



# TN2night

22/12/2022

D4-T02

# Indice

<b>Indice</b>	<b>2</b>
<b>Scopo del Documento</b>	<b>4</b>
<b>User Flow</b>	<b>5</b>
<b>Application implementation and documentation</b>	<b>6</b>
Project Structure	6
API del progetto	7
API del Front-End	7
Project Dependencies	7
Project Data or DB	8
Tipo di dato user:	9
Tipo di dato evento:	9
Relazione tra utenti denominata friends:	9
Tipo di dato spots:	9
Relazione tra utente ed evento denominata user_events:	9
Relazione tra utente e spots denominata user_like:	10
<b>Project API</b>	<b>10</b>
API Diagram	10
Resource Model	11
Sviluppo API	13
Autenticazione	13
Registrazione di un utente	14
Logout utente	15
Visualizza tutti gli eventi	16
Creazione di uno spot	16
Lista di tutti gli spot	18
Aggiunta di un utente alla lista amici	19
Lista amici	19
Utente segue un evento	21
Like ad uno spot	22
<b>API documentation</b>	<b>23</b>
<b>Front-End Implementation</b>	<b>24</b>
<b>GitHub Repository &amp; Deployment Guide</b>	<b>24</b>

<b>API Testing</b>	<b>24</b>
Operazioni testate	25

## Scopo del Documento

Il presente documento riporta tutte le informazioni necessarie per lo sviluppo di una parte dell'applicazione TN2night. In particolare, presenta tutti gli artefatti necessari per gestire gli utenti registrati all'applicazione, gestire la creazione e la visualizzazione degli spot, la visualizzazione degli eventi, la visualizzazione di altri utenti e la gestione degli amici.

Partendo dalla descrizione degli user flow legati alle possibili azioni che un utente può intraprendere sull'applicazione, il documento prosegue con la presentazione delle API necessarie (tramite l'API Model e il Modello delle risorse) per poter eseguire quanto descritto sopra.

Per ogni API realizzata, oltre ad una descrizione delle funzionalità fornite, il documento presenta la sua documentazione e i test effettuati.

Infine una sezione è dedicata alle informazioni del Git Repository e il deployment dell'applicazione stessa.

