Trillium

Documento di sviluppo - D4

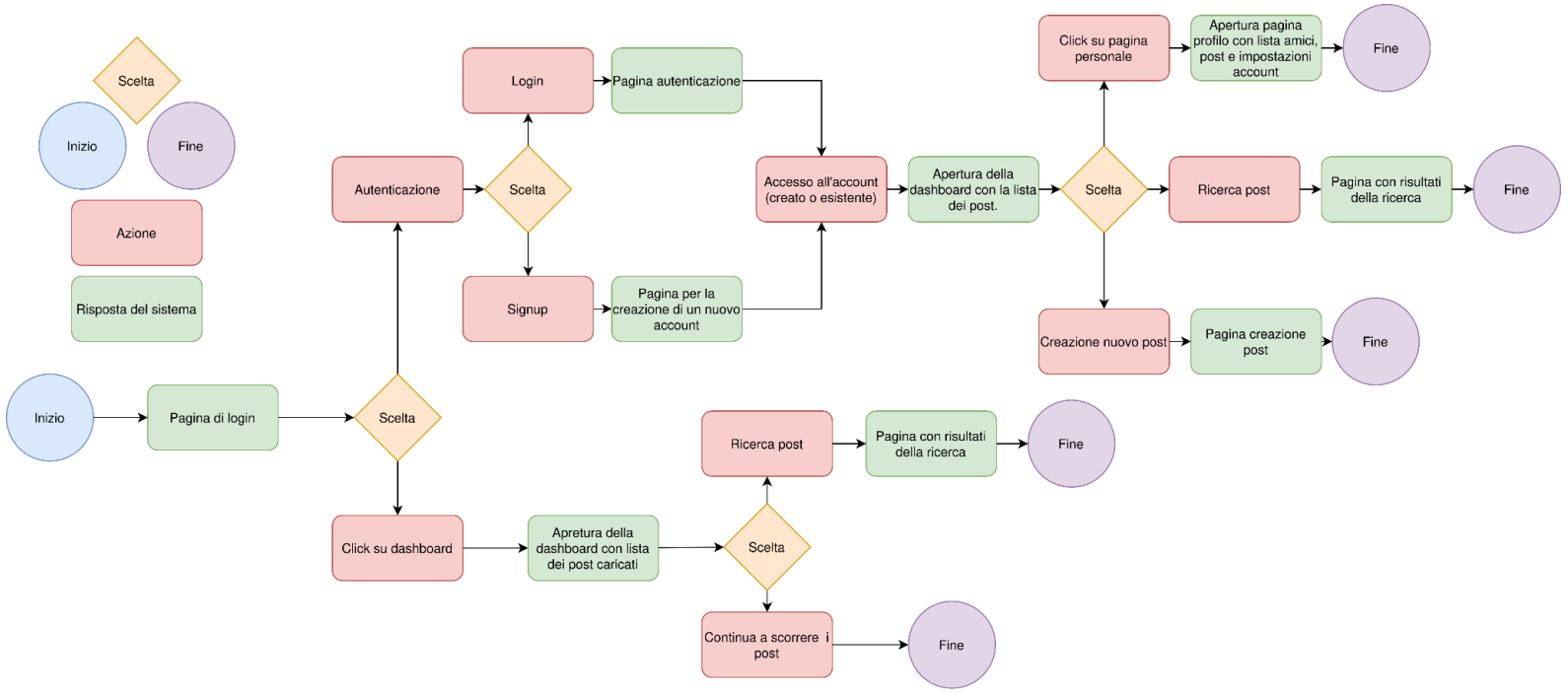
Indice

Scopo del documento

Il presente documento riporta le principali informazioni necessarie per lo sviluppo dell'applicazione web "Trillium".

User flows

In questa sezione vengono riportati gli "**User Flows**" di un utente che apre il sito web. L'utente, una volta che arriva nella pagina, puo' decidere se autenticarsi (quindi creare un nuovo account oppure accedere con uno gia' esistente) oppure aprire la dashboard, che conterrà i post pubblici caricati dagli altri utenti. Sulla sinistra e' presente una piccola legenda che fornisce una breve descrizione dei simboli utilizzati.

