUE" })

  try {
    const rv = await DB.deleteUser(req.params.id)
    if (rv)
      return sendCustomRes(res, "1201")
    else
      return sendCustomRes(res, "1401")
  } catch (error) {
    return sendCustomRes(res, "-1", error)
  }
});

```

Figura 42: Funzione deleteUser()

- **createUser:** Crea un nuovo utente

```

// creates a user. request should have at least emailAddress and password fields specified
this.app.put('/create-user', async (req, res) => {
  dprint("PUT -> create-user", { color: "BLUE" })
  try {
    const { emailAddress, password } = req.body;
    if (!(emailAddress && password))
      return sendCustomRes(res, "402")
    const user = req.body
    user.password = hashPassword(user.password)

    const userId = await DB.createUser(user);
    if (userId == null)
      return sendCustomRes(res, "1403")
    else
      return sendCustomRes(res, "1200", userId.insertedId)
  } catch (error) {
    return sendCustomRes(res, "-1", error)
  }
});

```

Figura 43: Funzione createUser()

- **deleteAllUsers:** Elimina tutti gli utenti (solo per gli amministratori)

```

// Deletes all users. Requires admin token
this.app.delete('/users', isAdmin, async (req, res) => {
  dprint("DELETE -> users", { color: "BLUE" })
  try {

    const removedCount = await DB.resetUsersCollection()
    return sendCustomRes(res, "1250", removedCount)

  } catch (error) {
    res.status(500).send(error);
    return sendCustomRes(res, "-1")
  }
});

```

Figura 44: Funzione deleteAllUsers()

- **loginUser:** Autentica un utente e genera un token di refresh

```

// Returns a user refresh token with eat of 30 days *
this.app.post('/login-user', loginUser, async (req, res) => {
  dprint("POST -> login-user", { color: "BLUE" })

  try {
    const refreshToken = jwt.sign({ uid: req.uid, role: "user" }, process.env.REFRESH_PRIVATE_KEY)
    res.cookie('refreshToken', refreshToken, {
      /*           path: '/refresh-aut-token', */
      httpOnly: true, // Prevents JavaScript access
      secure: true, // Ensures the cookie is sent over HTTPS only
      maxAge: 24 * 60 * 60 * 1000 * 30, // 30 days in milliseconds
      path: '/refresh-aut-token',
      sameSite: 'None', // Prevents the cookie from being sent in cross-site requests
      domain: getCookieDomainFromUrl(process.env.CURRENT_DOMAIN)
    });
    return sendCustomRes(res, "250")
  }
  catch (error) {
    return sendCustomRes(res, "-1")
  }
});

```

Figura 45: Funzione loginUser()

- **loginUserAdmin:** Come loginUser ma autentica l'admin

```

// Returns a admin refresh token with eat of 30 days *
this.app.post('/login-admin', loginAdmin, async (req, res) => {
  dprint("POST -> login-admin", { color: "BLUE" })

  try {
    const refreshToken = jwt.sign({ uid: req.uid, role: "admin" }, process.env.REFRESH_PRIVATE_KEY)
    res.cookie('refreshToken', refreshToken, {
      // sameSite: 'strict', // Prevents the cookie from being sent in cross-site requests
      // secure: true, // Ensures the cookie is sent over HTTPS only
      maxAge: 24 * 60 * 60 * 1000 * 30, // 30 days in milliseconds
      secure: true,
      httpOnly: true, // Prevents JavaScript access
      path: '/refresh-aut-token',
      sameSite: 'None', // Prevents the cookie from being sent in cross-site requests
      domain: getCookieDomainFromUrl(process.env.CURRENT_DOMAIN)
    });
    return sendCustomRes(res, "250")
  }
  catch (error) {
    return sendCustomRes(res, "-1")
  }
});

```

Figura 46: Funzione loginUserAdmin()

- **logout:** Annulla la sessione dell'utente eliminando il token di aggiornamento

```
this.app.post('/logout', async (req, res) => {
  dprint("POST -> login-user", { color: "BLUE" })

  try {
    res.cookie('refreshToken', "", {
      httpOnly: true,
      secure: true,
      maxAge: 0,
      path: '/refresh-aut-token',
      sameSite: 'None',
      domain: getCookieDomainFromUrl(process.env.CURRENT_DOMAIN)
    });
    return sendCustomRes(res, "250")
  }
  catch (error) {
    return sendCustomRes(res, "-1")
  }
});
```

Figura 47: Funzione logout()

4.8 API per la gestione dei gruppi

- **getGroupById:** Recupera i dettagli di un gruppo specifico

```

// Gets group by id
this.app.get('/group/:id', async (req, res) => {
  dprint(`GET -> group/${req.params.id}`, { color: "GREEN" })

  try {
    const groupId = req.params.id;
    const group = await DB.getGroupById(groupId);
    if (group)
      sendCustomRes(res, "2203", group)
    else
      sendCustomRes(res, "2404", group)

  } catch (error) {
    sendCustomRes(res, "-1", error)
  }
});

```

Figura 48: Funzione getGroupById()

- **updateGroupProperty:** Aggiorna una proprietà specifica di un gruppo