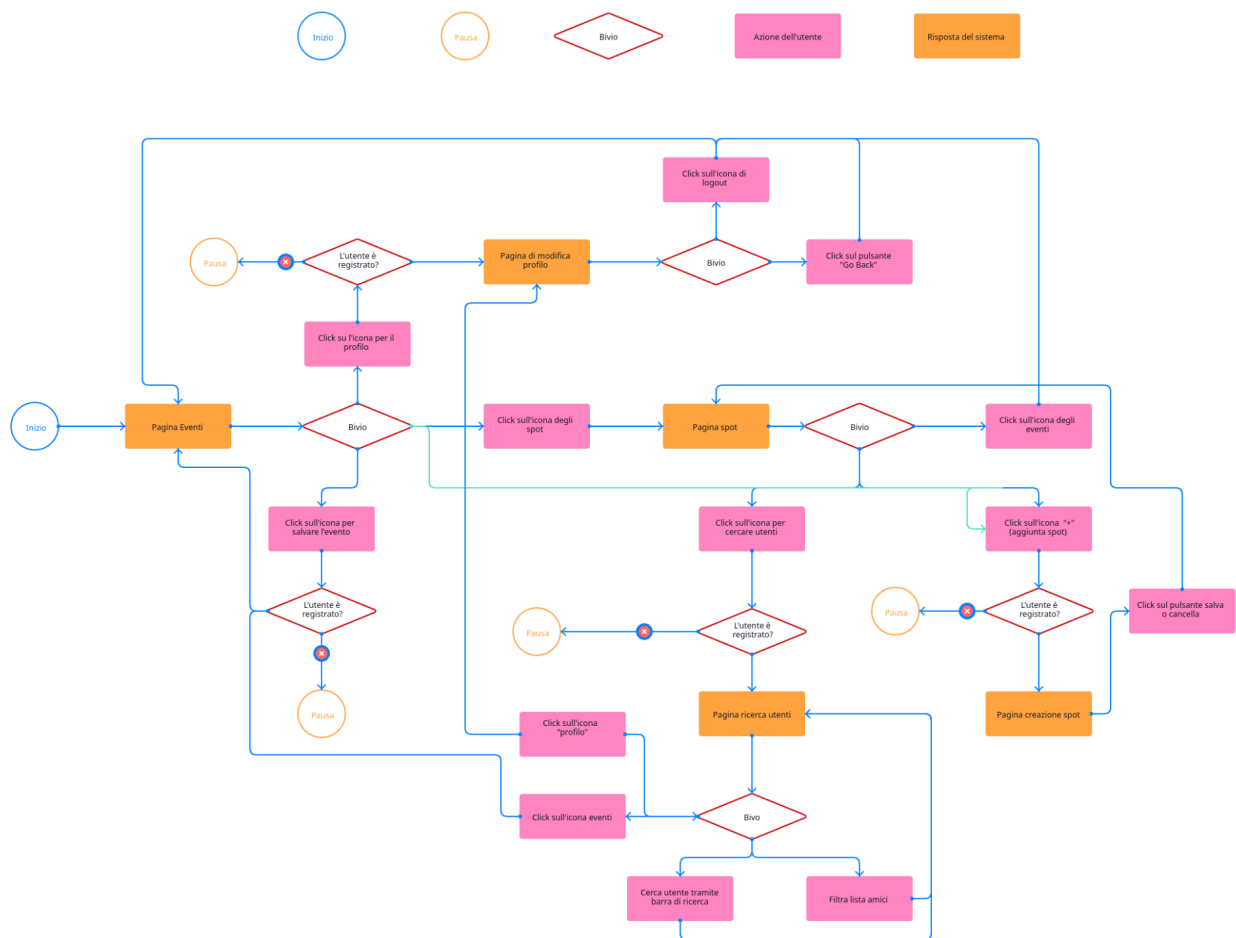
# User Flow

Il nostro user flow descrive le possibili azioni che un utente può intraprendere nell'applicazione, mostrandone graficamente il percorso. Il tutto parte dalla pagina degli eventi in quanto l'applicazione aprendosi mostra questa sia ad utenti registrati che non registrati.

L'utente da questa pagina può svolgere azioni sugli eventi, visitare il proprio profilo o la pagina degli spot. Per eseguire le prime due azioni è necessario che l'utente sia registrato e loggato sull'applicazione.

Infine l'utente può eseguire diverse azioni su queste pagine, ad esempio modificare il proprio profilo, fare il logout dall'applicazione o creare un nuovo spot.

In figura 2 presentiamo tutte le relazioni tra le varie azioni che l'utente può fare e alcune features descritte in Sezione 2. Una legenda che descrive i simboli usati nello user flow è anche presentata in Figura 2.