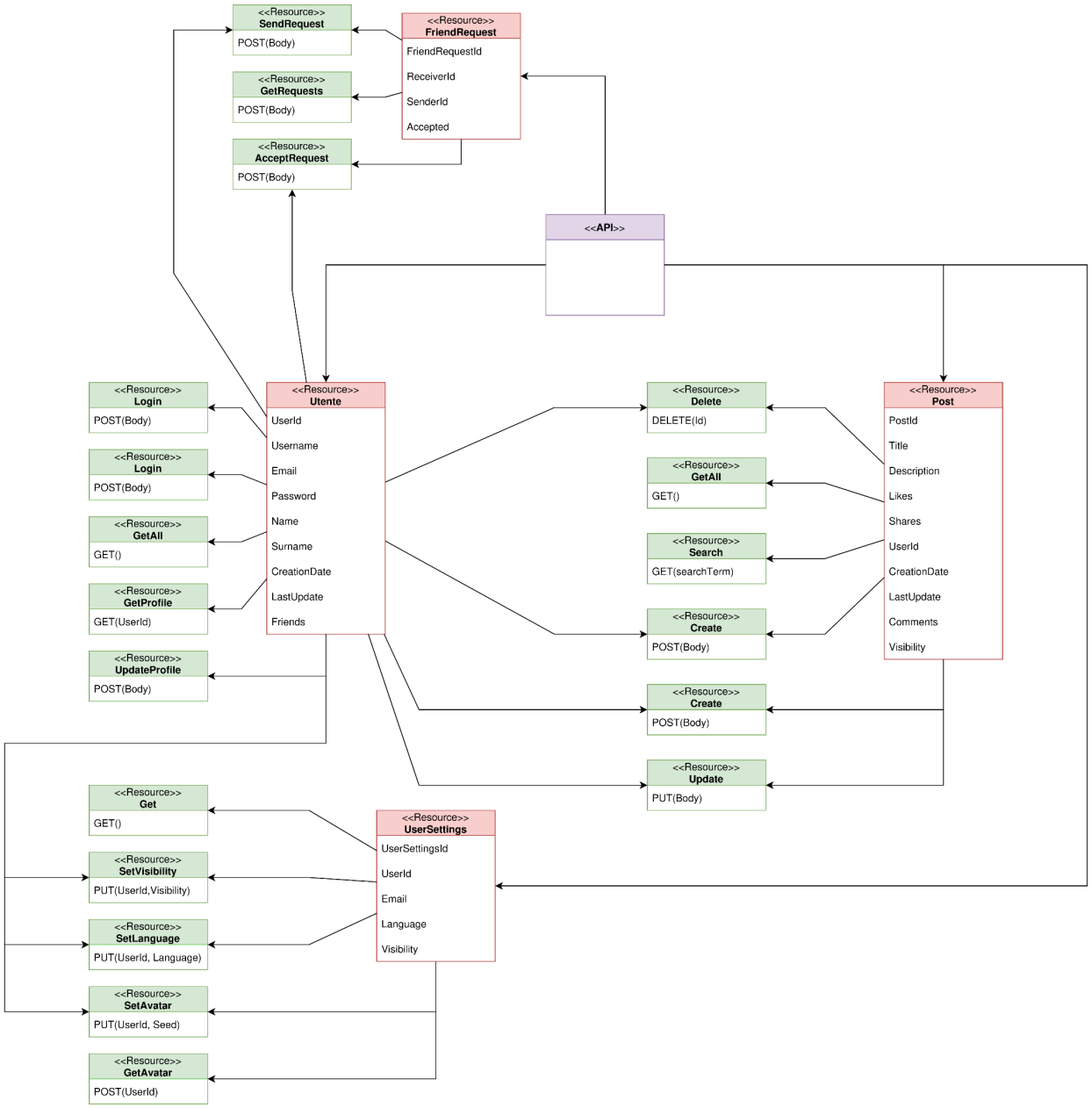


Come si può notare dal flusso, gran parte delle azioni sono disponibili solamente all'utente autenticato. Un utente "visitatore" potra' eseguire solamente qualche azione di base, come visualizzare i post e ricercarli.

APIs Specification from Class Diagram

Questa sezione contiene il modello delle risorse estratte dal class diagram, ognuna con le API dedicate e le relative dipendenze.

Modello delle risorse



APIs Implementation

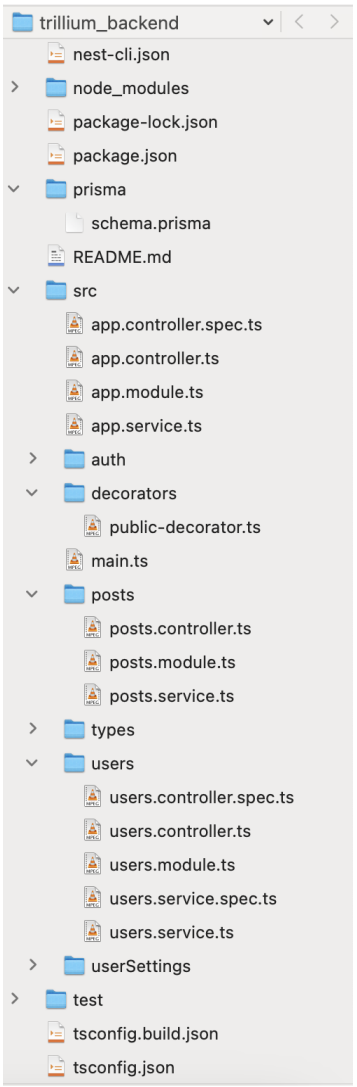
Questa sezione rappresenta l'implementazione effettiva del progetto, quindi la sua struttura, le dipendenze, i modelli di dati e la descrizione di alcune delle molte API implementate.

Project structure

Il progetto è diviso in 2 sotto-progetti:

1. **Backend**: Tutto ciò che riguarda le API che la parte frontend utilizza e la comunicazione con il DB (e altri servizi che utilizziamo.)
2. **Frontend**: Quello che riguarda l'interazione con l'utente, quindi la parte di interfaccia grafica ed esposizione del contenuto.

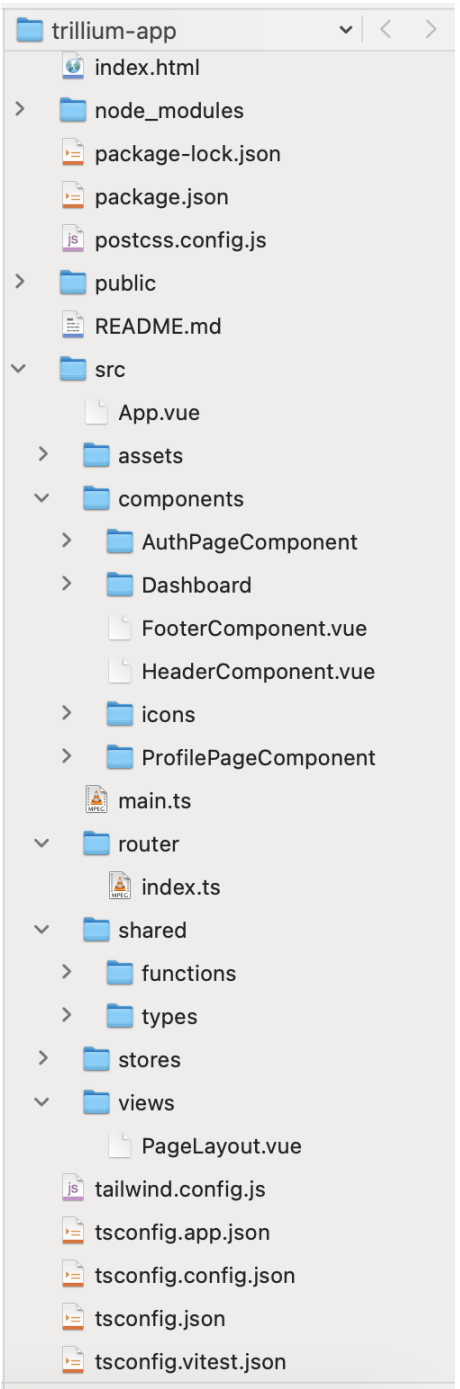
La struttura del progetto di **back-end** e' presentata nella seguente figura:



