Sviluppo Applicazione

Edoardo Ghirardello, Giulio Cappelli, Elia Casotti

Gruppo T42

2022

Indice

1 Scopo del documento

Nel documento corrente vengono riportate ulteriori e definitive informazioni riguardo allo sviluppo dell'applicazione Fen Festa.
Nello specifico, presenta tutti gli artefatti necessari per il login, la ricerca e la creazione degli eventi dell'applicazione. In partenza viene analizzato lo User-flow legato ad un utente registrato dell'applicazione, dopodiché vengono analizzate le API (tramite l'API Model) per la creazione, modifica e login di un profilo, le strutture dati, la visualizzazione, creazione e modifica di un evento necessari a Fen Festa.

Per ogni API che è stata utilizzata, vengono presentate descrizione delle funzionalità, documentazione e test utilizzati

In ultima istanza vengono fornite informazioni del Git Repository e, infine, il deployment dell'applicazione.

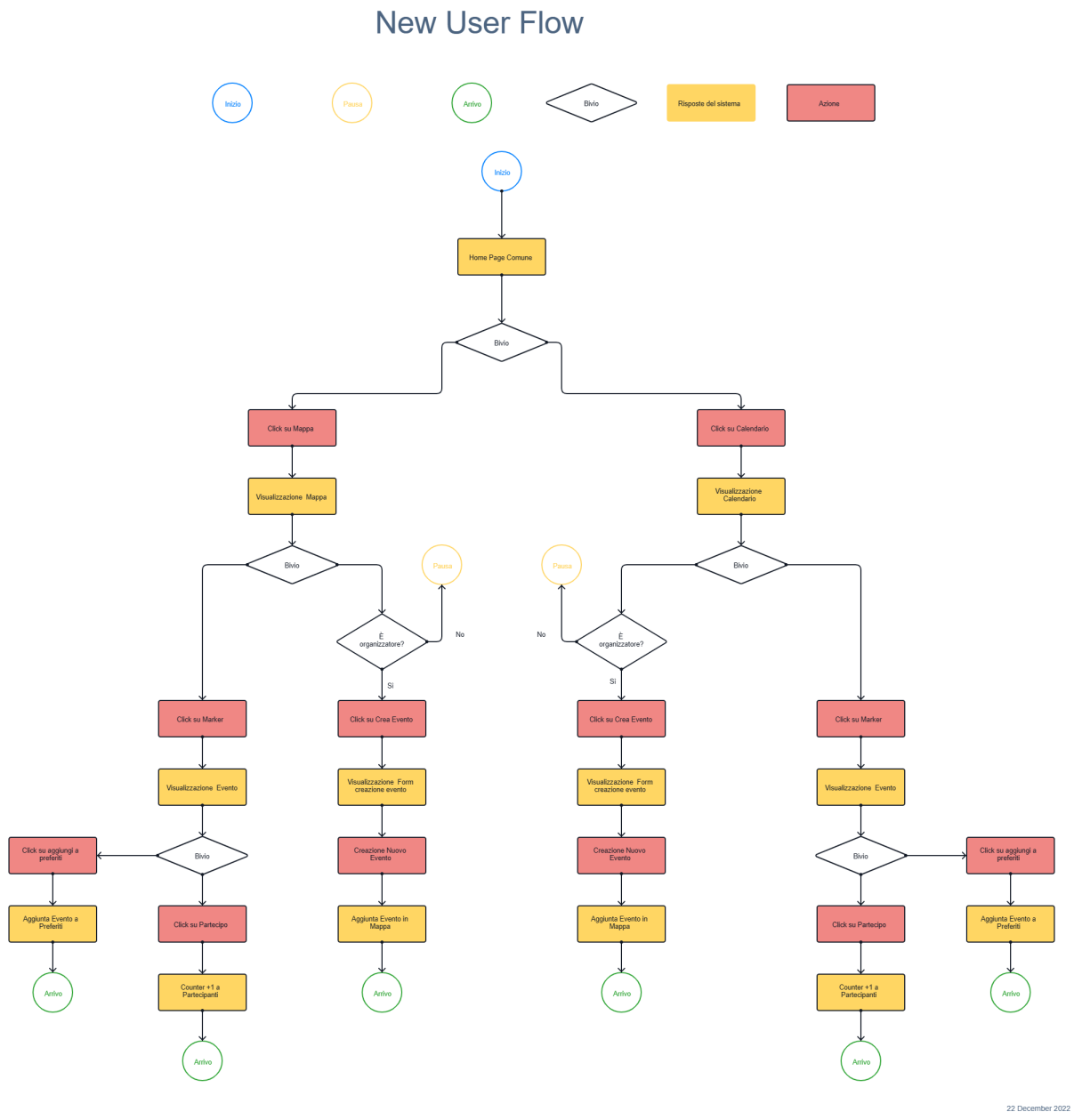
2 User-Flows

In questa sezione vengono riportati gli “user-flows” dell’utente registrato (che quindi può anche essere organizzatore o admin) della nostra applicazione.

In Figura 1 viene mostrato lo user-flow relativo alle azioni disponibili dall’homepage dagli utenti: visualizzazione, partecipazione e aggiunta preferiti degli eventi; in aggiunta anche la creazione evento se l’utente ha i permessi da organizzatore.

L’utente può visualizzare gli eventi tramite mappa o calendario e aggiungerli ai preferiti, in entrambi i casi, una volta avvenuta la visualizzazione di questi ultimi. Lo schema utilizza la notazione “Pausa” quando l’azione non è disponibile e la notazione “Arrivo” quando viene raggiunta l’ultima azione disponibile per quel ramo.

Viene, inoltre, presentata una legenda che descrive i simboli utilizzati, sempre in Figura 1.



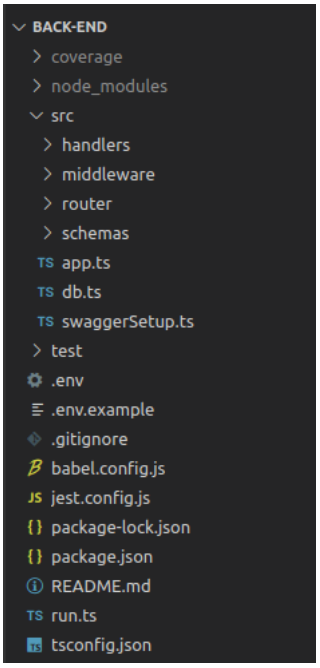
3 Application Implementation and Documentation

Nei precedenti documenti sono stati identificati tutti i requisiti funzionali e non funzionali dell'applicazione. Nella sezione precedente son state descritte le features che servono ad un Utente nel suo flusso applicativo.

Nella seguente sezione vengono analizzati i software e i tools di sviluppo di Fen Festa. L'applicazione è stata sviluppata utilizzando React (front-end) e Express (back-end). Per la gestione dei dati è stato utilizzato MongoDB.

3.1 Project Structure

Presentiamo in questa sezione la struttura del back-end, osservabile nell'immagine successiva (Figura 2).



Nella **root** del progetto troviamo il file principale che esegue il server e la connessione al database, avvalendosi di vari file presenti nella cartella **src**.

Sempre nella **root** sono presenti i vari file di configurazione del server necessari alla corretta esecuzione del server e del testing di cui parleremo in una successiva sezione.

La cartella **src** contiene i due file principali eseguiti dal main **run.ts**, **app.ts** e **db.ts**, e una serie di sottocartelle.

Handlers contiene le funzioni che elaborano le richieste costruendo le risposte da inviare al front-end.

Middleware contiene funzioni di supporto agli handlers.

Router contiene i router per gestire le richieste in arrivo e invocare l'handler appropriato.

Schemas contiene le strutture date utilizzate.

3.2 Project Dependencies

Sono stati utilizzati e aggiunti al file **package.json** e seguenti moduli Node:

- bcrypt
- dotenv
- express
- jsonwebtoken
- mongoose
- swagger-jsdoc
- swagger-ui-express
- ts-node
- typescript

Sono presenti, sempre nel file **package.json**, le seguenti dipendenze di development:

- @babel/core
- @babel/preset-env
- @babel/preset-typescript
- @types/bcrypt
- @types/express
- @types/jest
- @types/jsonwebtoken
- @types/swagger-jsdoc
- @types/swagger-ui-express
- babel-jest
- jest
- node-fetch
- nodemon
- ts-jest

3.3 Project Data

Per la gestione dati dell'applicazione viene usato il database MongoDB Atlas. Sono state definite due strutture principali: **User** e **Evento**.