```

this.app.patch('/group/:id', checkGroupPropertyModifiable, async (req, res) => {
  dprint(`PATCH -> group/${req.params.id}`, { color: "YELLOW" })
  try {
    const groupId = req.params.id;
    const reqBody = req.body;
    if (await DB.patchGroup(groupId, reqBody) == true)
      sendCustomRes(res, 2204)
    else
      sendCustomRes(res, -1, "DB error")
  } catch (error) {
    sendCustomRes(res, -1, error)
  }
});

```

Figura 49: Funzione updateGroupProperty()

- **deleteGroup:** Elimina un gruppo specifico

```

// deletes the specified group
this.app.delete('/group/:id', checkGruopDeletionPriv, async (req, res) => {
  dprint(`DELETE -> group/${req.params.id}`, { color: "BLUE" })

  try {
    const rv = await DB.deleteGroup(req.params.id)
    if (rv)
      sendCustomRes(res, "2201")
    else
      sendCustomRes(res, "2401")
  }
  catch (error) {
    return sendCustomRes(res, "-1", error)
  }
});


```

Figura 50: Funzione deleteGroup()

- **createGroup:** Crea un nuovo gruppo

```

//Uso put poiché metodo idempotente
this.app.put('/create-group', checkAut, async (req, res) => {
  dprint("PUT -> create-group", { color: "BLUE" })
  try {
    const { name, description } = req.body;
    const groupId = await DB.createGroup(name, req.uid, description, { [req.uid]: { createdAt: new Date(), joinedAt: new Date() } });
    if (groupId == null)
      return sendCustomRes(res, "2403")
    else
      return sendCustomRes(res, "2200", groupId.insertedId)
  } catch (error) {
    return sendCustomRes(res, "-1", error)
  }
});


```

Figura 51: Funzione createGroup()

- **deleteAllGroups:** Elimina tutti i gruppi (solo per gli amministratori)

```

// Deletes all groups. Requires admin token
this.app.delete('/groups', isAdmin, async (req, res) => {
  dprint("DELETE -> groups", { color: "BLUE" })
  try {

    const removedCount = await DB.resetGroupsCollection()
    return sendCustomRes(res, "2250", removedCount)

  } catch (error) {
    return sendCustomRes(res, "-1")
  }
});

```

Figura 52: Funzione deleteAllGroups()

4.9 API per la gestione delle relazioni tra utenti e gruppi

- **unlinkUserFromGroup:** Rimuove il collegamento tra un utente e un gruppo

```

// Removes user from group
this.app.patch('/unlink', checkUnlinkPriv, async (req, res) => {
  dprint(`PATCH -> unlink`, { color: "YELLOW" })
  dprint(req.body, { debuglvl: 2 })

  try {
    const { userId, groupId } = req.body;
    const rv = await DB.removeLink(userId, groupId);
    if (rv == true)
      return sendCustomRes(res, "3201")
    else
      return sendCustomRes(res, "-1")
  } catch (error) {
    return sendCustomRes(res, "-1", error)
  }
});

```

Figura 53: Funzione unlinkUserFromGroup()

- **linkUserToGroup:** Collega un utente a un gruppo

```

// Adds users to group
this.app.patch('/link', checkLinkPriv, async (req, res) => {
  dprint(`PATCH -> link`, { color: "YELLOW" })
  dprint(req.body, { debuglvl: 2 })

  try {
    const { userId, groupId } = req.body;
    const rv = await DB.addLink(userId, groupId);
    if (rv == true)
      return sendCustomRes(res, "3200")
    else
      return sendCustomRes(res, "-1")
  } catch (error) {
    return sendCustomRes(res, "-1", error)
  }
});

```

Figura 54: Funzione linkUserToGroup()

4.10 API per la gestione del calendario

- **deleteGroupCalendarEvent:** Rimuove un evento dal calendario di un gruppo specifico.

```

this.app.delete('/group/cal/:id/add-event', checkGroupRemoveEventPriv, async (req, res) => {
  dprint(`PATCH -> group/cal${req.params.id}/add-event`, { color: "YELLOW" })
  try {
    const userId = req.params.id;
    const reqBody = req.body;

    if (await DB.removeEventFromUserCal(userId, reqBody.calendar ?? "default", reqBody.index) == true)
      sendCustomRes(res, 4200)
    else
      sendCustomRes(res, -1, "DB error")
  } catch (error) {
    sendCustomRes(res, -1, error)
  }
});

```

Figura 55: Funzione deleteGroupCalendarEvent()

- **deleteUserCalendarEvent:** Rimuove un evento dal calendario di un utente specifico.

```

this.app.delete('/user/cal/:id/add-event', checkUserRemoveEventPriv, async (req, res) => {
  dprint(`PATCH -> user/cal${req.params.id}/add-event`, { color: "YELLOW" })
  try {
    const userId = req.params.id;
    const reqBody = req.body;

    if (await DB.removeEventFromGroupCal(userId, reqBody.calendar ?? "default", reqBody.index) == true)
      sendCustomRes(res, 4200)
    else
      sendCustomRes(res, -1, "DB error")
  } catch (error) {
    sendCustomRes(res, -1, error)
  }
});

```

Figura 56: Funzione deleteUserCalendarEvent()

- **addGroupCalendarEvent:** Aggiunge un evento al calendario di un gruppo specifico.