Le cartelle principali sono:

1. **prisma**: ORM che utilizziamo per gestire le query e i modelli del db. Il file schema.prisma contiene la definizione di tutti i modelli e tipi del DB.
2. **src**: Contiene tutti i componenti, quindi i controller con gli endpoint esposti, la parte di logica chiamata dai controller e i test dei relativi componenti.
3. **types**: Cartella condivisa che contiene vari tipi che vengono utilizzati da più componenti.

La struttura della parte di **front-end** e' invece la seguente:



Strutturalmente e' molto simile alla parte di backend. Anche qui infatti abbiamo i vari oggetti suddivisi in componenti e views.

I componenti sono tutti quegli oggetti che compongono le pagine create, mentre le viste sono i template delle pagine.

Come template ne abbiamo utilizzato uno: header sopra e footer sotto, con il contenuto al centro.

Tra i componenti principali abbiamo:

1. **Dashboard**: Contiene la lista dei post creati dagli altri utenti e una sezione per creare un nuovo post.
2. **Auth**:Contiene i componenti di login e signup, per accedere / registrare l'utente.
3. **ProfilePage**: Contiene tutte le informazioni (post, amici, nome, ecc) dell'utente loggato.

Project dependencies

Sono stati impiegati diversi moduli di note per la realizzazione del progetto.
Nella parte di backend abbiamo:

- **NestJs**: Un framework per lo sviluppo del backend.
- **Jest**: Un framework per la definizione e l'esecuzione dei test.
- **EsLint**: Uno strumento per l'analisi statica del codice, utile per individuare errori nel codice.
- **Prettier**: Uno strumento per formattare automaticamente il codice in modo uniforme.
- **Prisma**: Un ORM (Object-Relational Mapping) che funge da strato di mappatura dati. È utilizzato come gestore del database.
- **Typescript**: Una variante di JavaScript che supporta la tipizzazione, migliorando la gestione del codice soprattutto in contesti di lavoro di gruppo.

Nella parte di frontend abbiamo:

- **VueJs**: Libreria di Javascript per lo sviluppo di frontend.
- **Tailwindcss**: Framework CSS che fornisce delle astrazioni alle classiche regole CSS, permettendo uno sviluppo piu' rapido della componente estetica.
- **Prettier**: Uno strumento per formattare automaticamente il codice in modo uniforme.
- **Vite**: Server per lo sviluppo locale compatibile con Typescript e Javascript.
- **EsLint**: Uno strumento per l'analisi statica del codice, utile per individuare errori nel codice.

Database

Per la gestione del dati abbiamo definito le seguenti strutture all'interno del database:

trillium-db

LOGICAL DATA SIZE: 30.74KB STORAGE SIZE: 132KB INDEX SIZE: 232KB TOTAL COLLECTIONS: 7 CREATE COLLECTION

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
FriendRequest	7	553B	79B	20KB	1	20KB	20KB
Friends	0	0B	0B	4KB	1	4KB	4KB
Post	41	9.18KB	230B	20KB	1	20KB	20KB
PostMedia	0	0B	0B	4KB	1	4KB	4KB
User	80	16.05KB	206B	44KB	3	108KB	36KB
UserSettings	62	4.96KB	82B	36KB	2	72KB	36KB

Espandendo, abbiamo:

- **FriendRequest**: Tabella che contiene tutte le richieste di amicizia pendenti oppure accettate / rifiutate.
- **Friends**: Tabella che contiene le coppie di utenti che compongono un'amicizia
- **Post**: Contiene i post e tutti i loro relativi dati (titolo, descrizione, ecc).
- **PostMedia**: Tabella relativa ai post caricati all'interno del post.
- **User**: Dati dell'utente (username, email, password, ecc).
- **UserSettings**: Impostazioni specifiche dell'utente, come la lingua o la visibilità del profilo.

Per rappresentare gli utenti e i post abbiamo utilizzato la seguente struttura:

```
1  _id: ObjectId('637f2f8e9342d8d6ce9fe95d')
2  email: "cristianTest@cristian.it"
3  password: "2b085d7fbb8717b253fe4283c564ab2592f4f4ac962dee11e296129b504d35ff"
4  username: "Test"
5  name: "CristianTest"
6  surname: "Cristian"
7  creationDate: 2022-11-24T08:47:10.902+00:00
8  lastUpdate: 2023-05-11T07:31:49.054+00:00
9  ▸ friends: Array
```

```
_id: ObjectId('64eda0a15598d2f6abcae78d')
title: "Ciaoo!"
description: "Oggi c'e' il sole"
likes: 0
shares: 0
creationDate: 2023-08-29T07:39:13.344+00:00
lastUpdate: 2023-08-29T07:39:13.344+00:00
deletedOn: null
userId: ObjectId('637f2e369342d8d6ce9fe95c')
▸ postComments: Array
visibility: "public"
```

Project APIs

Di seguito alcune delle API sviluppate con il relativo codice. Vista la grande quantita' ne riporteremo solamente alcune.