Il progetto utilizza inoltre un databese (e server) separato unicamente per il testing per evitare di mescolare deployment e testing.

Per descrivere documenti di tipo **User** è stata definita la seguente interfaccia (Figura 3):

```
1  import { Types, SchemaTypes, Schema, model } from "mongoose";
2
3  export interface UserInterface {
4    _id?: Types.ObjectId;
5    email: string;
6    password: string;
7    preferiti: Types.ObjectId[];
8    isAdm: boolean;
9    isOrg: boolean;
10   alias: string;
11   img: string;
12 }
13
14 const UserSchema = new Schema<UserInterface>({
15   email: { type: String, required: true },
16   password: { type: String, required: true },
17   preferiti: [{ type: SchemaTypes.ObjectId, ref: "Event" }],
18   isAdm: { type: Boolean, default: false },
19   isOrg: { type: Boolean, default: false },
20   alias: String,
21   img: String,
22 });
23
24 const User = model("User", UserSchema);
25 export { User };
26
```

Per descrivere documenti di tipo **Evento** è stata definita la seguente interfaccia (Figura 4):

```ts
import { Types, SchemaTypes, Schema, model } from "mongoose";

export interface EventoInterface {
  _id?: Types.ObjectId;
  idOwner: any;
  location: {
    _id?: Types.ObjectId;
    name: string;
    city: string;
    street: string;
    lat: number;
    lon: number;
  };
  dateStart: Date;
  dateFinish: Date;
  title: string;
  tags: string[];
  image: string;
  description: string;
  nParticipants: number;
}

interface LocationInterface {
  _id?: Types.ObjectId;
  name: string;
  city: string;
  street: string;
  lat: number;
  lon: number;
}

const LocationSchema = new Schema<LocationInterface>({
  name: { type: String, required: true },
  city: { type: String, required: true },
  street: { type: String, required: true },
  lat: { type: Number },
  lon: { type: Number },
});

const EventoSchema = new Schema<EventoInterface>({
  idOwner: { type: SchemaTypes.ObjectId, ref: "User" },
  location: { type: LocationSchema },
  dateStart: { type: Date, required: true },
  dateFinish: { type: Date, required: true },
  title: { type: String, required: true },
  tags: [{ type: String, required: true }],
  image: { type: String },
  description: { type: String, required: true },
  nParticipants: { type: Number, default: 0 },
});

const Evento = model("Evento", EventoSchema);
export { Evento };
```

Come si può notare è stata definita un'interfaccia di supporto **location** per definire la location dell'Evento.

3.4 Project APIs

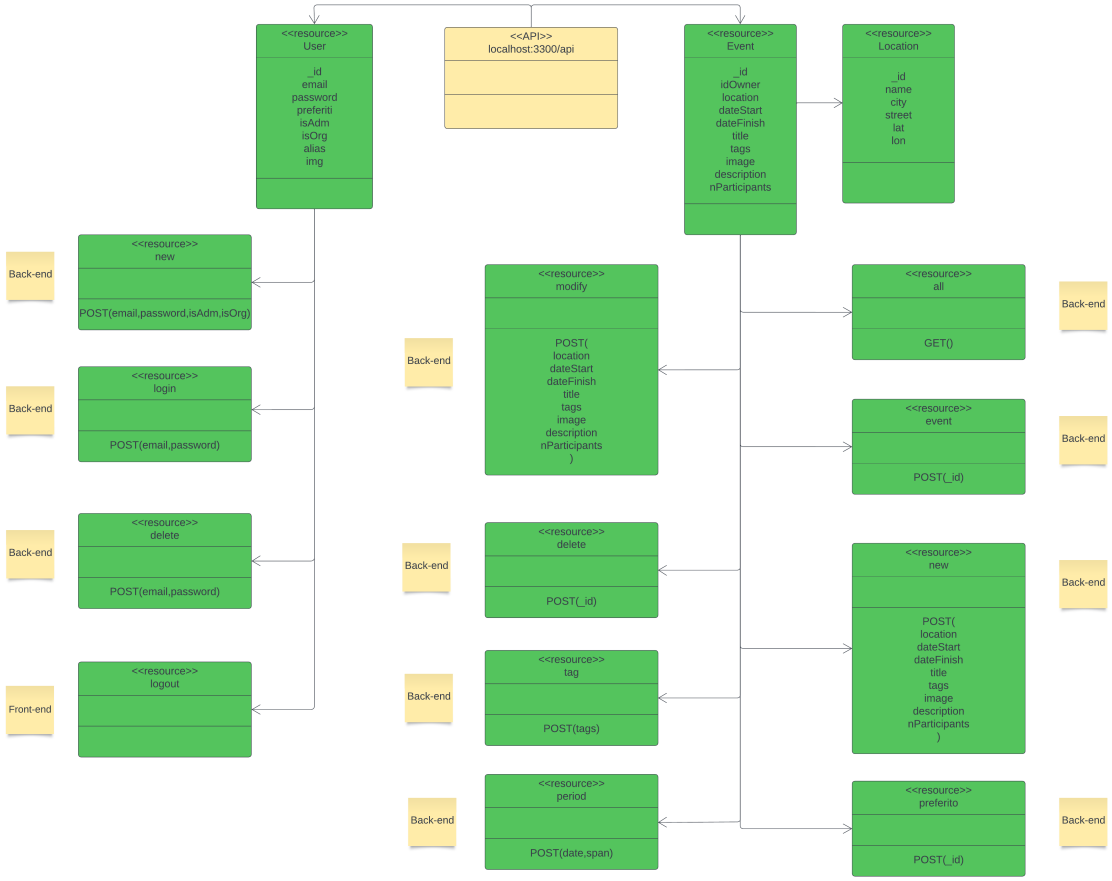
3.4.1 Resources Extraction from the Class Diagram

Dal Class Diagram sono state estratte le risorse rappresentate in Figura 5.

Identifichiamo **User** e **Evento** (con **Location** a supporto) da cui accediamo a varie richieste eseguibili dall'app.

Per **User** abbiamo lato back-end le risorse **new** (per creare un nuovo utente), **login** (per identificarci univocamente) e **delete** (per eliminare il nostro account), mentre lato front-end la risorsa **logout** (per effettuare il logout dall'app).