```

// 4200 -1
this.app.patch('/group/cal/:id/add-event', checkGroupAddEventPriv, async (req, res) => {
  dprint(`PATCH -> group/cal${req.params.id}/add-event`, { color: "YELLOW" })
  try {
    const userId = req.params.id;
    const reqBody = req.body;

    if (await DB.addEventToUserCal(userId, reqBody.calendar ?? "default", { startDate: reqBody.startDate, endDate: reqBody.endDate }) == true)
      sendCustomRes(res, 4200)
    else
      sendCustomRes(res, -1, "DB error")
  } catch (error) {
    sendCustomRes(res, -1, error)
  }
});

```

Figura 57: Funzione addGroupCalendarEvent()

- **addUserCalendarEvent:** Aggiunge un evento al calendario di un utente specifico.

```

this.app.patch('/user/cal/:id/add-event', checkUserAddEventPriv, async (req, res) => {
  dprint(`PATCH -> user/cal${req.params.id}/add-event`, { color: "YELLOW" })
  try {
    const userId = req.params.id;
    const reqBody = req.body;

    if (await DB.addEventToUserCal(userId, reqBody.calendar ?? "default", { startDate: reqBody.startDate, endDate: reqBody.endDate }) == true)
      sendCustomRes(res, 4200)
    else
      sendCustomRes(res, -1, "DB error")
  } catch (error) {
    sendCustomRes(res, -1, error)
  }
});

```

Figura 58: Funzione addUserCalendarEvent()

4.11 API per la gestione dello stato del sistema

- **getSystemState:** Restituisce lo stato corrente del sistema, ovvero la lista di tutti gli utenti e di tutti i gruppi.

```
// returns a list of all groups and users
this.app.get('/state', isAdmin, async (req, res) => {
  dprint("GET -> /state", { color: "BLUE" })

  try {
    const groups = await DB.groupsCollection.find().toArray();
    const users = await DB.usersCollection.find().toArray();
    return sendCustomRes(res, "253", { groups, users })
  } catch (error) {
    return sendCustomRes(res, "-1", error)
  }
});
```

Figura 59: Funzione getSystemState()

- **testServerStatus:** API di test utilizzata per verificare il corretto funzionamento del server.

```
// Test whether server is working correctly
this.app.get('/test', async (_, res) => {
  dprint("GET -> test", { color: "BLUE" })

  try {
    return sendCustomRes(res, "2")
  } catch (error) {
    return sendCustomRes(res, "-1")
  }
});
```

Figura 60: Funzione testServerStatus()

- **restrictAccessAPI:** API di test per verificare l'autenticazione dell'utente.

```
// Generic api to test user authentication
this.app.get('/restricted-api', checkAut, async (req, res) => {
  dprint("-> restricted-api", { color: "BLUE" })

  try {
    return sendCustomRes(res, "250")
  }
  catch (error) {
    return sendCustomRes(res, "-1")
  }
})
```

Figura 61: Funzione restrictAccessAPI()

- **tokenRefresh:** Restituisce un token JWT temporaneo per accedere alle API protette.

```
// Restituisce un token jwt temporaneo che serve a garantire l'accesso alle api protette. Richiede un token refresh nella richiesta
this.app.get('/refresh-aut-token', async (req, res) => {
  dprint("GET -> refresh-aut-token", { color: "BLUE" })
  dprint(req.cookies, { color: "BLUE", debuglvl: 2 })

  try {
    const refreshToken = req.cookies.refreshToken
    if (!refreshToken)
      return sendCustomRes(res, "475")

    jwt.verify(refreshToken, process.env.REFRESH_PRIVATE_KEY, (err, data) => {
      if (err)
        return sendCustomRes(res, "477")
      else {
        if (!(data.uid && data.role))
          return sendCustomRes(res, "476")
        const accessToken = jwt.sign({ uid: data.uid, role: data.role }, process.env.AUTH_PRIVATE_KEY, { expiresIn: process.env.AUTH_TOKEN_TTL ?? "60s" })
        return sendCustomRes(res, "252", accessToken)
      }
    })
    catch (error) {
      return sendCustomRes(res, "-1")
    }
  });
})
```

Figura 62: Funzione tokenRefresh()

- **currentId:** Restituisce l'ID dell'utente attualmente autenticato.

```
this.app.get('/current-uid', checkAut, async (req, res) => {
  try {
    | sendCustomRes(res, 1205, req.uid)
  }
  catch (error) {
    | sendCustomRes(res, -1, error)
  }
})  
}
```

Figura 63: Funzione currentId()

5 API documentation

Le API locali fornite dall'applicazione Mergify sono state documentate tramite Swagger UI Express, un modulo NodeJS. Questo metodo consente di rendere la documentazione delle API facilmente accessibile a chiunque abbia accesso al codice sorgente. Per la creazione dell'endpoint dedicato alla visualizzazione delle API, è stato impiegato Swagger UI, che genera una pagina web basata sulle specifiche OpenAPI.

Di seguito è disponibile la pagina web che presenta la documentazione delle API necessarie per la gestione dei dati all'interno della nostra applicazione. L'endpoint per accedere a questa documentazione è: <https://mergify-backhand.onrender.com/docs>

 Swagger
Supported by SMARTBEAR

Mergify backend apis documentation 1.0.0 OAS 3.0

Below is the documentation for the Mergify backend APIs. Source code available [here](#)

Authenticated apis notes:

Authentication is managed via JWT tokens, exchanged through the Authorization header (`autToken`) and httponly cookies (`refreshToken`). Due to the different domains between the backend and frontend, it may be necessary to disable cross-site tracking prevention in browsers such as *Safari*.

APIs labeled with a  icon require a JWT `autToken` in the Authorization header of the request. This token can be obtained from the user/admin login API, which sets the `refreshToken` as a cookie. This mechanism provides two levels of privilege: user and admin. To test restricted access APIs, follow these steps:

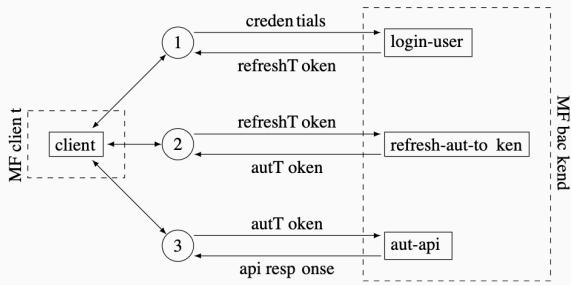
Testing authenticated apis

1. login API -> Obtain the `refreshToken` as a cookie.