Creazione di un utente

L'API **SignUp** permette di creare un nuovo oggetto **User** nel database, per permettere successivamente di effettuare il login dello stesso.

```
async signup(user: signupUserType) {
  try {
    const { password, ...newUserData } = await this.usersService.createUser(
      user,
    );
    return newUserData;
  } catch (e: any) {     ■ Unexpected any. Specify a different type.
    console.log(e.meta.target);
    if (e.meta.target === 'User_email_key') {
      throw new HttpException('EMAIL_ALREADY_TAKEN', HttpStatus.CONFLICT);
    } else if (e.meta.target === 'User_username_key') {
      throw new HttpException('USERNAME_ALREADY_TAKEN', HttpStatus.CONFLICT);
    }
  }
}
```

Questa API chiama la seguente funzione per procedere con la creazione effettiva dell'utente:

```
async createUser(user: signupUserType): Promise<User> {
  const newUser = await this.user.create({
    data: {
      username: user.username,
      email: user.email,
      password: user.password,
      name: user.name,
      surname: user.surname,
      settings: {
        create: {
          language: Languages.IT,
          visibility: AccountVisibility.public,
        },
      },
    },
  });

  return newUser;
}
```

Questa funzione viene chiamata solo se username e email non sono già stati utilizzati da un altro utente della piattaforma.
Se **username** oppure **email** sono già stati utilizzati da qualcun altro, viene lanciata un'eccezione all'applicativo di frontend (Http status **409**).
In caso di successo, viene ritornato http status **200**.

Login di un utente

La seguente API permette di autenticare un utente che e' stato precedentemente collegato. La funzione, dato in input **username** e **password**, valida l'utente e successivamente, tramite il **jwtService** genera un token da ritornare all'applicativo per autenticarsi nelle richieste successive.

```
async validateUser(username: string, password: string): Promise<any> {
  const user = await this.usersService.findOne(username);
  if (user && user.password === password) {
    const { password, ...result } = user;

    return result;
  }

  return null;
}

async login(user: any) {     ■ Unexpected any. Specify a different type.
  const payload = { username: user.username, id: user.id };

  return {
    access_token: this.jwtService.sign(payload),
    user: user,
  };
}
```

Creazione di un post

```
public async createNewPost(newPost: CreatePostDto, user: User) {
  try {
    const post = await this.post.create({
      data: {
        title: newPost.title,
        description: newPost.description,
        userId: user.id,
        deletedOn: null,
        visibility: newPost.visibility ?? PostVisibility.public,
      },
    });
    return post;
  } catch (exp) {
    throw new HttpException(exp + '', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Questa API permette invece di creare un nuovo Post, dopo che un utente loggato clicca su “Crea nuovo post”. Richiede come parametri il contenuto del **post** e l’utente che ha effettuato la richiesta (cioe’ quello loggato).
In caso di errore viene lanciata eccezione con http status **500** e il relativo errore.
In caso di successo il back end restituisce il **post** creato e http status **201**.

Lista dei post creati

```
public async getAllPosts(page: number, size: number, user:User) {
  if (isNaN(page) || isNaN(size)) {
    throw new HttpException(
      'Page or size are not valid',
      HttpStatus.BAD_REQUEST,
    );
  }

  try {
    let userFriends: string[] = [];
    if (user) {
      const loggedUser = await this.user.findUnique({
        where: {
          id: user.id,
        },
      })

      userFriends = !loggedUser ? [] : loggedUser.friends;
    }

    const posts = await this.post.findMany({
      skip: page * size,
      take: size,
      where: {
        OR: [
          { visibility: PostVisibility.public },
          {
            user: {
              id: { in: userFriends },
            },
          },
        ],
        NOT: {
          visibility: PostVisibility.hidden,
        },
      },
    });

    return posts;
  } catch (exp) {
    throw new HttpException(exp + '', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Questa API permette la visualizzazione dei post caricati nella piattaforma. Si può notare come nella ricerca vengono inclusi anche i **post** nascosti dell'utente loggato, se colui che ha creato il post e' lo stesso che lo sta visualizzando.
Quest' API e' stata implementata con la "**paginazione**", cioè dato un numero di pagina e un numero di elementi, ritorna gli elementi corrispondenti di quella pagina. (Per esempio, se mettiamo numero di elementi = 5 e in totale abbiamo 50 elementi, saranno in automatico gestiti in 10 pagine. Se facciamo pagine da 10, avremmo 5 pagine totali. Questo viene gestito in modo automatico e dinamico in base ai dati di input).

Cancellazione di un post

Questa API permette la cancellazione di un **post** precedentemente creato.
La funzione accetta come input l'**id** univoco del post e l'**utente** correntemente loggato.
Se sia l'utente che il post esistono, viene ritornato l'http status **200**.
Altrimenti viene ritornata l'eccezione NOT FOUND (**404**)
.

```
public async deletePost(postId: string, user: User) {
  try {
    const deletePost = await this.post.deleteMany({
      where: {
        id: postId,
        userId: user.id,
      },
    });

    if (deletePost.count === 0) {
      throw new HttpException('Not found', HttpStatus.NOT_FOUND);
    }

    return deletePost;
  } catch (exp) {
    throw new HttpException('Not found', HttpStatus.NOT_FOUND);
  }
}
```