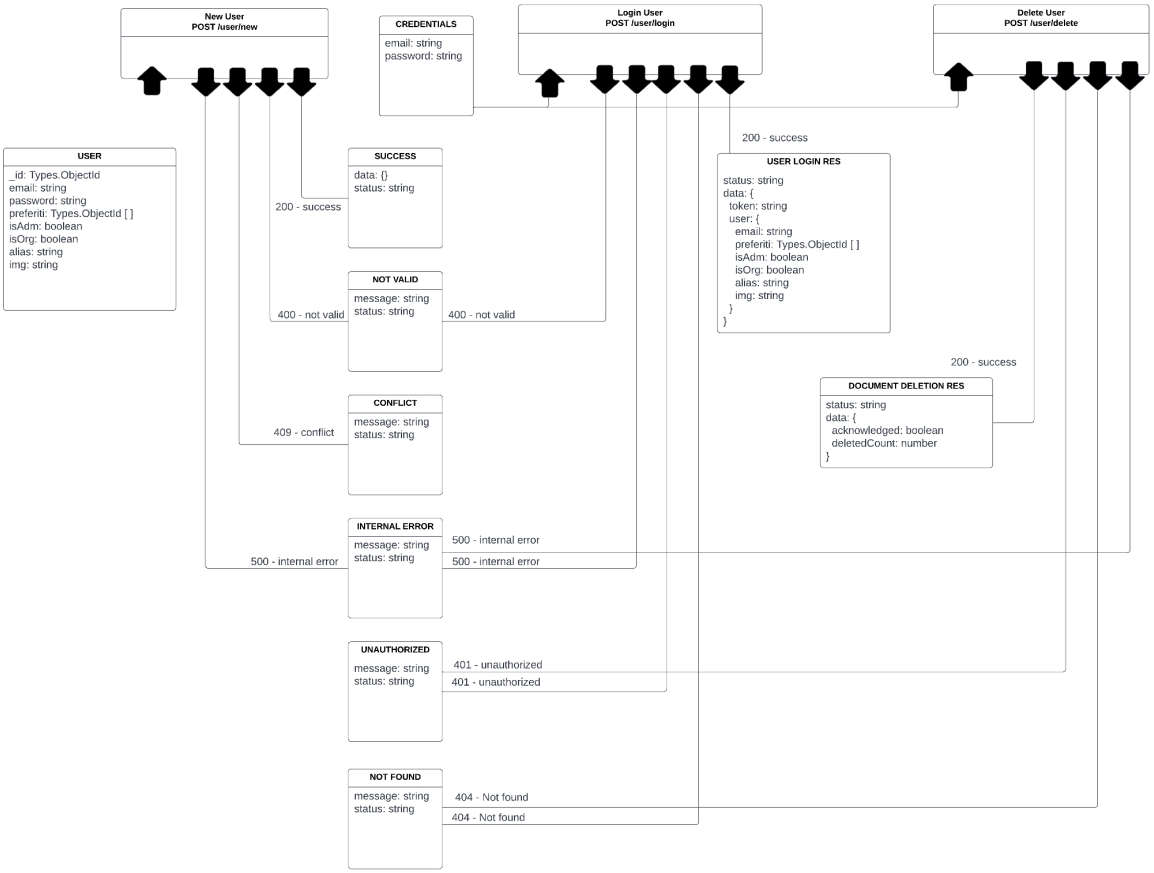
Per **Evento** abbiamo lato back-end le risorse **all** (per richiedere l'elenco completo degli eventi nel DB), **event** (per richiedere un singolo evento in base all'id), **new** (per creare un nuovo evento), **modify** (per modificare un evento esistente), **delete** (per eliminare un evento), **tag** (per effettuare una ricerca tramite tag), **period** (per effettuare una ricerca a livello giornaliero, settimanale o mensile), **preferito** (per aggiungere o rimuovere un evento dai preferiti).



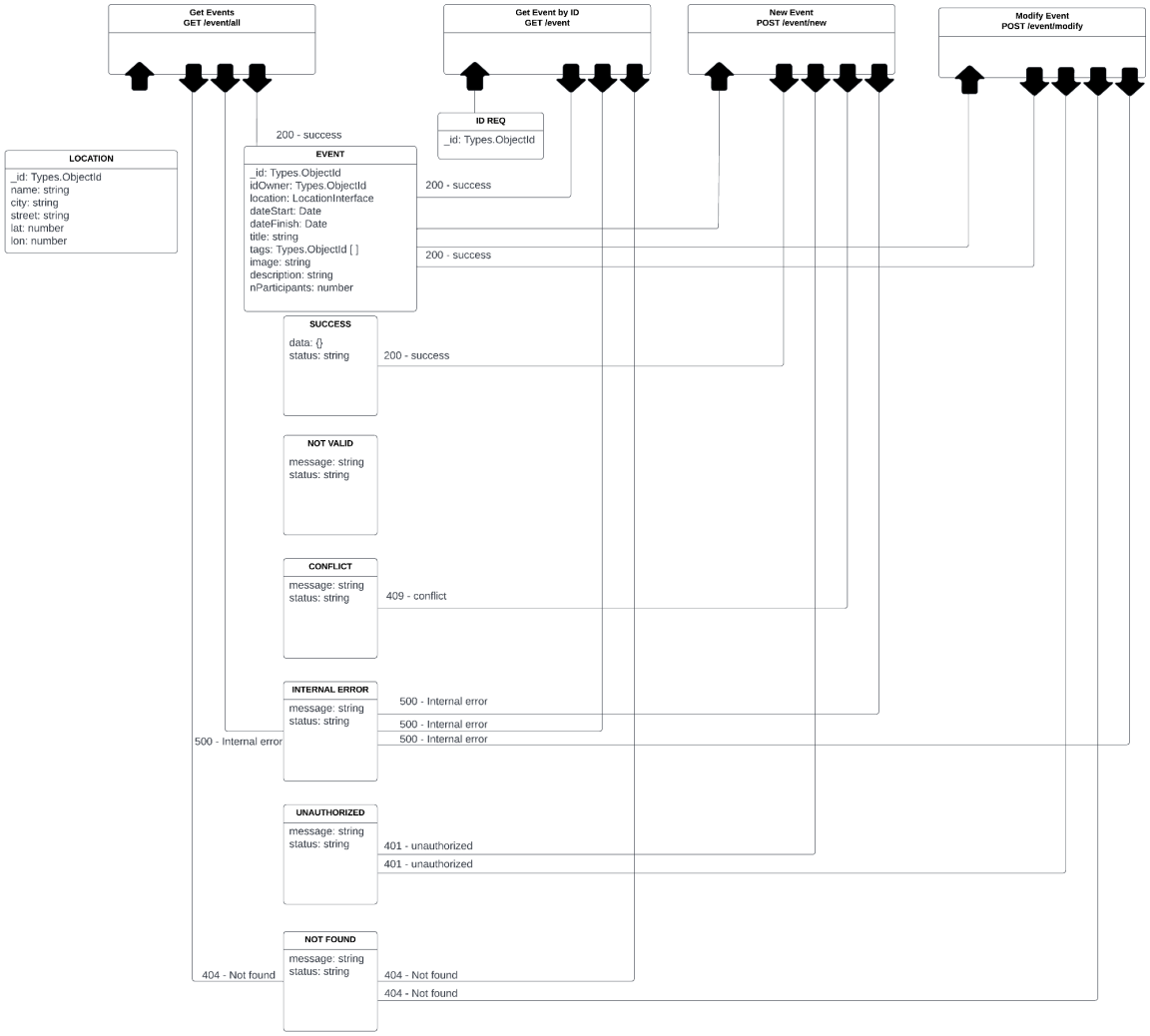
3.4.2 Resources Models

Identificate le risorse è stato costruito un diagramma delle API per identificare i request e response body e le varie tipologie di risposta che possiamo aspettarci dalle API.

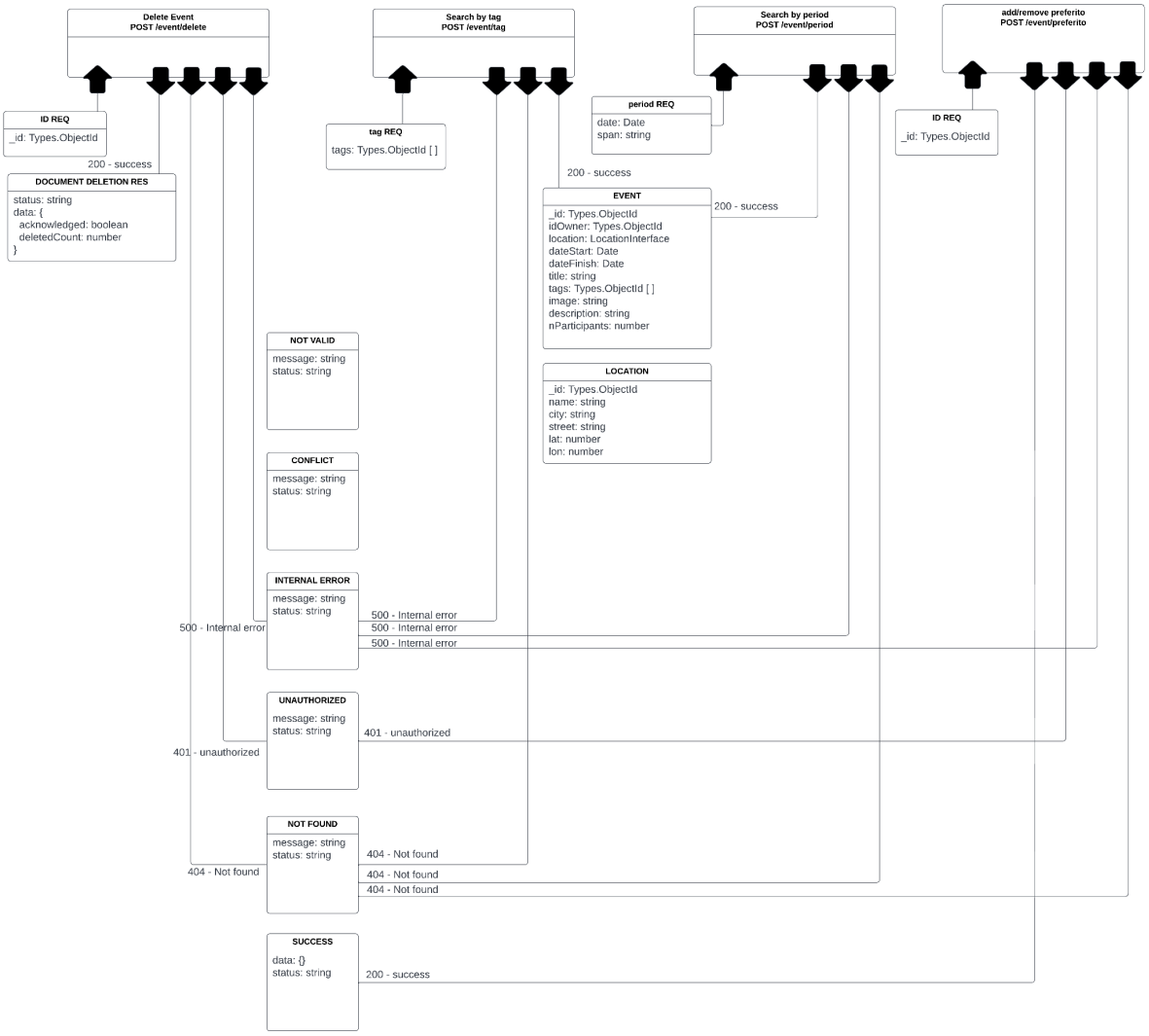
Osserviamo in Figura 6 il diagramma inerente agli User:



Osserviamo in Figura 7 il diagramma inerente agli Eventi (parte 1):



Osserviamo in Figura 8 il diagramma inerente agli Eventi (parte 2):



3.5 Sviluppo API

Le API vengono gestite dai router presenti nella cartella router dell'app.

```
import express from "express";
import { authToken } from "../middleware/token";
import { createUser, loginUser, deleteUser } from "../handlers/userHandle";

const userRouter = express.Router();
// NOT REQUIRE TOKEN
userRouter.post("/new", createUser);
userRouter.post("/login", loginUser);
// REQUIRE TOKEN
userRouter.use(authToken);
userRouter.post("/delete", deleteUser);
export default userRouter;
```

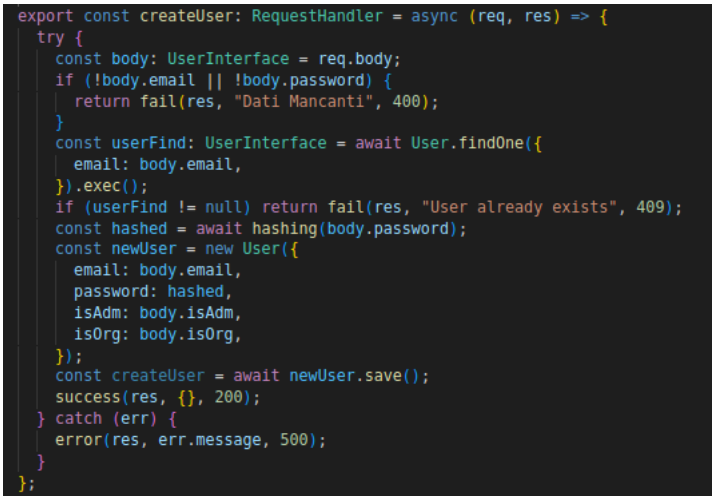
```
import express from "express";
import { authToken } from "../middleware/token";
import {
  createEvent,
  modifyEvent,
  deleteEvent,
  getEvents,
  getEvent,
  searchByTag,
  periodEvento,
  preferito,
} from "../handlers/eventHandle";

const eventRouter = express.Router();
eventRouter.get("/all", getEvents);
eventRouter.post("/", getEvent);
eventRouter.post("/tag", searchByTag);
eventRouter.post("/period", periodEvento);
// REQUIRE TOKEN
eventRouter.use(authToken);
eventRouter.post("/new", createEvent);
eventRouter.post("/modify", modifyEvent);
eventRouter.post("/delete", deleteEvent);
eventRouter.post("/preferito", preferito);
export default eventRouter;
```

3.5.1 Creazione nuovo User

Tramite questa API è possibile creare un nuovo utente, purché la mail fornita non sia già utilizzata da un altro utente (in questo caso l'app ci risponderà con un errore 409).

L'utente deve inviare sia una mail che una password, in caso contrario l'app risponderà con un errore 400.

In caso di successo riceveremo un messaggio di conferma con codice 200.



```
export const createUser: RequestHandler = async (req, res) => {
  try {
    const body: UserInterface = req.body;
    if (!body.email || !body.password) {
      return fail(res, "Dati Mancanti", 400);
    }
    const userFind: UserInterface = await User.findOne({
      email: body.email,
    }).exec();
    if (userFind != null) return fail(res, "User already exists", 409);
    const hashed = await hashing(body.password);
    const newUser = new User({
      email: body.email,
      password: hashed,
      isAdm: body.isAdm,
      isOrg: body.isOrg,
    });
    const createUser = await newUser.save();
    success(res, {}, 200);
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

3.5.2 Login User

Tramite questa API lo User può effettuare il login all'app utilizzando le credenziali con cui è stato creato l'utente.

Se la richiesta dovesse andare a buon fine l'app risponderà con i dati principali dell'utente (email, array dei preferiti, eventuali poteri superiori da admin o organizzatore) e con un token JWT che lo User utilizzerà per tutte quelle richieste che necessitano di verifica dei poteri posseduti.

L'utente deve inviare sia una mail che una password, in caso contrario l'app risponderà con un errore 400.

Se la password dovesse essere scorretta l'app risponderà con un messaggio di errore con codice 401.

Se non venisse trovato nessun utente identificato dall'email fornita l'app risponderà con un messaggio di errore con codice 404.

```
export const loginUser: RequestHandler = async (req, res) => {
  const body: {
    email: string;
    password: string;
  } = req.body;
  if (!body.email || !body.password)
    return fail(res, "Login credentials not submitted", 400);
  try {
    const userFind: UserInterface = await User.findOne({
      email: body.email,
    }).exec();
    if (userFind != null) {
      if (await compare(body.password, userFind.password)) {
        const token = generateToken({
          _id: userFind._id,
          sub: userFind.email,
          isAdm: userFind.isAdm,
          isOrg: userFind.isOrg,
        });
        const userResData = {
          email: userFind.email,
          preferiti: userFind.preferiti,
          isAdm: userFind.isAdm,
          isOrg: userFind.isOrg,
          alias: userFind.alias,
          img: userFind.img,
        };
        success(res, { token, userResData }, 200);
      } else {
        fail(res, "incorrect password", 401);
      }
    } else {
      fail(res, "not found", 404);
    }
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

3.5.3 Eliminazione User

Tramite questa API è possibile elimiare il proprio account inserendo mail e password (oppure, se si possiedono poteri da amministratore, eliminare un account fornendone solo la mail).

In caso di successo l'app risponderà con un messaggio di conferma con codice 200.

Essendo necesasrio il JWT per compiere questa operazione in caso di errore dell'elaborazione dello stesso riceveremo in risposta un messaggio di errore con codice 401.

Immettendo una email incorretta l'app, non trovando lo user, risponderà con un messaggio di errore con codice 404.

```
export const deleteUser: RequestHandler = async (req, res) => {
  const body: {
    email: string;
    password: string;
  } = req.body;
  const auth: JwtPayload = req.body.auth;
  const userFind: UserInterface = await User.findOne({
    email: body.email,
  }).exec();
  if (userFind == null) {
    return error(res, "user not found", 404);
  }
  try {
    if (auth.isAdm) {
      const deletedRes = await User.deleteOne({
        email: body.email,
      });
      success(res, deletedRes, 200);
    } else if (
      auth.sub == body.email &&
      (await compare(body.password, userFind.password))
    ) {
      const deletedRes = await User.deleteOne({
        email: body.email,
      });
      success(res, deletedRes, 200);
    } else {
      unauthorized(res);
    }
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

3.5.4 Stampa tutti gli Eventi

Tramite questa API è possibile ottenere l'elenco di tutti gli eventi salvati nel DB.

Nel caso in cui ci fossero effettivamente degli eventi salvati l'app ce li restituirebbe sotto forma di array.

In caso contrario riceveremo un messaggio di errore con codice 404.