```
{
  "intId": 252,
  "msg": "Refresh token successfully generated",
  "value": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlaWQiOiI2NmQzMTh2N2E2MGYzY2ZkMGUzZDgiLCJyb2xlIjoiYWRtaW4iLCJpYXQiOjE3MjUzMdkyOTcsImV4cCI6MTcyNTMyMDA5N30.uzeBwZiuSwLLiWpKRL2zehpP4li0yQSSSB28qpwkA"
}
```

2. refresh-aut-token API -> Obtain the `autToken` in the response body.
3. Paste the obtained `autToken` into the authorization field in green at the top right of the page.

Once completed, Swagger will automatically attach the token to every request that requires it, and will signal it with the  icon



```

sequenceDiagram
    participant Client as MF client
    participant LoginUser as login-user
    participant RefreshAutToKen as refresh-aut-to-ken
    participant AutApi as aut-api

    Client->>LoginUser: credentials
    activate LoginUser
    LoginUser->>Client: refreshToken
    deactivate LoginUser
    Client->>RefreshAutToKen: refreshToken
    activate RefreshAutToKen
    RefreshAutToKen->>Client: autToken
    deactivate RefreshAutToKen
    Client->>AutApi: autToken
    activate AutApi
    AutApi->>Client: api response
    deactivate AutApi

```

Just for testing purposes, here is a valid authentication tokens with no expiry date, to test apis more quickly:
Admin: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlaWQiOiJhZG1pbkiIsInJvbGUiOiJhZG1pbkiIsImhdCI6MTcyNTU2Njc5Nn0.79UDbt_bb6fibwXMhs80GdVT5bKPXf0p4gZGpn24ji0`

Figura 64: Swagger UI Documentazione 1

Servers <https://mergify-backhand.onrender.com> [Authorize](#)

Users All APIs related to users

- PUT** /create-user Create a new user
- GET** /user/{id} Retrieve a user by ID
- DELETE** /user/{id} Delete a user by ID
- PATCH** /user/{id} Change a property of a user
- GET** /current-uid Retrieve user's ID

Groups All APIs related to groups of users

- PUT** /create-group Create a new group
- GET** /group/{id} Retrieve a group by ID
- DELETE** /group/{id} Delete a group by ID
- PATCH** /group/{id} Change a property of a group

Calendar All APIs to manage user/group calendars

- PATCH** /user/cal/{id}/add-event Add event to the user personal cal
- DELETE** /user/cal/{id}/add-event Remove event from the user cal
- PATCH** /group/cal/{id}/add-event Add event to the group shared cal