Aggiornamento di un post esistente

La seguente API permette, dato in input il nuovo contenuto di un post, il suo id e l'utente correntemente loggato, di modificare tutti i dati relativi ad un post (titolo, descrizione, ecc).
In caso di avvenuta modifica, l'applicazione ritorna http **200**.
In caso di post non trovato, viene ritornata l'eccezione http **404**.
Per qualsiasi altro errore viene invece ritornato http **500**.

```
public async updatePost(newPost: CreatePostDto, postId: string, user: User) {
  try {
    let queryArgs: {
      where: {
        id: string;
        userId: string;
      };
      data: CreatePostDto;
    } = {
      where: {
        id: postId,
        userId: user.id,
      },
      data: {
        title: newPost.title,
        description: newPost.description,
      },
    };

    if (newPost.visibility) {
      queryArgs = {
        where: {
          ...queryArgs.where,
        },
        data: {
          ...queryArgs.data,
          visibility: newPost.visibility,
        },
      };
    }

    const updatePost = await this.post.updateMany({
      ...queryArgs,
    });

    if (updatePost.count < 1) {
      throw new HttpException('Post not found', HttpStatus.NOT_FOUND);
    }

    return {
      statusCode: HttpStatus.OK,
      message: 'Post updated successfully!',
    };
  } catch (exp) {
    throw new HttpException(exp + '', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Ricerca di post

La seguente API permette di ricercare dei **post** in base ad una stringa, che puo' essere contenuta sia nel titolo che nella descrizione dello stesso.

Questa API fa sempre uso della paginazione, descritta precedentemente nell'API di fetch dei post.
La query a database controlla la presenza della stringa prima nel titolo e poi nella descrizione.

In caso di errore viene ritornata l'eccezione http **500**.
In caso di successo http **200**.

```
public async searchPost(searchParams: SearchPostParamsDto) {
  try {
    if (!searchParams.searchQuery) {
      return new HttpException(
        'Provide search params',
        HttpStatus.BAD_REQUEST,
      );
    }

    const pageSize = searchParams.pageSize ?? 5;
    const page = searchParams.page ?? 0;

    const posts = await this.post.findMany({
      skip: page * pageSize,
      take: pageSize,
      where: {
        OR: [
          { title: { contains: searchParams.searchQuery } },
          { description: { contains: searchParams.searchQuery } },
        ],
      },
    });

    return posts;
  } catch (exp) {
    throw new HttpException(exp + '', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Fetch delle impostazioni dell'utente

Questa API, dato in input l'utente loggato, permette di ottenere le impostazioni del profilo (come visibilita', lingua preferita e avatar).
L'API ritorna errore se l'utente non viene trovato (**404**), oppure va in success altrimenti (**200**).

```
async getUserSettings(user: User): Promise<UserSettings | null> {
  try {
    const userSettings = await this.userSettings.findUniqueOrThrow({
      where: {
        userId: user.id,
      },
    });

    return userSettings;
  } catch (NotFoundError) {
    throw new HttpException('User not found!', HttpStatus.NOT_FOUND);
  }
}
```

Aggiornamento della lingua dell'utente

Questa API permette all'utente di cambiare la propria lingua preferita.
La funzione accetta l'utente loggato e un ENUM che rappresenta la lingua.
L'enum puo' avere 2 valori:

1. **IT**.
2. **EN**.

In caso di errore restituisce http **500**. In caso di successo http **200**.

```
async setUserLanguage(user: User, language: Languages) {
  try {
    await this.userSettings.updateMany({
      where: {
        userId: user.id,
      },
      data: {
        language: language,
      },
    });

    return await this.userSettings.findUniqueOrThrow({
      where: {
        userId: user.id,
      },
    });
  } catch (exp) {
    throw new HttpException(exp + '', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Aggiornamento della visibilita' dell'utente

Questa API permette all'utente di cambiare la propria visibilita' del profilo.
La funzione accetta l'utente loggato e un ENUM che rappresenta la visibility.
L'enum puo' avere 2 valori:

3. **public**.
4. **hidden**.

In caso di errore restituisce http **500**. In caso di successo http **200**.

```
async setUserVisibility(user: User, visibility: AccountVisibility) {
  try {
    await this.userSettings.updateMany({
      where: {
        userId: user.id,
      },
      data: {
        visibility: visibility,
      },
    });

    return await this.userSettings.findUniqueOrThrow({
      where: {
        userId: user.id,
      },
    });
  } catch (exp) {
    throw new HttpException(exp + '', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Fetch e aggiornamento dell avatar utente

Gli avatar degli utenti sono gestiti da un tool chiamato **Dicebear**, che permette di creare avatar randomici in base al nome utente, in modo tale che ogni utente possa avere un'immagine diversa in automatico.
L'API per settare l'avatar richiede l'utente correntemente loggato e un seed che verra' utilizzato per generare l'avatar randomico.
In caso di errore restituisce http **500**. In caso di successo http **200**.

L'API per ottenere l'avatar utente richiede solamente il seed (un seed genera sempre lo stesso avatar. Questo permette di passare alla funzione il nome utente dell'utente che ci interessa per ottenere il relativo avatar).