```
export const getEvents: RequestHandler = async (req, res) => {
  try {
    const searchEvento: EventoInterface[] = await Evento.find({});
    if (searchEvento.length == 0) return error(res, "Not found", 404);
    success(res, searchEvento, 200);
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

3.5.5 Stampa singolo Evento

Tramite questa API è possibile ottenere informazioni riguardo ad un evento selezionandolo tramite ID.

Se l'ID identifica effettivamente un evento lo otterremo in risposta, in caso contrario riceveremo un messaggio di errore con codice 404.

```
export const getEvent: RequestHandler = async (req, res) => {
  const body: {
    _id: Types.ObjectId;
  } = req.body;
  try {
    const searchEvento: EventoInterface = await Evento.findOne({
      _id: body._id,
    })
      .populate("idOwner", "alias img")
      .exec();
    if (searchEvento == null) return error(res, "Not found", 404);
    success(res, searchEvento, 200);
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

3.5.6 Creazione nuovo Evento

Tramite questa API è possibile creare un nuovo evento fornendone i dati necessari se si dispone dell'autorizzazione necessaria (essere organizzatori).

In caso di successo si riceverà un messaggio di conferma con codice 200.

Se il token fornito per l'autorizzazione non ci identificasse come organizzatori l'app ci risponderebbe con un messaggio di errore con codice 401.

Non è possibile creare un evento con lo stesso titolo e la stessa esatta data di inizio, se stessimo cercando di creare un evento infrangendo questa regola l'app ci risponderebbe con un messaggio di errore con codice 409.

```
export const createEvent: RequestHandler = async (req, res) => {
  const body: EventoInterface = req.body;
  const auth: JwtPayload = req.body.auth;
  try {
    if (auth.isOrg) {
      const eventoFind: EventoInterface = await Evento.findOne({
        title: body.title,
        dateStart: body.dateStart,
      }).exec();
      if (eventoFind != null) return fail(res, "Event already exists", 409
      const newEvento = new Evento({
        idOwner: auth._id,
        location: {
          name: body.location.name,
          city: body.location.city,
          street: body.location.street,
          lat: body.location.lat,
          lon: body.location.lon,
        },
        dateStart: body.dateStart,
        dateFinish: body.dateFinish,
        title: body.title,
        tags: body.tags,
        image: body.image,
        description: body.description,
      });
      const createEvento = await newEvento.save();
      success(res, {}, 200);
    } else {
      unauthorized(res);
    }
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

3.5.7 Modifica Evento

Tramite questa API è possibile modificare un evento se ne siamo noi stessi il creatore (ci identifichiamo tramite JWT) o se siamo un admin. È necessario fornire i dati da modificati.

In caso di successo l'app ci restituirà l'evento modificato.

Se non possedessimo l'autorizzazione necessaria riceveremmo dall'app un messaggio di errore con codice 401.

Se ci fosse un errore con l'id fornito (che identifica l'evento nel DB) e quindi l'app non riuscisse a trovare nessun evento da modificare, riceveremmo in risposta un messaggio di errore con codice 404.

```
export const modifyEvent: RequestHandler = async (req, res) => {
  const body: EventoInterface = req.body;
  const auth: JwtPayload = req.body.auth;
  try {
    const searchEvento: EventoInterface = await Evento.findOne({
      _id: body._id,
    }).exec();
    if (searchEvento == null) return error(res, "Not found", 404);
    if (!(searchEvento.idOwner == auth._id || auth.isAdm)) {
      return unauthorized(res);
    }
    for (const key in body) {
      if (body.hasOwnProperty(key)) {
        searchEvento[key] = body[key];
      }
    }
    const modifiedEvento = new Evento(searchEvento);
    const createEvento = await modifiedEvento.save();
    success(res, searchEvento, 200);
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

3.5.8 Eliminazione Evento

Tramite questa API è possibile eliminare un nostro evento (o un evento qualsiasi se siamo admin).

La mancanza di autorizzazione necessaria comporta un messaggio di errore con codice 401 in risposta.

Se l'ID non dovesse corrispondere ad alcun evento riceveremmo in risposta un messaggio di errore con codice 404.

```
export const deleteEvent: RequestHandler = async (req, res) => {
  const body: {
    _id: Types.ObjectId;
  } = req.body;
  const auth: JwtPayload = req.body.auth;
  let owner: UserInterface;
  const eventoFind: EventoInterface = await Evento.findOne({
    _id: body._id,
  }).exec();
  if (eventoFind == null) {
    return error(res, "Not found", 404);
  }
  try {
    if (auth.isAdm) {
      const deletedRes = await Evento.deleteOne({
        title: eventoFind.title,
      });
      success(res, deletedRes, 200);
    } else if (auth.isOrg && auth._id == eventoFind.idOwner) {
      const deletedRes = await Evento.deleteOne({
        title: eventoFind.title,
      });
      success(res, deletedRes, 200);
    } else {
      unauthorized(res);
    }
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

3.5.9 Ricerca per Tag

Tramite questa API è possibile effettuare una ricerca per tag (singolo o molteplici).

La richiesta ha successo se riceviamo in risposta l'elenco degli eventi con almeno un tag presente nell'array tag.

Se l'elenco è vuoto la risposta sarà un messaggio di errore con codice 404.

```
export const searchByTag: RequestHandler = async (req, res) => {
  const body: {
    tags: string[];
  } = req.body;
  const eventoFind: EventoInterface[] = await Evento.find({
    tags: { $in: body.tags },
  });
  if (eventoFind[0] == null) {
    return error(res, "Not found", 404);
  }
  success(res, eventoFind, 200);
};
```

3.5.10 Ricerca per Periodo

Tramite questa API è possibile effettuare una ricerca per periodo, ovvero fornita una data base è possibile richiedere all'app tutti gli eventi che iniziano o finiscono a partire da quella data fino al giorno/settimana/mese successivo.

Come per la ricerca tramite tag o riceveremo un elenco di eventi o un messaggio di errore con codice 404.

```
export const periodEvento: RequestHandler = async (req, res) => {
  const body: {
    date: string;
    span: string;
  } = req.body;
  const date = body.date;
  let ddate: number;
  try {
    const baseDate = new Date(date);
    switch (body.span) {
      case "day":
        ddate = baseDate.getTime() + 86400000;
        break;
      case "week":
        ddate = baseDate.getTime() + 604800000;
        break;
      case "month":
        ddate = baseDate.getTime() + 86400000 * 30;
        break;
      default:
        break;
    }
    const eventoFind: EventoInterface[] = await Evento.find()
      .or([
        { dateFinish: { $gte: date, $lte: ddate } },
        { dateStart: { $gte: date, $lte: ddate } },
      ])
      .sort({ dateStart: "asc" });
    if (eventoFind[0] == null) return error(res, "Not found", 404);
    success(res, eventoFind, 200);
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

3.5.11 Aggiunta/Rimozione dai Preferiti

Tramite questa API è possibile aggiungere (se non è già presente) o rimuovere (se già è presente) un evento alla lista preferiti dell'utente.

Un errore con il token di autenticazione comporta un messaggio di errore con codice 401.

Se l'ID evento fornito non dovesse esistere riceveremmo un messaggio di errore con codice 404.