- DELETE** /group/cal/{id}/add-event Remove event from the group shared cal

Figura 65: Swagger UI Documentazione 2

Links All APIs to add/remove users from groups

- PATCH** /link Link a user to a group
- PATCH** /unlink Unlink a user from a group

Authorization All APIs related to user/admin authorization

- GET** /refresh-aut-token Refresh authentication token
- POST** /login-user Login a user
- POST** /login-admin Login an admin
- POST** /logout Logout

Admin All APIs restricted to admin use

- DELETE** /users Delete all users
- DELETE** /groups Delete all groups
- GET** /state Get the state of the system

System Misc convenience apis

- GET** /test Test endpoint

Figura 66: Swagger UI Documentazione 3

The screenshot shows the Swagger UI interface for a 'System' API. At the top, it displays a 'GET /test Test endpoint'. Below this, under 'Parameters', there are no parameters listed. Under 'Responses', there are two entries:

- 200 OK**: Description: 'The request was successful.'
Media type: application/json
Example Value: { "intId": 2, "intMsg": "Server is working fine" }
Links: No links
- 500 Internal Server Error**: Description: 'The server has encountered a situation it doesn't know how to handle.'
Media type: application/json
Example Value: { "intId": -1, "intMsg": "Unknown server error" }
Links: No links

Figura 67: Swagger UI Documentazione 4

6 FrontEnd implementation

Il front end è composto dalle seguenti pagine:

- Login
- Signup
- My cal
- My groups
- Info utente
- Group cal

Dalle pagine di login e registrazione è possibile passare dall'una all'altra. Una volta effettuato l'accesso, è disponibile una barra di navigazione superiore che consente di accedere rapidamente alle pagine "My Cal" e "My Groups". Inoltre, da questa barra è possibile eseguire il logout e ritornare alla pagina di login.

6.1 Login

La pagina è suddivisa in due sezioni: a sinistra è possibile interagire con il sito, mentre a destra è fornita una breve descrizione.

Nella pagina di login, sono presenti due campi di testo per inserire l'email e la password. L'autenticazione avviene tramite il pulsante di login. Inoltre, sono disponibili una casella di controllo per il salvataggio della password e un link per il reset della stessa. È anche presente un collegamento per accedere alla pagina di registrazione.