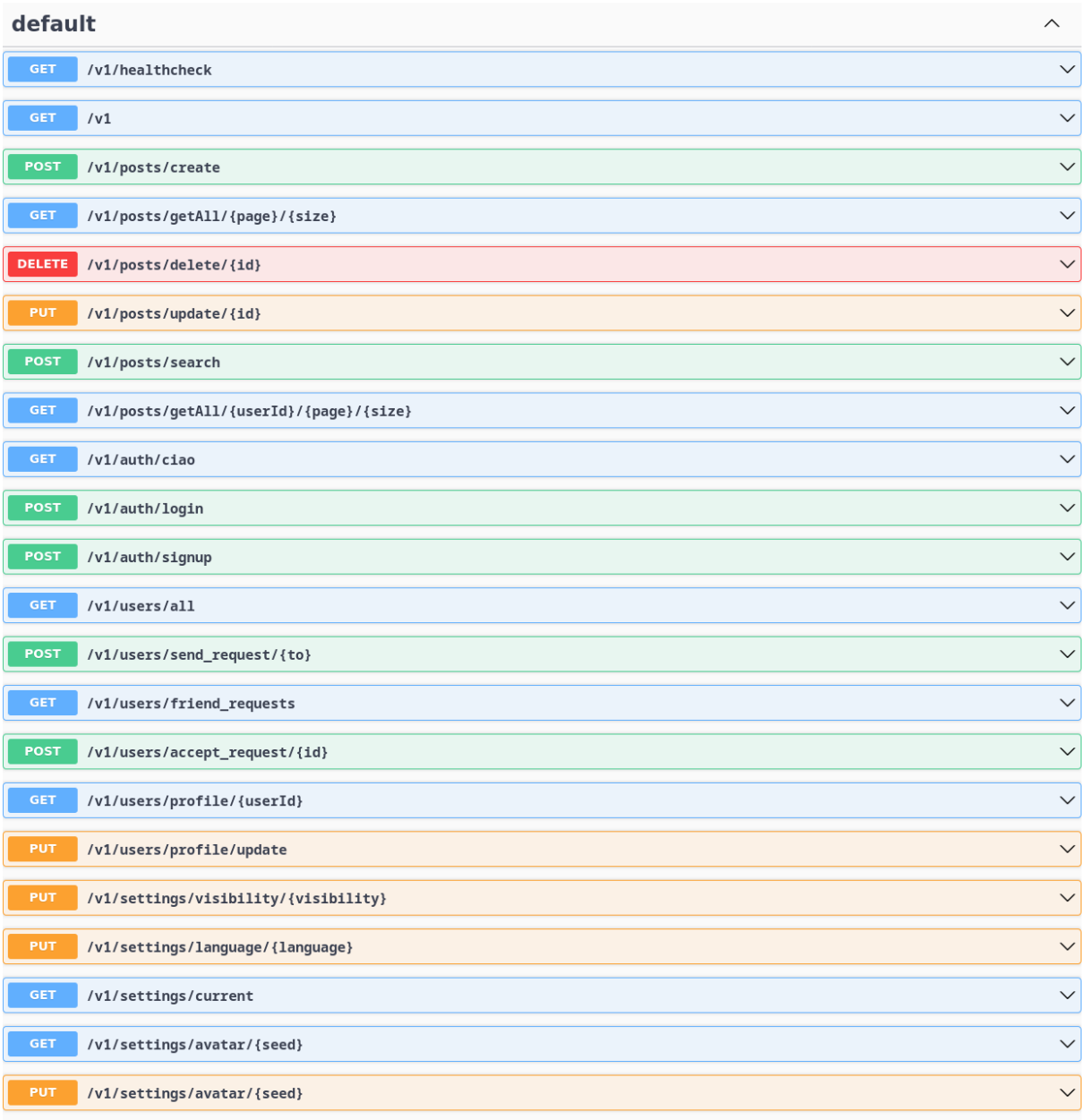
```
async getAvatar(seed: string) {
  const dicebear = await import('@dicebear/core');
  const collection = await import('@dicebear/collection');

  const createAvatar = dicebear.createAvatar;
  const micah = collection.micah;

  const avatar = createAvatar(micah, {
    seed: seed,
  });
  return avatar;
}

async setAvatar(user: User, seed: string) {
  try {
    await this.userSettings.updateMany({
      where: {
        userId: user.id,
      },
      data: {
        avatar: seed,
      },
    });

    return await this.userSettings.findUniqueOrThrow({
      where: {
        userId: user.id,
      },
    });
  } catch (err) {
    throw new HttpException(err + '', HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

APIs Documentation

Swagger APIs

Le API locali sono state descritte utilizzando **Swagger**, una suite di tools che permette di descrivere e testare le API definite nel progetto.
E' possibile consultare tutte le API collegandosi a "***https://trilliumbackend-production.up.railway.app/api***".

Lo screenshot di seguito riportato e' la pagina principale creata da **Swagger** con le varie API esposte dal server.

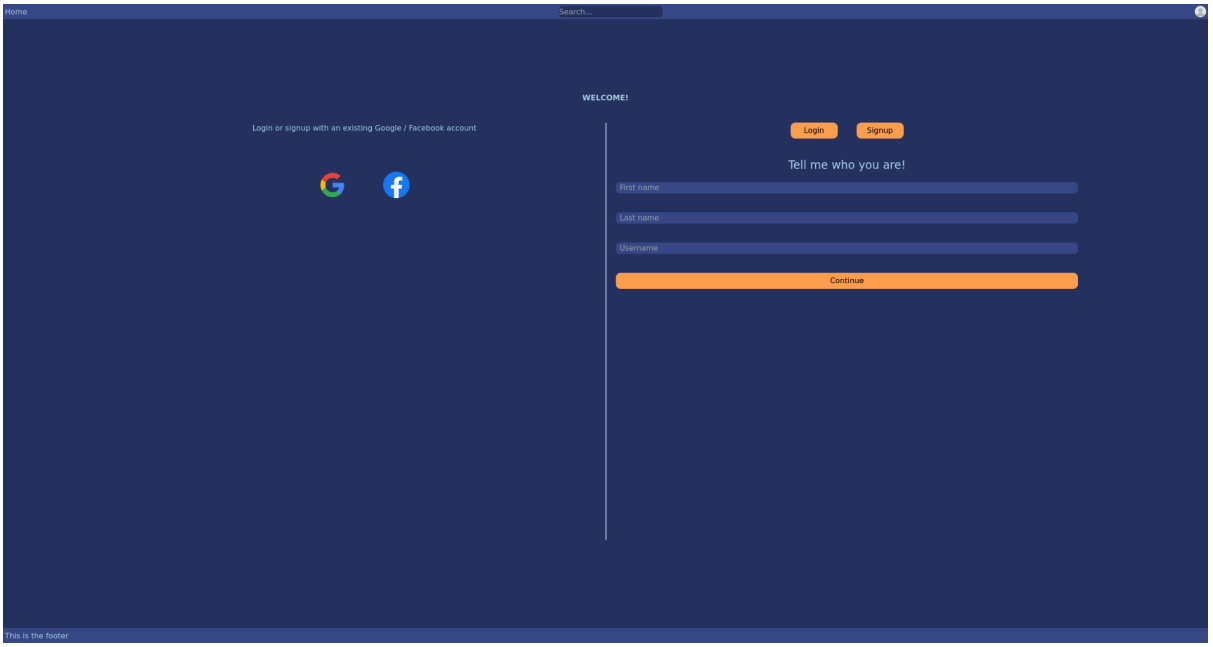