```
export const preferito: RequestHandler = async (req, res) => {
  const body: {
    _id: Types.ObjectId;
  } = req.body;
  const auth: JwtPayload = req.body.auth;
  try {
    const eventoFind: EventoInterface = await Evento.findOne({
      _id: body._id,
    }).exec();
    if (eventoFind == null) {
      return error(res, "Not found", 404);
    }
    const userFind: UserInterface = await User.findOne({
      _id: auth._id,
    }).exec();
    const index = userFind.preferiti.indexOf(eventoFind._id);
    if (index > -1) {
      userFind.preferiti.splice(index, 1);
      const modifiedUser = new User(userFind);
      const createUser = await modifiedUser.save();
      eventoFind.nParticipants = eventoFind.nParticipants - 1;
      const modifiedEvento = new Evento(eventoFind);
      const createEvento = await modifiedEvento.save();
      return success(res, {}, 200);
    }
    if (index == -1) {
      userFind.preferiti.push(eventoFind._id);
      const modifiedUser = new User(userFind);
      const createUser = await modifiedUser.save();
      eventoFind.nParticipants = eventoFind.nParticipants + 1;
      const modifiedEvento = new Evento(eventoFind);
      const createEvento = await modifiedEvento.save();
      return success(res, {}, 200);
    }
  } catch (err) {
    error(res, err.message, 500);
  }
};
```

4 API Documentation

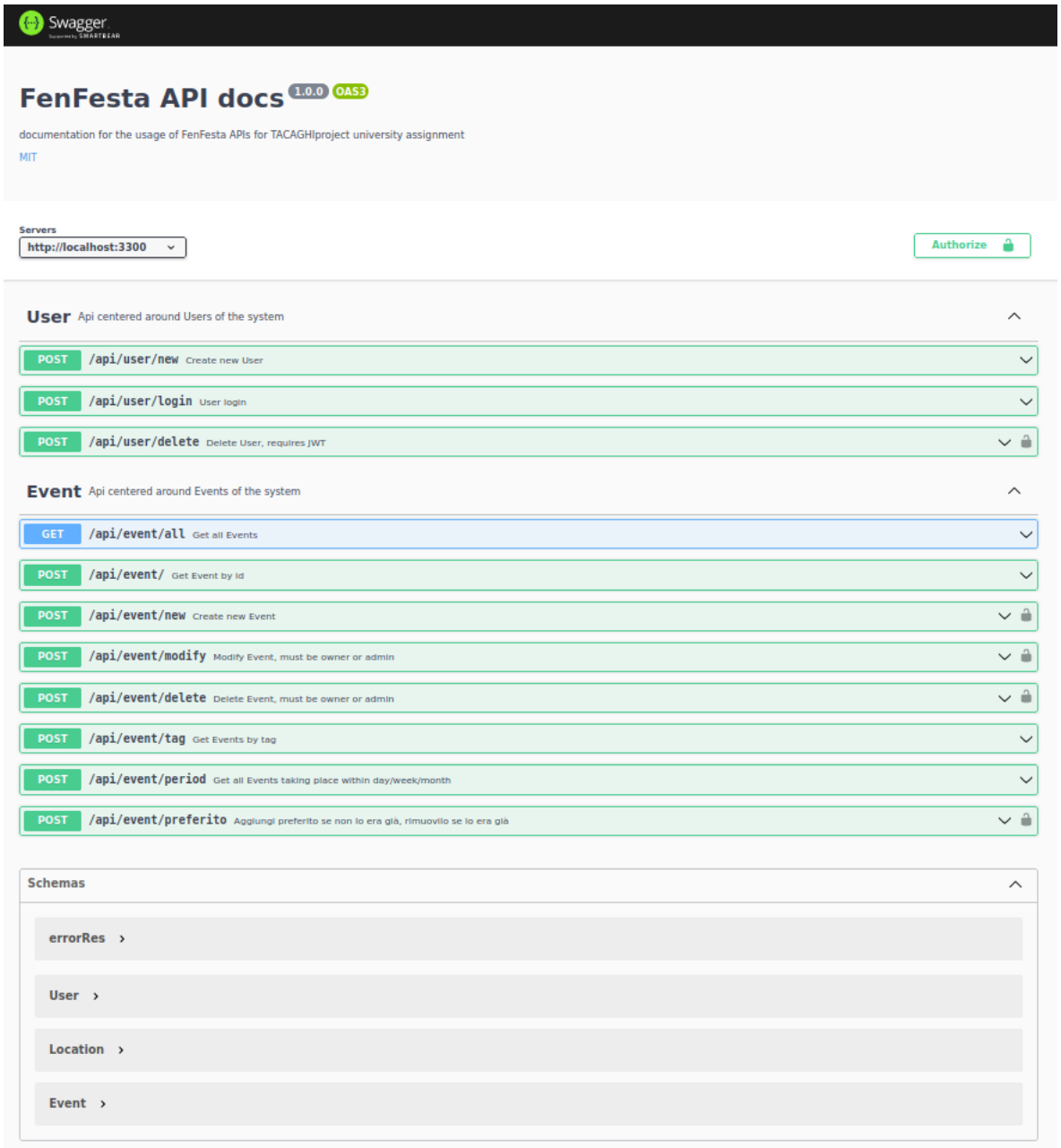
Le API Locali, descritte nella sezione precedente, fornite dall'applicazione FenFesta sono state documentate utilizzando il modulo NodeJS chiamato Swagger UI Express.

Così facendo, aprendo la pagina designata, il reperimento della documentazione relativa alle suddette API è facilmente consultabile da qualunque persona.

Per poter generare l'endpoint dedicato alla presentazione delle API è stato utilizzato Swagger UI in quanto crea una pagina web dalle definizioni delle specifiche OpenAPI.

Di seguito (Figura 9) viene mostrata la pagina web relativa alla documentazione che presenta le API utilizzabili dall'utente generico per la visualizzazione degli eventi, partecipazione e salvataggio dell'evento, oltre a quelle utilizzabili da un utente con autorizzazioni più elevate per la creazione, modifica e eliminazione di eventi.

L'endpoint da invocare per raggiungere la seguente documentazione è: `http://localhost:3300/api-docs`.



Tramite la documentazione Swagger è possibile eseguire tutte le API fornite dall'app, comprese quelle che richiedono autenticazione tramite JWT (accessibile inserendo un token valido premento su Authorize).

5 Front-End Implementation

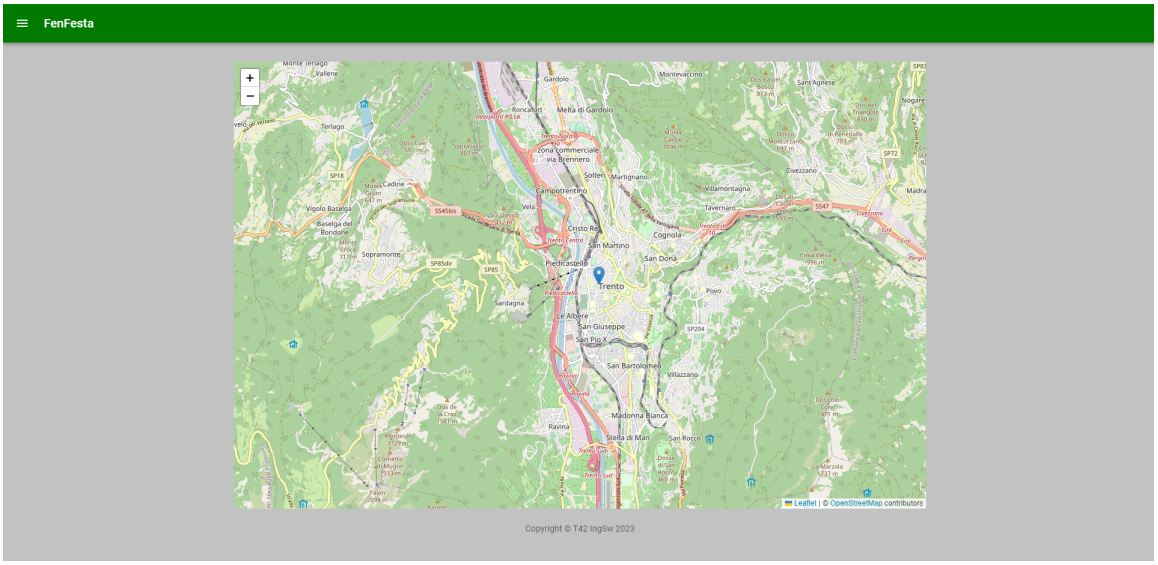
Il Font-End fornisce una visualizzazione grafica rispetto alle funzioni di:

- Creazione dell'Account
- Mappa Eventi
- Calendario Eventi
- Creazione di un Nuovo Evento
- Visualizzazione Evento

In particolare, tali funzioni sono suddivise nelle 5 schermate rinominate:

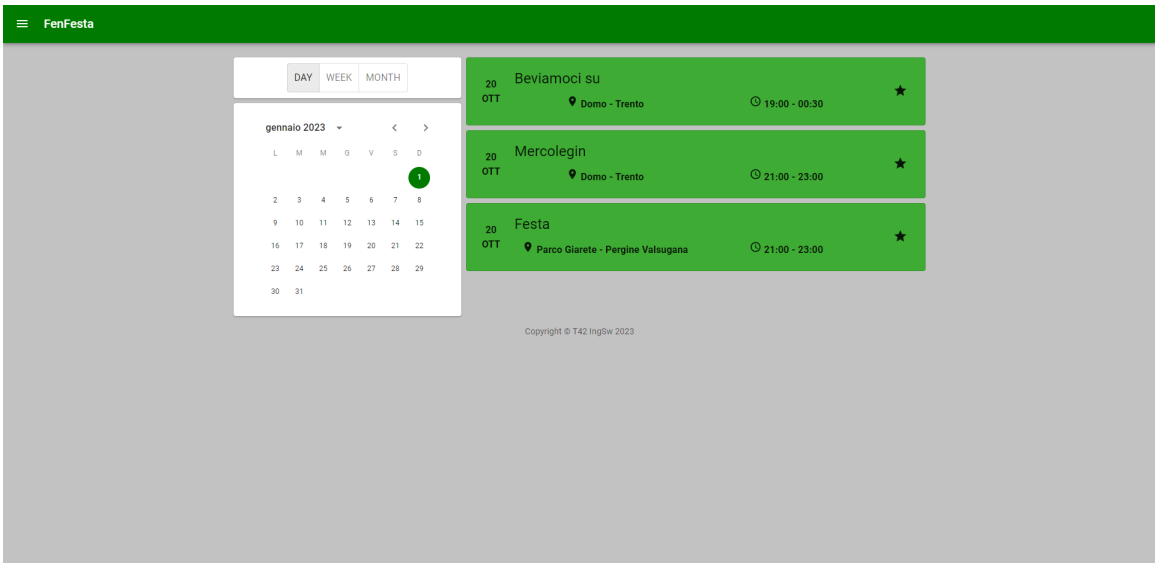
- Creazione Account
- Mappa
- Calendario
- Creazione
- Visualizza Evento

5.1 Mappa Eventi



Nella sezione Mappa si può visualizzare una cartina interagibile, inserita grazie alle API di OpenStreetMap, nella quale sono posizionati dei marker che determinano la presenza di un Evento in quel punto. Cliccandoci sopra si aprirà la sezione “Visualizza Evento”.

5.2 Elenco Eventi

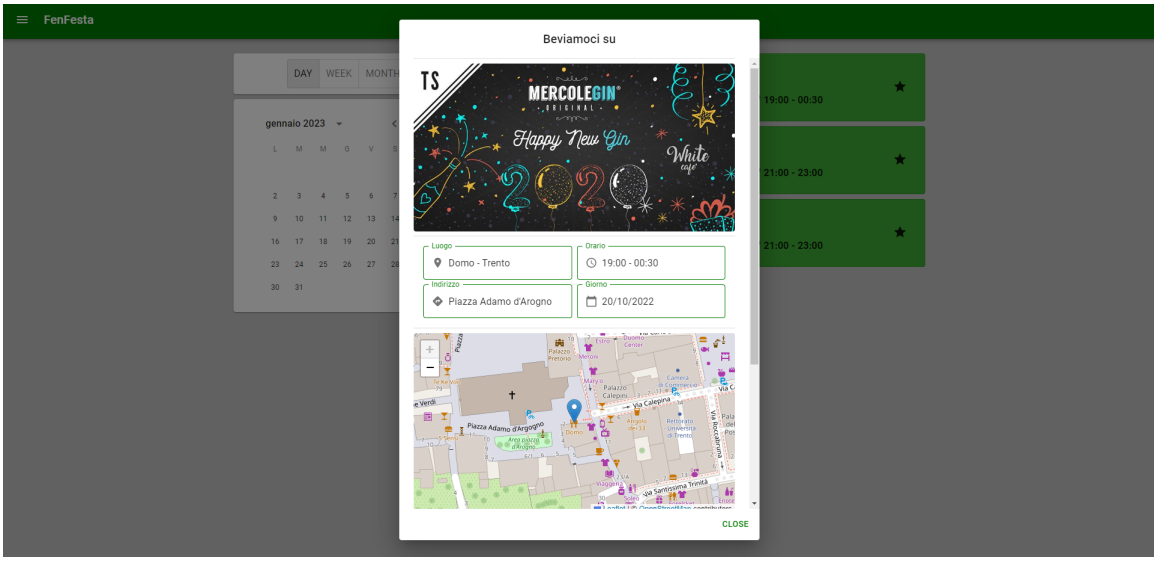


Nella schermata Calendario Eventi è disponibile un calendario con i giorni dell’anno ad accompagnare una lista di eventi disponibili giornalieri, settimanali o mensili

Come nella Mappa, anche in questa schermata selezionando l’evento a cui si è interessati si aprirà la schermata di visualizzazione evento.

Per selezionare l’Elenco o la Mappa è presente un menù laterale dove si può selezionare una delle due schermate

5.3 Visualizza Evento

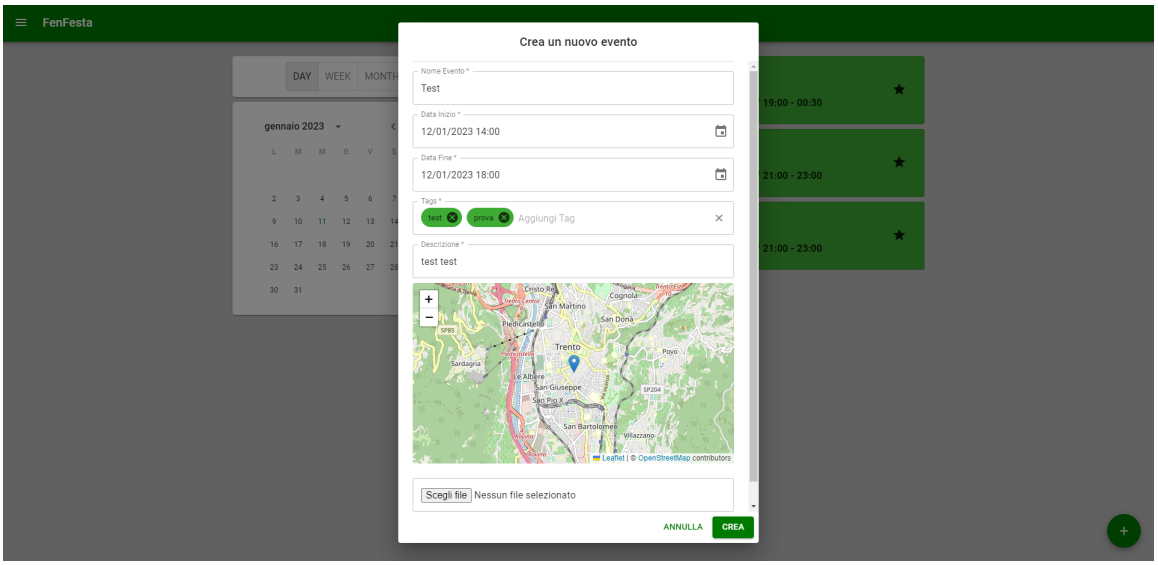