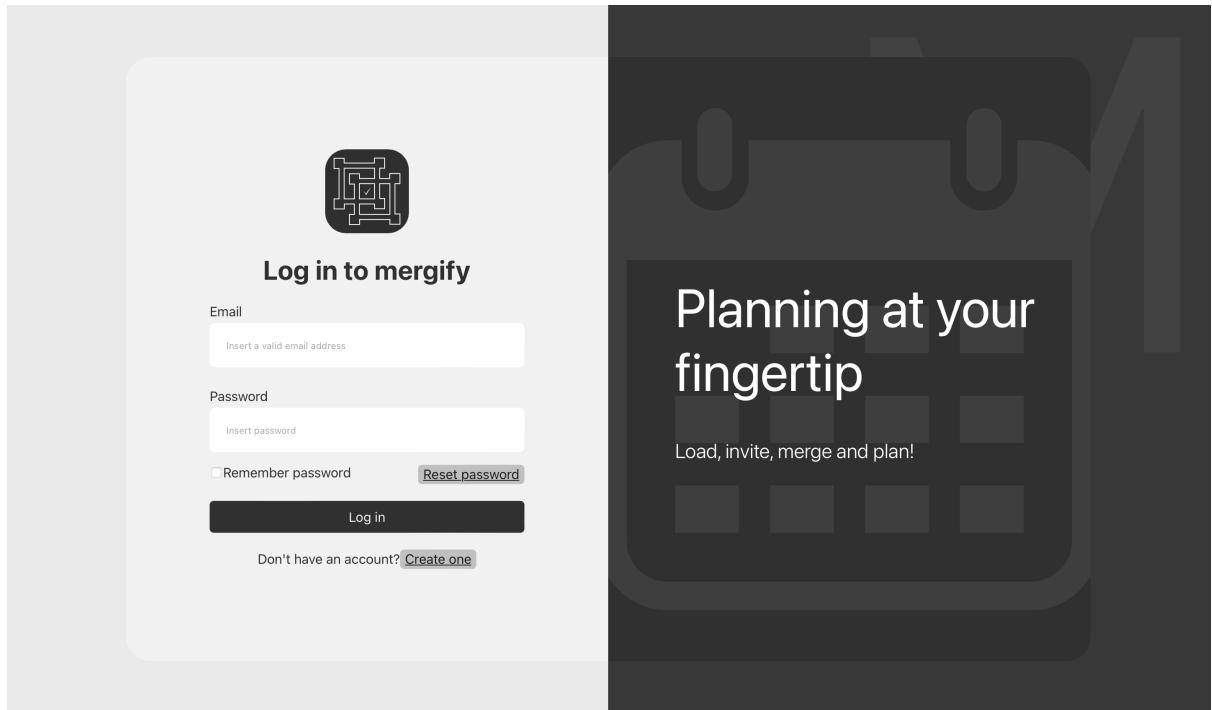


Figura 68: Pagina di login

6.2 Signup

Anche la pagina di registrazione è suddivisa in due sezioni. Qui, oltre ai campi per l'email e la password, è richiesta una doppia conferma della password. In basso sono presenti i pulsanti per completare la registrazione o per tornare alla pagina di login.

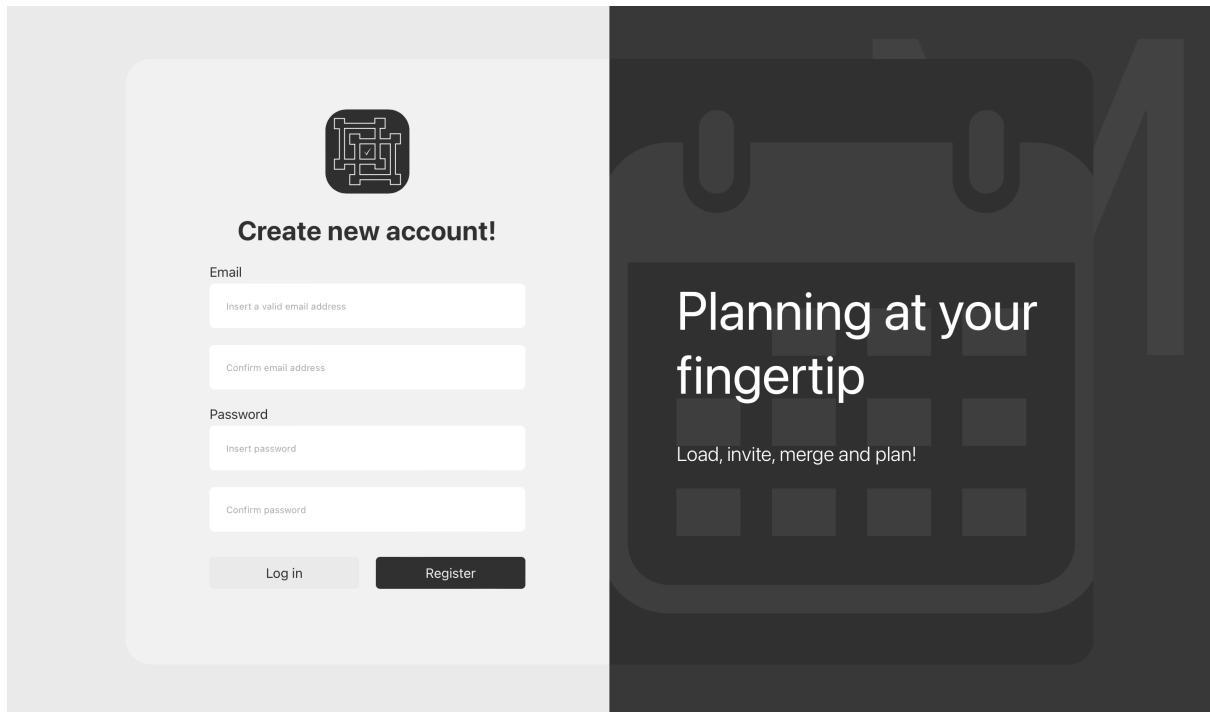


Figura 69: Pagina di registrazione

6.3 My cal

Nella pagina "My Cal", l'utente può visualizzare il proprio calendario.

The screenshot shows a web-based calendar interface. At the top, there are navigation tabs: "My cal" (selected), "My groups", "Mio calendario", "Mario Rossi", and "Log out". Below the tabs is a weekly grid from Monday to Sunday, showing time slots from 00:00 to 11:00. To the right of the grid is a monthly calendar for September 2024. The month header says "Settembre 2024". The days of the week are labeled M, T, W, T, F, S, S. The dates are arranged as follows: Row 1: 26, 27, 28, 29, 30, 31, 1; Row 2: 2, 3, 4, 5, 6, 7, 8; Row 3: 9, 10, 11, 12, 13, 14, 15; Row 4: 16, 17, 18, 19, 20, 21, 22; Row 5: 23, 24, 25, 26, 27, 28, 29; Row 6: 30, 1, 2, 3, 4, 5, 6. A small red circle highlights the date "5". Below the calendar are buttons for "Today", "Upload calendar", and "Delete calendar".

Figura 70: Pagina "My Cal"

Nella parte destra dello schermo, l'utente ha la possibilità di cambiare visualizzazione del calendario, selezionando tra mese e settimana. È inoltre possibile tornare alla visualiz-

zazione della settimana corrente utilizzando il pulsante "Today". In questa sezione sono presenti anche opzioni per caricare nuovi calendari o eliminarli.

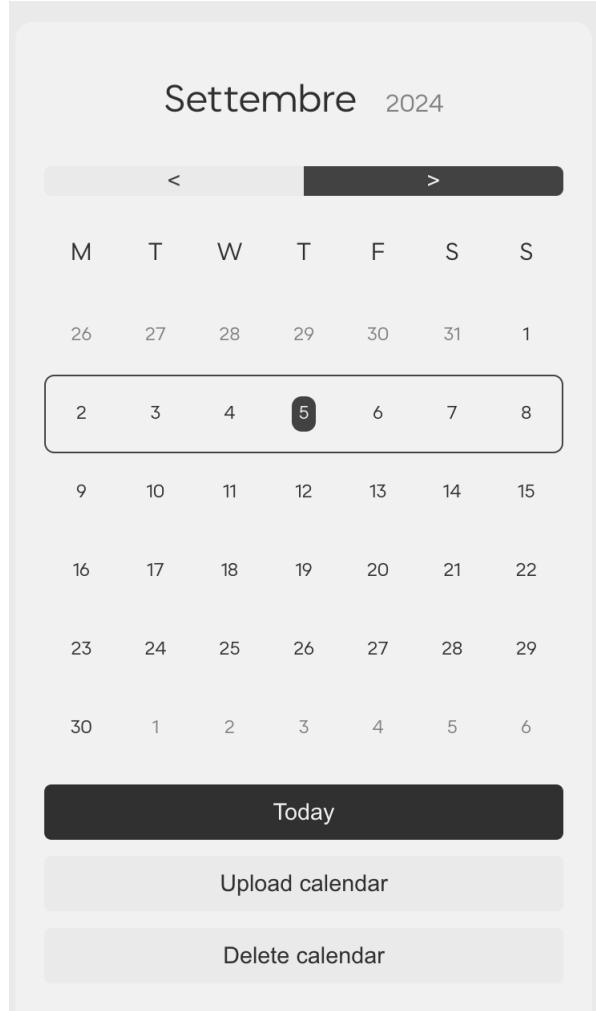


Figura 71: Opzioni di visualizzazione e gestione del calendario

Nella parte sinistra della pagina, l'utente può visualizzare e aggiungere eventi. È presente una barra di navigazione che consente di spostarsi tra le diverse pagine, accedere alle proprie informazioni personali o eseguire il logout.



Figura 72: Navigazione e gestione eventi

6.4 My groups

La sezione "My Groups" presenta un elenco di gruppi, ciascuno accompagnato da una breve descrizione. Selezionando un gruppo, si apre una schermata laterale che fornisce ulteriori dettagli.

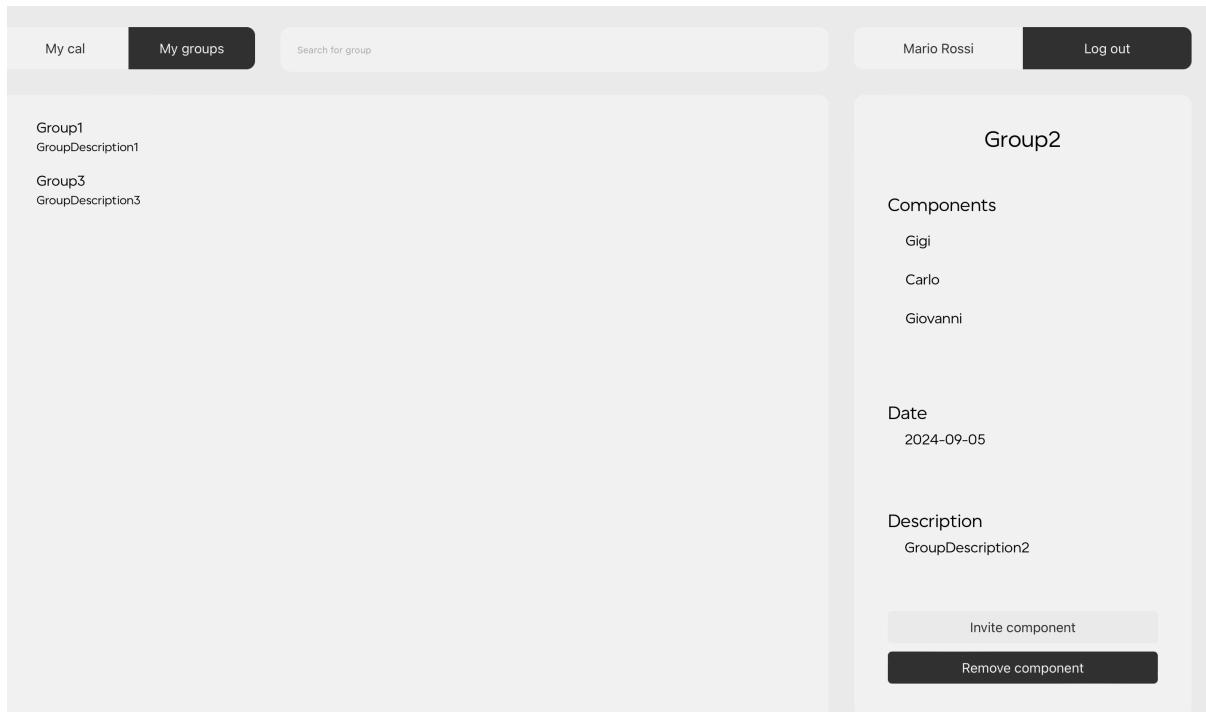