Come si può notare dallo screenshot tutte le API contengono una versione. Abbiamo utilizzato questa tecnica di sviluppo cosicché fosse facilmente versionare le API utilizzate dal frontend (che basta che aumenti il numero di versione. Se qualcosa non funziona si ritorna alla versione precedente semplicemente cambiando endpoint).

Front-end implementation

Il frontend dell'applicazione e' stato sviluppato in VueJs.

Pagina autenticazione

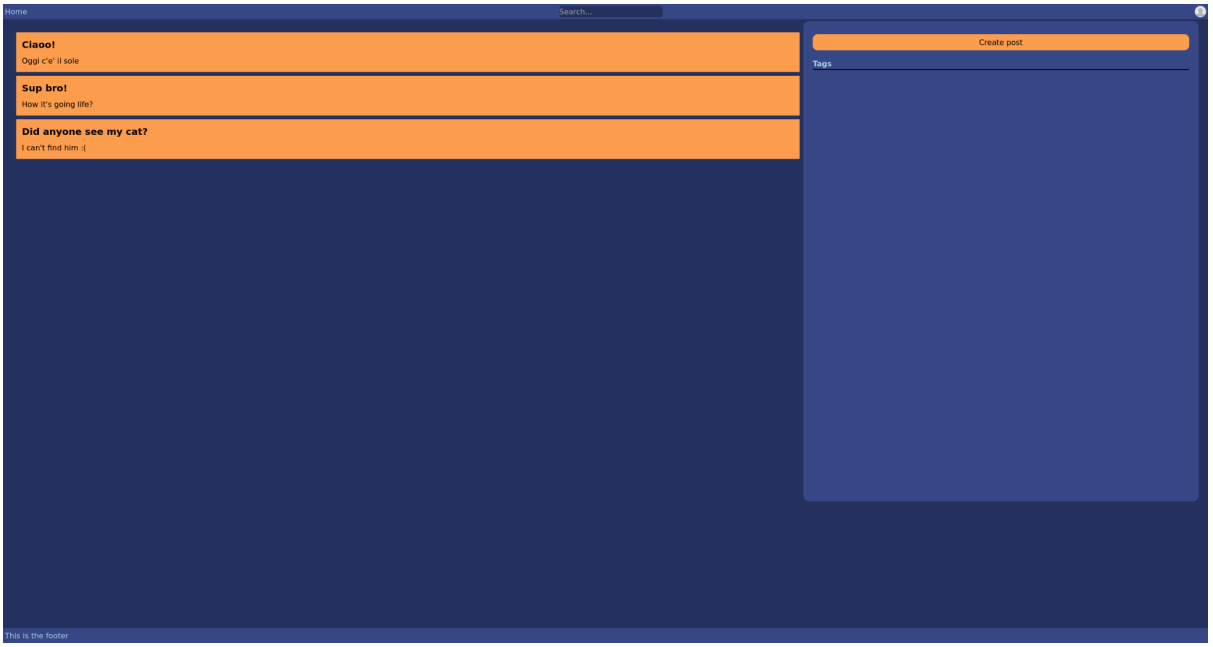


Questa pagina permette all'utente di registrarsi per poter caricare nuovi post, mandare amicizie e visualizzare post privati di altre persone.

I bottoni nella parte di destra permettono di passare dalla modalita' registrazione alla modalita' login, e viceversa.

Nella parte di sinistra risulta invece possibile effettuare il login / registrazione tramite le piattaforme "**Facebook**" e "**Google**".

Dashboard



La pagina dashboard mostra i post pubblici, privati dell’utente e quelli degli amici.
Sulla destra si trova il bottone per creare un nuovo post e la lista dei tag.

Nella parte di sinistra e’ presente la lista dei post.

APIs Testing

Per lo sviluppo e l’esecuzione dei test e’ stato utilizzato **Jest**.
Di seguito alcune immagini di codice ed esecuzione dei test.
Nota: i test non sono stati inseriti tutti per motivi di brevita’ del documento. Se si vogliono vedere nella loro completezza sono presenti nella repository.

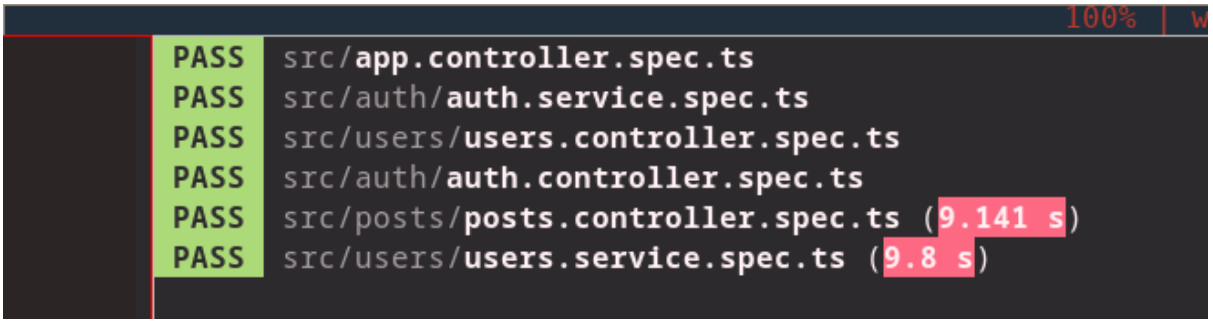
Modulo di test relativo al componente ***Users***.