In questa sezione sono contenute tutte le informazioni relative all'evento (prese dal database). Sono presenti: immagine dell'evento, nome, data, orario, luogo, descrizione, tags, numero partecipanti, tasto "Partecipo", tasto "Salva Evento".

Tramite il primo tasto viene aggiunta una persona al counter dei partecipanti; con la pressione del secondo, invece, si viene notificati della prossimità dell'evento tramite mail, il giorno stesso dell'evento.

Questa schermata viene visualizzata sia quando si seleziona un evento dall'Elenco sia selezionando l'evento attraverso la Mappa

5.4 Creazione Evento



Mediante la pressione di un tasto con l'icona di un + in una delle due schermate: Elenco o Mappa Eventi, e solo se si ha un account organizzatore, viene aperta la schermata di creazione Evento. È presente un form per l'inserimento di tutte le informazioni necessarie per la creazione, quali: immagine dell'evento, nome, data e orario, luogo, descrizione, tags.

Premendo il tasto "Crea", l'evento verrà registrato nel database e inserito nell'elenco eventi visualizzato tramite Mappa o Calendario

6 GitHub Repository and Deployment Info

Il progetto FenFesta è disponibile al seguente link: `https://github.com/T42CaCaGhi-Project`

Per esegiure il front-end è necessario clonare la repository e successivamente utlizzare i comandi:

1. `npm install` per installare le varie dipendenze
2. `npm start` per far partire il front-end

Per esegiure il back-end è necessario clonare la repository e successivamente utlizzare i comandi:

1. `npm install` per installare le varie dipendenze
2. `npm start` per far partire il back-end

Per esegiure i test sulle API è necessario clonare la repository del back-end e successivamente utlizzare i comandi:

1. `npm install` per installare le varie dipendenze
2. `npm run test` per far partire i test sulle API

7 Testing

Sono stati effettuati dei test usando jest su tutte le API implementate coprendo tutte o quasi (ad eccezione di errori interni del sistema) le tipologie di risposta identificate a partire dalle figure 6, 7 e 8.

I test vengono eseguiti in modo sequenziale resettando e ripopolando dopo ogni test un database utilizzato esclusivamente per il testing.

Il comando con cui vengono eseguiti i test è '**jest –runInBand –coverage**', è stato generata quindi anche la cartella **coverage** contenente il file **index.html** che se aperto in un browser fornisce informazioni addizionali sul completamento dei test effettuati.

Tutti i test eseguiti sono riusciti. Nella seguente immagine è possibile vedere come il 100% delle funzioni viene coperto e quasi sempre la copertura delle righe di codice supera il 90% (principalmente vengono saltate porzioni di codice che gestiscono eventuali errori interni al sistema che non si sono verificati durante il testing).

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	93.33	91.42	100	93.1	
src	91.66	75	100	90.9	
app.ts	100	100	100	100	
db.ts	83.33	50	100	80	9
swaggerSetup.ts	100	100	100	100	
src/handlers	92.54	93.33	100	92.25	
base.ts	100	100	100	100	
eventHandle.ts	93.57	92.1	100	93.26	54,195,265,350,482,493,580
userHandle.ts	89.58	95.45	100	89.36	15,98,221,350-353
src/middleware	92.85	83.33	100	92.85	
token.ts	92.85	83.33	100	92.85	31
src/router	100	100	100	100	
eventRouter.ts	100	100	100	100	
mainRouter.ts	100	100	100	100	
userRouter.ts	100	100	100	100	
src/schemas	100	100	100	100	
evento.ts	100	100	100	100	
schemaIndex.ts	0	0	0	0	
user.ts	100	100	100	100	

Di seguito proponiamo un esempio dei test effettuati. La struttura dei test effettuati è la stessa per ogni test: i vari test sono raggruppati per API (salvo una singola eccezione dove era necessario un ambiente differente, per questo il test è stato svolto in maniera isolata) e divisi in due file, uno contenente i test per le API con percorso **/api/user** e uno contenentei test per le API con percorso **/api/event**.