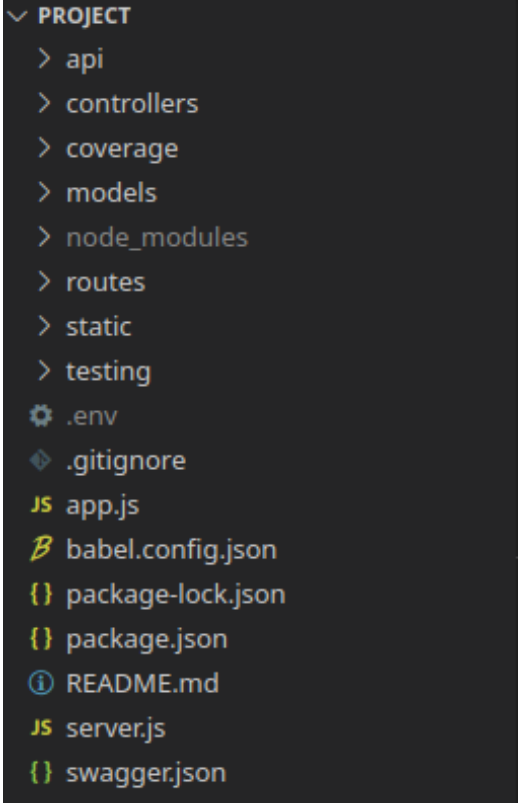
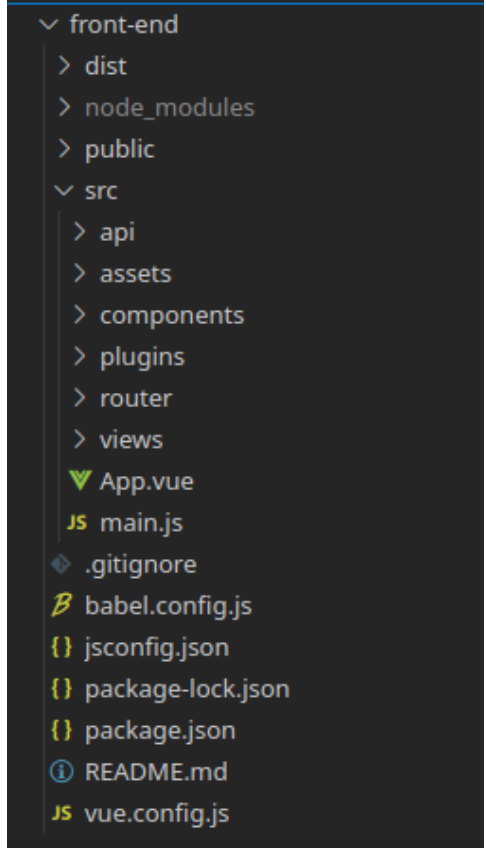
# Application implementation and documentation

Nelle sezioni precedenti abbiamo identificato le varie features che devono essere implementate per la nostra applicazione con un'idea di come il nostro utente finale puo' utilizzarle nel suo flusso applicativo. L'applicazione è stata sviluppata utilizzando NodeJS e VueJS. Per la gestione dei dati abbiamo utilizzato MongoDB.

## Project Structure

Il progetto è stato diviso in due moduli e ognuno di questi è organizzato in diverse cartelle.

- **Back-end:** Il tutto parte dal file **server.js** che si occupa di chiamare le rotte delle api invocate dal front-end. Queste rotte sono salvate in **routes/routes.js**. Da queste rotte poi vengono invocate le funzioni necessarie a soddisfare le richieste. Queste funzioni si trovano in diversi file nella cartella controllers. I modelli usati per rappresentare i diversi dati si trovano in diversi file nella cartella **models**. Infine i file per eseguire i test sono nella cartella **testing**.
- **Front-end:** Nelle seguenti figure uno screenshot delle due strutture.

API del progetto	API del Front-End
	

## Project Dependencies

I seguenti moduli di node sono stati utilizzati e aggiunti al file **package.json**

```
"dependencies": {
  "@types/node-fetch": "^2.6.2",
  "body-parser": "^1.20.1",
  "dotenv": "^16.0.3",
  "express": "^4.18.2",
  "firebase": "^9.15.0",
  "jsonwebtoken": "^8.5.1",
  "moment": "^2.29.4",
  "mongoose": "^6.8.1",
  "mongoose-express-api": "^0.0.3",
  "multer": "^1.4.5-lts.1",
  "swagger-ui-express": "^4.6.0"
},
"devDependencies": {
  "@babel/preset-env": "^7.20.2",
  "babel-jest": "^29.3.1",
  "node-fetch": "^2.6.7",
  "supertest": "^6.3.3"
},
```

## Project Data or DB

Per la gestione dei dati utili all'applicazione abbiamo definito alcune strutture dati:

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
<a href="#">eventos</a>	57	5.47KB	99B	36KB	1	36KB	36KB
<a href="#">friends</a>	1	86B	86B	24KB	1	24KB	24KB
<a href="#">spots</a>	23	2.5KB	112B	36KB	1	36KB	36KB
<a href="#">user_events</a>	2	182B	91B	36KB	1	36KB	36KB
<a href="#">user_likes</a>	1	92B	92B	24KB	1	24KB	24KB
<a href="#">users</a>	14	2.02KB	148B	36KB	1	36KB	36KB

In particolare possiamo notare le strutture **user\_events** e **user\_likes** che permettono di tenere traccia di quali utenti hanno deciso di seguire un determinato evento e quali utenti hanno deciso di mettere like ad uno spot.