Figura 73: Elenco gruppi

Nella schermata laterale sono disponibili pulsanti per invitare o rimuovere membri del gruppo. Inoltre, vengono visualizzate informazioni sui componenti, le date e una descrizione dettagliata del gruppo.

La barra di navigazione è identica a quella della pagina "My Cal", ad eccezione della sezione "Mio Calendario", che è sostituita da una funzione di ricerca dei gruppi.

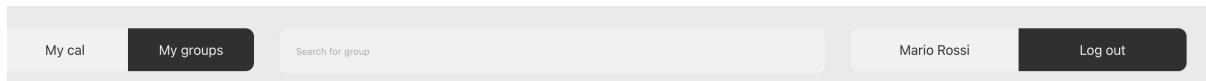


Figura 74: Dettagli gruppo e navigazione

6.5 Info utente

La schermata "Info Utente" visualizza l'icona, il nome e l'email dell'utente.

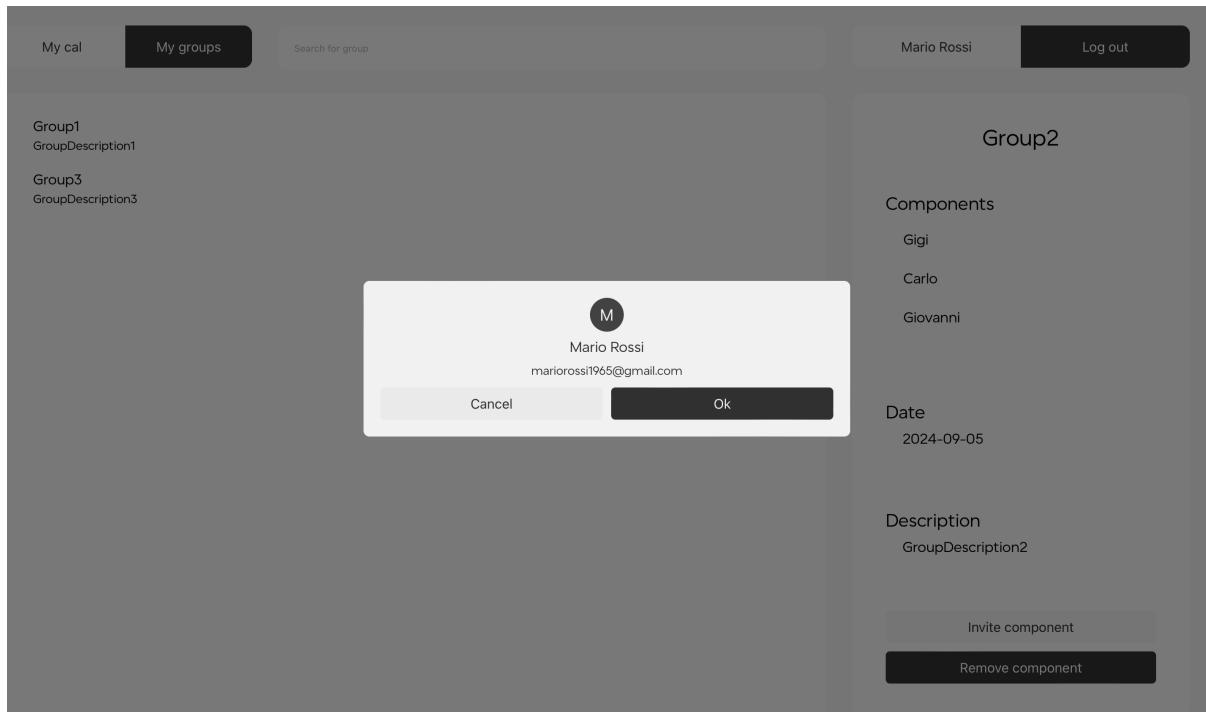


Figura 75: Informazioni utente

7 Testing

Tutte le API sono state testate manualmente utilizzando Swagger. In alternativa, è possibile eseguire i test tramite Postman.

Per scopi di testing, l'accesso come amministratore è stato configurato con le seguenti credenziali predefinite: nome utente "Username1" e password "password1".

N	Descrizione	Test Data	Precondizioni	Dipendenze	Risultato atteso
1	Registrazione.	Email Password	Email mai inserita nel sistema.	Nessuna.	L'account viene creato con successo.
2	Registrazione con email già utilizzata.	Email già presente nel database.	Email già presente nel database.	Dopo il caso 1.	Viene restituito un errore.
3	Registrazione co password non conforme ai criteri minimi	Password non conforme alle politiche di sicurezza.	Nessuna.	Nessuna.	Viene restituito un errore.
4	Login.	Email Password	Email e Password inserite nel database.	Dopo il caso 1.	L'accesso viene garantito e viene creato un cookie per la sessione.
5	Login con account non presente.	Email non presente nel database.	Nessuna.	Nessuna.	Viene restituito un errore.
6	Login con password errata.	Password non corretta associata a Email	Email presente nel database.	Dopo il caso 1.	Viene restituito un errore.
7	Leggere i dati di un utente.	Id	Id presente nel database.	Dopo il caso 4.	L'utente legge i dati corretti dell'utente.
8	Leggere i dati di un utente inesistente.	Id non esistente.	Nessuna.	Nessuna.	Viene restituito un errore.
9	Modificare i dati dell'utente.	id	id presente nel database.	Nessuna.	L'utente modifica i dati con successo.
10	Creare gruppo.	name description	Nessuna.	Nessuna.	L'utente crea un gruppo con successo.