```
265  //
266  //  GET EVENT
267  //
268  describe("Single Event print", () => {
269    test(`POST localhost:${port}/api/event/ should return single event corresponding to id`, async () => {
270      const input = { _id: eventIds[0].toString() };
271      const output = {
272        data: {
273          __v: 0,
274          _id: eventIds[0].toString(),
275          dateFinish: "2022-12-31T14:00:00.000Z",
276          dateStart: "2022-12-31T12:00:00.000Z",
277          description: "string",
278          idOwner: {
279            _id: userIds[2].toString(),
280            alias: "foo",
281            img: "bar",
282          },
283          image: "string",
284          location: {
285            _id: locationIds[0].toString(),
286            city: "string",
287            lat: 0,
288            lon: 0,
289            name: "string",
290            street: "string",
291          },
292          nParticipants: 0,
293          tags: ["tag1", "tag2"],
294          title: "string",
295        },
296        status: "success",
297      };
298      const res = await nodeFetch(`http://localhost:${port}/api/event/`, {
299        method: "POST",
300        headers: { "Content-Type": "application/json" },
301        body: JSON.stringify(input),
302      });
303      expect(res.status).toBe(200);
304      const data = await res.json();
305      expect(data).toEqual(output);
306    });
307    test(`POST localhost:${port}/api/event/ should return error if no event is found, but _id is a valid format`, async () => {
308      const input = { _id: eventIds[4].toString() };
309      const output = {
310        status: "error",
311        message: "Not found",
312      };
313      const res = await nodeFetch(`http://localhost:${port}/api/event/`, {
314        method: "POST",
315        headers: { "Content-Type": "application/json" },
316        body: JSON.stringify(input),
317      });
318      expect(res.status).toBe(404);
319      const data = await res.json();
320      expect(data).toEqual(output);
321    });
322    test(`POST localhost:${port}/api/event/ should return error if _id is not a valid format`, async () => {
323      const input = { _id: "wrong" };
324      const output = {
325        status: "error",
326        message:
327          'Cast to ObjectId failed for value "wrong" (type string) at path "_id" for model "Evento"',
328      };
329      const res = await nodeFetch(`http://localhost:${port}/api/event/`, {
330        method: "POST",
331        headers: { "Content-Type": "application/json" },
332        body: JSON.stringify(input),
333      });
334      expect(res.status).toBe(500);
335      const data = await res.json();
336      expect(data).toEqual(output);
337    });
338  });
```

7.1 Testing API User

7.1.1 POST /api/user/new

Ha successo se sia la password che la mail vengono fornite e se la mail è unica.

Fallisce se uno dei due dati non viene fornito.

Fallisce se la mail è già in uso.

7.1.2 POST /api/user/login

Ha successo se la mail e la password fornite identificano un utente creato in precedenza.

Fallisce se le credenziali sono incomplete.

Fallisce se la password non è corretta.

Fallisce se la mail non identifica un utente precedentemente creato.

7.1.3 POST /api/user/delete

Ha successo se la mail e la password fornite identificano un utente creato in precedenza.

Fallisce se il token non è corretto,

Ha successo se chi richiede l'eliminazione di un utente che esiste è un admin.

Fallisce se l'email non è corretta.

7.1.4 POST /api/event/all

Ha successo se esistono eventi nel database.

Fallisce se non ci sono eventi nel database.

7.1.5 POST /api/event/

Ha successo se l'ID inserito è riferito ad un evento che esiste nel DB.

Fallisce in caso contrario.

Fallisce se l'ID fornito non è nel formato corretto di un ID di MongoDB.

7.1.6 POST /api/event/new

Ha successo se il request body è corretto e lo user è autorizzato.

Fallisce se il request body è corretto ma lo user non è autorizzato.

Fallisce se il request body è corretto, lo user è autorizzato, ma esiste già un evento con lo stesso titolo e la stessa data di inizio

7.1.7 POST /api/event/modify

Ha successo se se stiamo modificanto un evento che esiste nel DB e se lo user è autorizzato (organizzatore).

Ha successo se se stiamo modificanto un evento che esiste nel DB e se lo user è autorizzato (admin).

Fallisce se lo user non è autorizzato.

Fallisce se si sta cercando di modificare un evento che non esiste.

7.1.8 POST /api/event/delete

Ha successo se l'evento esiste e lo user è autorizzato (orgganizzatore).

Ha successo se l'evento esiste e lo user è autorizzato (admin).

Fallisce se lo user non è autorizzato.

Fallisce se l'evento non esiste.

7.1.9 POST /api/event/tag

Ha successo se almeno un evento contiene il/i tag presenti nel request body.

Fallisce se nessun evento contiene i tag desiderati.

7.1.10 POST /api/event/period

Ha successo se esiste almeno un evento nel periodo scelto (sia esso giorno, settimana o mese).

Fallisce se non viene trovato alcun evento.

7.1.11 POST /api/event/preferito

Ha successo se l'evento esiste e lo user si identifica correttamente con il suo token (aggiunta dell'evento ai preferiti).

Ha successo se l'evento esiste e lo user si identifica correttamente con il suo token (rimozione dell'evento dai preferiti).

Fallisce se il token non è valido.

Fallisce se l'evento non esiste.