Si riportano degli esempi di dati presenti nel database.

### Tipo di dato **user**:

```
_id: ObjectId('63976490a3b89ee206d44d45')
email: "francescodolini@gmail.com"
password: "verde32"
username: "Francesco312"
contatto: "telegram.com"
status: true
__v: 0
```

### Tipo di dato **evento**:

```
_id: ObjectId('63a2ee353717bf4e9724d77d')
nome: "Festa di compleanno"
data: 2022-08-01T17:30:00.000+00:00
luogo: "aKka Trento"
__v: 0
```

Relazione tra utenti denominata **friends**:

```
_id: ObjectId('63a6b0f9f6d7c5aa67666903')  
username: "Beencal4"  
friend_username: "Francesco1"  
__v: 0
```

Tipo di dato **spots**:

```
_id: ObjectId('63a32bf5a04114ce4f5523f6')  
testo: "Spotto Alessandra in mensa a povo 0"  
autore: "Francesco1"  
num_like: 0  
__v: 0
```

Relazione tra utente ed evento denominata **user\_events**:

```
_id: ObjectId('639da24a2ed5b7e449a20326')  
username: "Francesco1"  
id_evento: "6394a0a69223d52bae8336ee"  
__v: 0
```

Relazione tra utente e spots denominata **user\_like**:

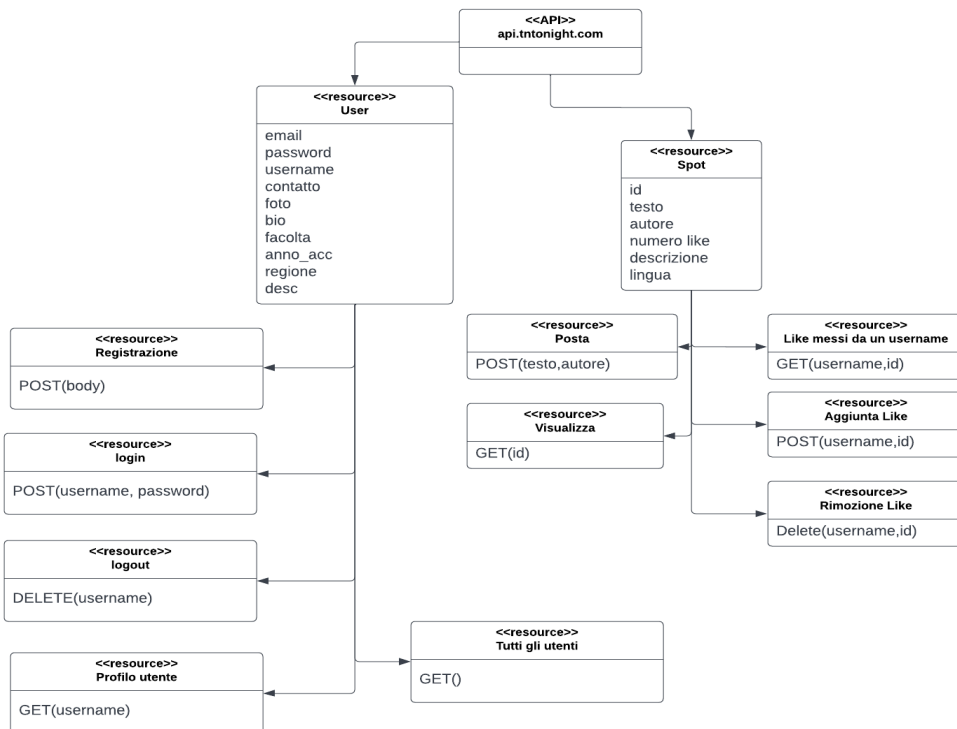
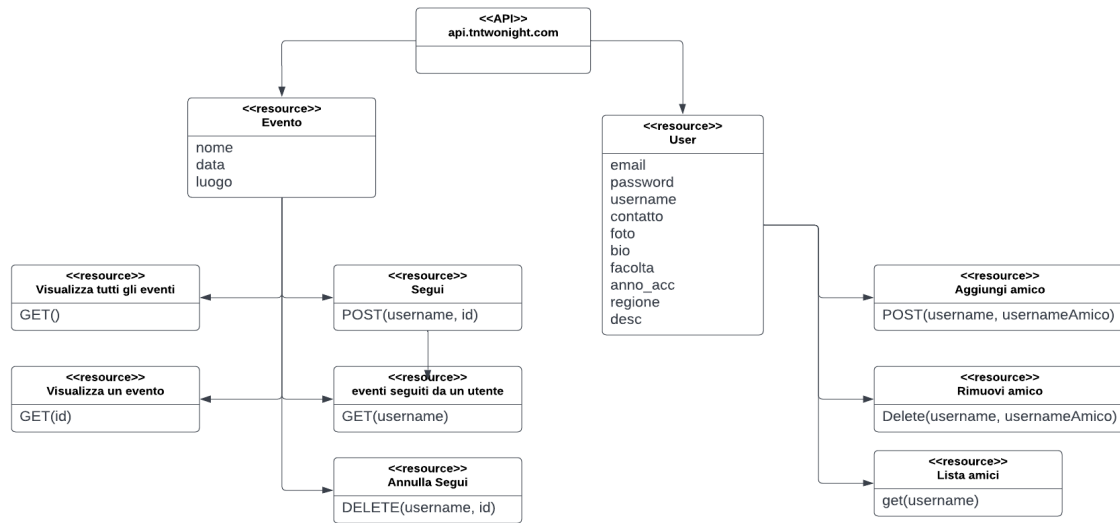
```
_id: ObjectId('63a6b08af6d7c5aa676668fd')  
username: "Beencal4"  
id_spot: "63a329cfa04114ce4f5523df"  
__v: 0
```