```
describe('UsersService', () => {
  let service: UsersService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [UsersService],
    }).compile();

    service = module.get<UsersService>(UsersService);
  });

  it('should be defined', () => {
    expect(service).toBeDefined();
  });

  describe('userCRUD', () => {
    const randNumber = Math.floor(Math.random() * 1000000);
    const user: signupUserType = {
      username: `unitTestUser${randNumber}`,
      surname: 'test',
      name: 'test',
      email: `unit${randNumber}@test.com`,
      password: 'unitest99',
    };
    let userId: string;

    it('should create user', async () => {
      const registerResponse = await service.createUser(user);

      expect(registerResponse.email).toBe(user.email);
      expect(registerResponse.password).toBe(user.password);
      expect(registerResponse.username).toBe(user.username);
      expect(registerResponse.name).toBe(user.name);
      expect(registerResponse.surname).toBe(user.surname);
      userId = registerResponse.id;
    });

    it('should get user', async () => {
      const findUserResponse = await service.findOne(user.username);

      expect(findUserResponse).toBeTruthy();

      // Only because lsp is trying to make me destroy the pc
      if (findUserResponse) {
        expect(findUserResponse.email).toBe(user.email);
        expect(findUserResponse.password).toBe(user.password);
        expect(findUserResponse.username).toBe(user.username);
        expect(findUserResponse.name).toBe(user.name);
        expect(findUserResponse.surname).toBe(user.surname);
      }
    });

    it('should get user info', async () => {
      const findUserResponse = await service.getUserProfile(userId);

      expect(findUserResponse).toBeTruthy();
```

Modulo relativo ai test del componente ***Posts***.

```
describe('PostController', () => {
  let postService: PostsService;

  let userService: UsersService;

  beforeEach(() => {
    postService = new PostsService();
    userService = new UsersService();
  });

  it('Should be defined', () => {
    expect(postService).toBeDefined();
    expect(userService).toBeDefined();
  });

  const randNumber = Math.floor(Math.random() * 1000000);
  const userData: signupUserType = {
    username: `unitTestUser${randNumber}`,
    surname: 'test',
    name: 'test',
    email: `unit${randNumber}@test.com`,
    password: 'unitest99',
  };
  let user: User;

  it('should create user', async () => {
    user = await userService.createUser(userData);

    expect(user.email).toBe(userData.email);
    expect(user.password).toBe(userData.password);
    expect(user.username).toBe(userData.username);
    expect(user.name).toBe(userData.name);
    expect(user.surname).toBe(userData.surname);
  });

  const postData: CreatePostDto = {
    title: `Test ${randNumber}`,
    description: 'Test description',
    visibility: PostVisibility.hidden,
  };

  let post;     ■ Variable 'post' implicitly has an 'any' type, but a better type m
  it('should create post', async () => {
    post = await postService.createNewPost(postData, user);

    expect(post.title).toBe(postData.title);
    expect(post.description).toBe(postData.description);
    expect(post.visibility).toBe(postData.visibility);
  });

  it('should delete created post', async () => {
    const deletedPost = await postService.deletePost(post.id, user);

    expect(deletedPost.count).toBe(1);
  });
});
```

Esempio di esecuzione dei test da CLI.



Definizione di alcuni casi di test

Nella seguente sezione sarà presente una breve descrizione di una parte dei test fatti e dei loro esiti.

- **Should Create User**: Dati dei dati randomici, questo test crea un nuovo utente e tenta l'inserimento nel database. Questo test viene passato solo se l'utente viene effettivamente creato nel DB e se i dati ritornati dal DB corrispondono a quelli inseriti durante la creazione.
- **Should get UserInfo**: Dato una stringa che rappresenta un'id, deve essere preso da DB il relativo utente e la response generata deve esistere (non essere *undefined* oppure *null*).
- **Should Update User**: Il seguente test preleva da DB un utente randomico e tenta di aggiornare mail e descrizione. Il test risulta superato solo se dopo una select dal DB dello stesso utente, i dati ritornati corrispondono a quelli aggiornati in precedenza.
- **Should Create Post**: Dato un utente randomico e dei dati per un post randomico, questo test tenta di creare un post. Il test e' superato solo se i dati ritornati dal DB sono gli stessi utilizzati per l'inserimento del post stesso.
- **Should Delete Post**: Dato l'id di un post e l'utente che l'ha creato, questo test deve tentare l'eliminazione del post indicato. Il test viene superato se e solo se la query del DB ritorna "1", cioè il numero di post eliminati con quell'operazione.

Gli altri test funzionano in modo simile, ma relativo ad altri componenti. Risulta inutile descriverli tutti nel presente documento.

GitHub Repository and Application Deployment

Struttura repository Github

Il progetto è diviso in 2 repository github distinte: **trillium_backend** e **trillium-app**.
Le due repository contengono i rispettivi progetti.

Le task e le issue del progetto sono state gestite tramite il sistema interno di Github, che permette di collegare i branch alle issue, e vedere a chi sono assegnate.

Deployment

Il deployment e' stato fatto attraverso la piattaforma **Railway** per il progetto di backend.
Per la parte di frontend è stata invece utilizzata la piattaforma **Vercel**.

I progetti sono raggiungibili ai seguenti link:

- ***https://trilliumbackend-production.up.railway.app/api***
- ***https://trillium-app.vercel.app/auth***