11	Leggere gruppo.	id	id presente nel database	Nessuna.	L'utente legge il gruppo.
12	Leggere gruppo inesistente.	id non esistente.	Nessuna.	Nessuna.	Viene restituito un errore.
13	Modificare i dati del gruppo.	id	id presente nel database	Nessuna.	L'utente modifica i dati del gruppo con successo.
14	Eliminare un gruppo essendo leader.	id	leaderid del gruppo uguale a id	Nessuna.	L'utente elimina il gruppo.
15	Eliminare un gruppo non essendo leader.	id	leaderid del gruppo diverso a id	Nessuna.	Viene restituito un errore.

Tabella 1: Tabella dei casi di test

N	Descrizione	Test Data	Precondizioni	Dipendenze	Risultato atteso
16	Utente elimina se stesso	id	id uguale a id utente	Nessuna.	L'utente elimina il proprio account
17	Utente elimina un'altro utente	id	id diverso da id utente	Nessuna.	Viene restituito un errore.
18	Utente elimina un utente inesistente	id	id non presente nel database.	Nessuna.	Viene restituito un errore.
19	Logout	Nessuna.	Nessuna.	Dopo il caso 4.	L'utente esegue il logout.
20	Login come admin.	Email Password	Email e Password inserite nel database.	Nessuna.	L'accesso viene garantito e viene creato un cookie per la sessione.
21	Admin elimina un utente	Nessuna.	Nessuna.	Dopo il caso 20	L'admin elimina l'utente.
22	Admin elimina tutti gli utenti	Nessuna.	Nessuna.	Dopo il caso 20	L'admin elimina tutti gli utenti.
23	Admin elimina tutti i gruppi	Nessuna.	Nessuna.	Dopo il caso 20	L'admin elimina tutti i gruppi.
24	Leggere i dati di tutti gli utenti e gruppi	Nessuna.	Nessuna.	Dopo il caso 20	L'admin legge i dati di tutti gli utenti e gruppi

8 Github repository e informazioni sul deployment

Il progetto Mergify è disponibile al seguente link: <https://github.com/orgs/G-51/>

È suddiviso in quattro repository:

- “frontend”: contiene il codice relativo al Front-End
- “backend”: contiene il codice relativo al Back-end
- “admin-dashboard”: contiene il codice per accedere alla dashboard e per utilizzare le api per gli amministratori
- “Deliverable”: contiene i PDF di tutti i deliverables

Il deployment è gestito attraverso la piattaforma Vercel ed è disponibile al seguente link:
<https://frontend-rho-khaki.vercel.app/login>

Per accedere alla dashboard per gli amministratori, usufruire del seguente link:
<https://admin-dashboard-virid-beta.vercel.app/>