## Project API

### API Diagram

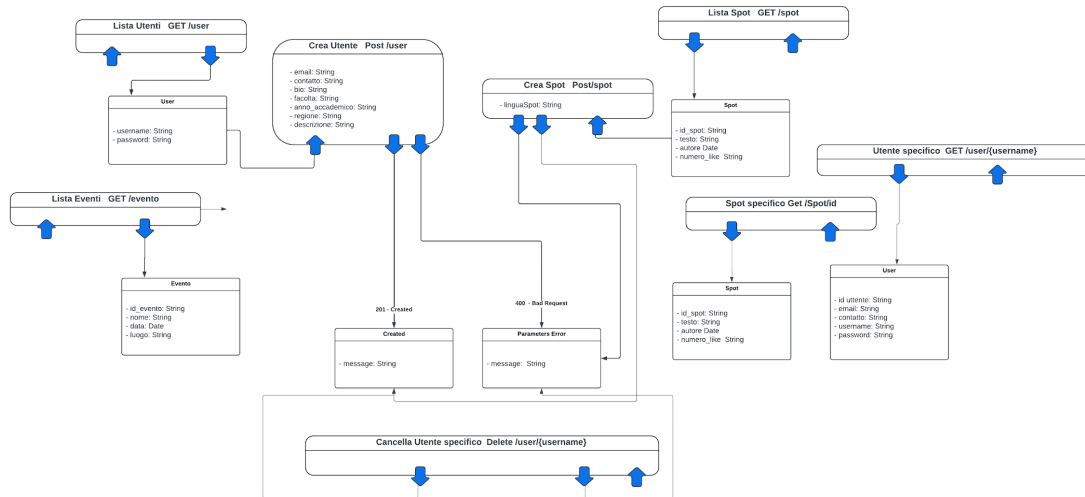
Nella seguente sezione riportiamo il diagramma delle api che siamo andati effettivamente a sviluppare.



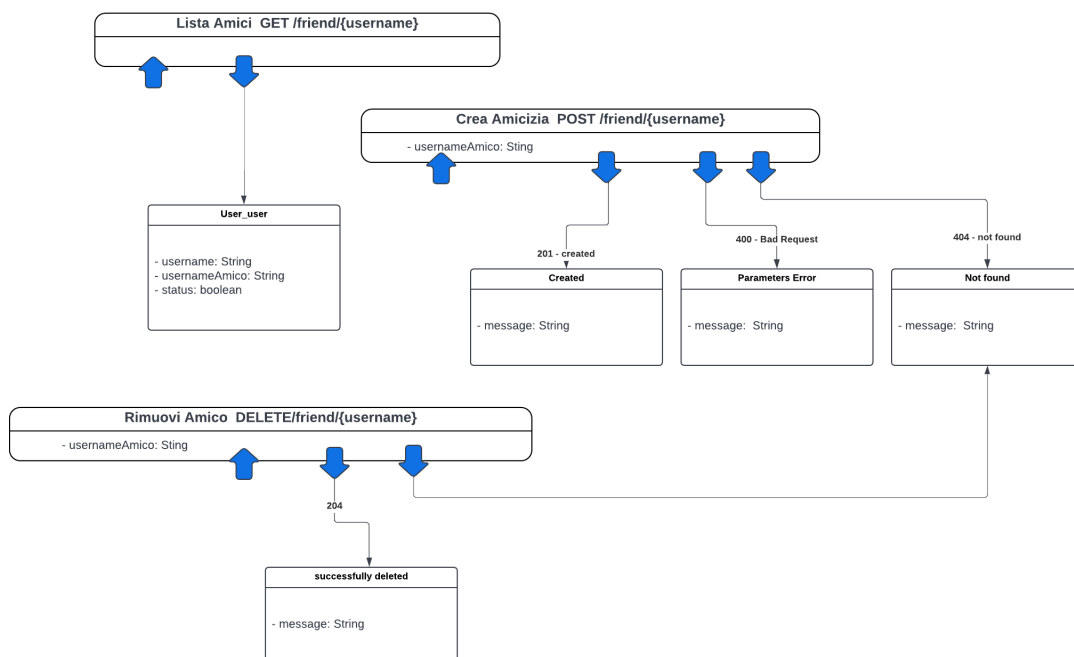


## Resource Model

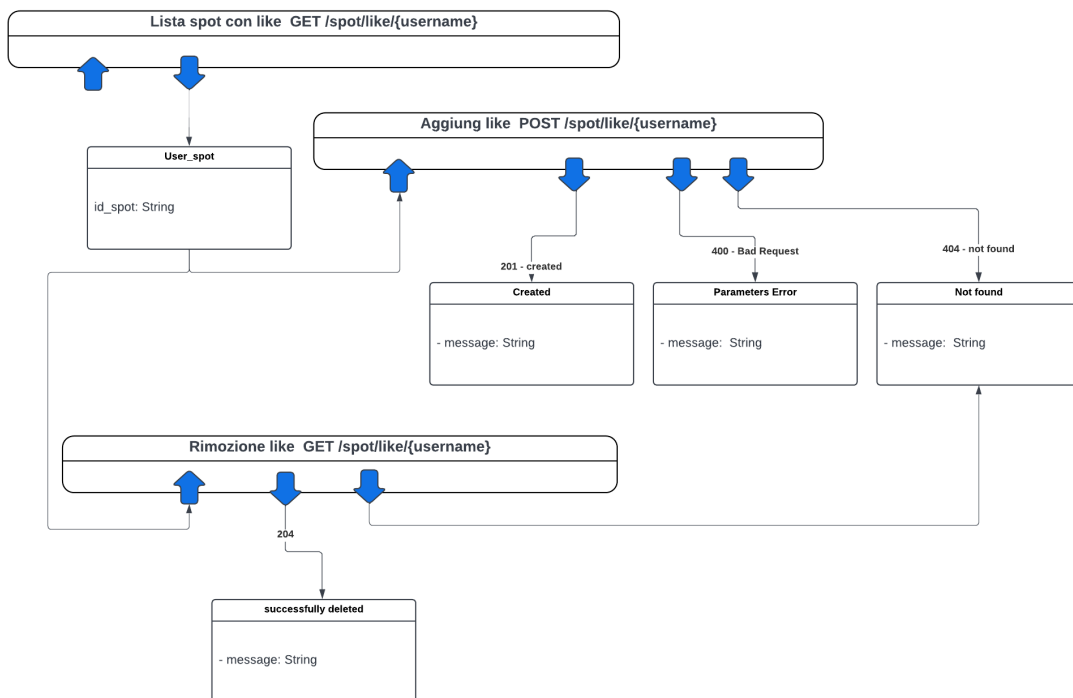
In questa sezione è presente il resource model dell'api dell'applicazione  
**Resource model riguardante: Utente, Eventi, Spot.**



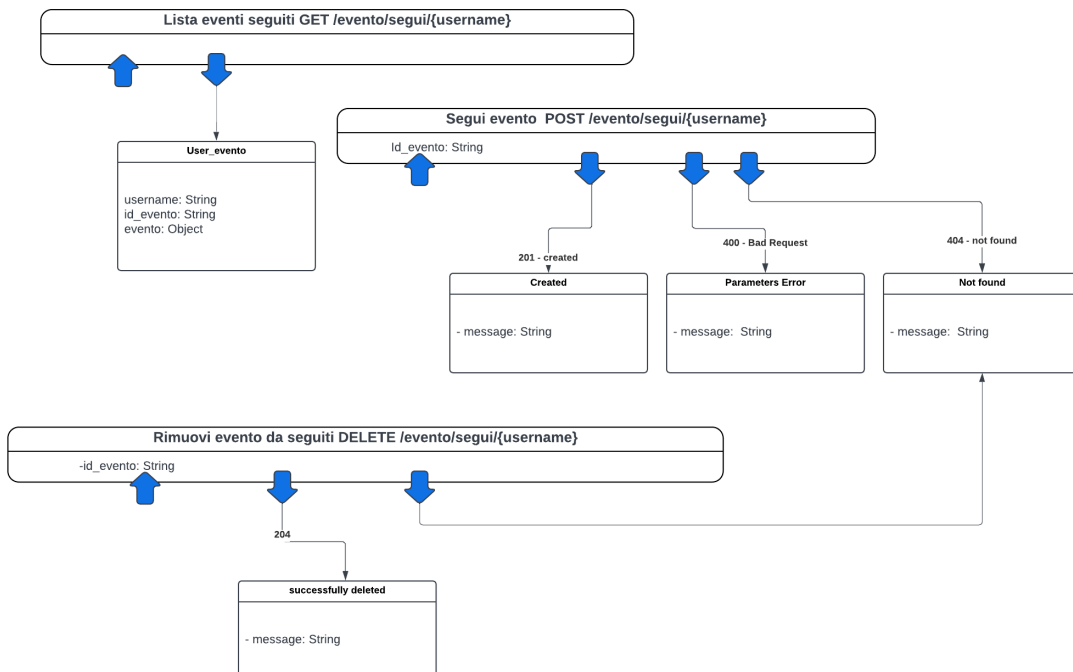
**Resource model riguardante: Amici** cioè le relazioni che si possono formare tra utenti.



**Resource model riguardante: Spot\_like** gestendo i like che gli utenti mettono ad uno spot



**Resource model riguardante: User\_event** cioè gli gestisce le modalità con cui un utente segue un evento



## Sviluppo API

Di seguito viene mostrato il codice di alcune delle api più utilizzate dal progetto. I dettagli sui parametri richiesti da una api e i dati che restituisce sono specificati in seguito nel punto 4 con la documentazione swagger. Bisogna tenere in mente che alcuni aspetti di sicurezza sono stati tralasciati, ad esempio attacchi man-in-the-middle o simili non vengono considerati. L'unico livello di sicurezza che viene implementato è l'autenticazione tramite token e la verifica che non vengano inseriti dati doppi o che vengano richiesti dati non esistenti. Il token viene utilizzato per riconoscere utenti registrati e per permettergli di accedere ad alcune risorse a cui gli utenti non registrati non possono arrivare.

### Autenticazione

Questa api viene usata quando un utente si deve autenticare. Vengono inviati username e password, inviati al sistema attraverso una pagina di login, e ne viene controllata l'esistenza e la correttezza nel database. Se i dati sono corretti viene inviato un token con cui autenticarsi ad ogni richiesta. Se invece sono errati viene inviato un messaggio di errore.

```
const auth = async function(req, res) {
  let user = await User.findOne({username: req.body.username}).exec();
  if(!user) {
    return res.json({success: false, message: 'Authentication failed. User not found'});
  }

  if (user.password !== req.body.password) {
    return res.json({ success: false, message: 'Authentication failed. Wrong password.' });
  }

  // if user is found and password is right create a token
  var payload = {
    email: user.email,
    username: user.username
    // other data encrypted in the token
  }
  var options = {
    expiresIn: 86400 // expires in 24 hours
  }
  var token = jwt.sign(payload, process.env.SUPER_SECRET, options);

  res.status(200).json({
    success: true,
    message: 'Enjoy your token!',
    token: token,
    email: user.email,
    username: user.username,
    self: "user/" + user._id
  });
}
```

## Registrazione di un utente

In questa api si vede il meccanismo di registrazione di un nuovo utente. Innanzitutto viene verificato che il nome utente non sia già presente. Nel caso non sia già presente viene creato un nuovo utente con i dati ricevuti dal front-end. Altrimenti si invia un messaggio di errore. Al momento della registrazione l'utente deve obbligatoriamente specificare solo alcuni campi mentre altri sono facoltativi, come da requisiti.

```
//POST user
const newUser = (req, res) => {
  //check if the user email already exists in db
  User.findOne({ username: req.body.username }, (err, data) => {
    //if user not in db, add it
    if (!data) {
      //create a new user object using the User model and req.body
      const newUser = new User({
        email: req.body.email,
        password: req.body.password,
        username: req.body.username,
        contatto: req.body.contatto,
        foto: req.body.foto,
        bio: req.body.bio,
        facolta: req.body.facolta,
        anno_acc: req.body.anno_acc,
        regione: req.body.regione,
        desc: req.body.desc,

        status: true,
      })
      // save this object to database
      newUser.save((err, data) => {
        if (err) return res.json({ Error: err });
        return res.status(201).json({ message: "Utente registrato con successo" });
      })
    } else {
      //if there's an error or the user is in db, return a message
      if (err) return res.status(400).json('Something went wrong, please try again. ${err}');
      return res.status(400).json({ message: "User already exists" });
    }
  })
};
```

## Logout utente

Questa api viene utilizzata per fare il logout dell'utente. Per segnare il logout nel database viene aggiornata a false la flag dello status (questa flag viene principalmente usata per far sapere ad altri utenti se questo specifico utente è online oppure no).

```
const logoutUser = async (req, res) => {  
  const outUser = await User.findOne({username: req.params.username})  
  if(!outUser) {  
    res.status(404).json({res: 'User not found'}).send()  
    return;  
  }  
  await outUser.updateOne({status: false})  
  res.status(204).json({  
    res: "Logout effettuato"  
  })  
  return;  
}
```

## Visualizza tutti gli eventi

Questa api restituisce tutti gli eventi presenti nel database. La data viene formattata per seguire il formato italiano. Nel front-end inoltre si possono vedere le persone che seguono un certo evento, questa informazione viene fornita da un altro modulo.

```
const getAllEvents = async (req, res) => {  
  let events = await Evento.find({});  
  events = events.map( (event) => {  
    var data = moment(event.data).format('DD/MM/YYYY HH:mm')  
    return {  
      id_evento: event._id,  
      nome: event.nome,  
      data: data,  
      luogo: event.luogo  
    };  
  });  
  res.status(200).json(events);  
};
```

## Creazione di uno spot

Attraverso questa api si può creare un nuovo spot, sia generale che specifico. Nel caso di uno spot generico la descrizione non viene specificata e quindi nel database viene settata a 'none'. A fine processo viene restituito l'esito del salvataggio dello spot sul database.

```
const newSpot = async (req, res) => {  
  
  const exists = await User.findOne({  
    username: req.body.autore,  
  });  
  
  if(exists) {  
    let desc  
    let lang  
    if(req.body.desc == undefined)  
      desc = 'none'  
    if(req.body.lang == undefined)  
      lang = 'Italiano'  
    const newSpot = new Spot({  
      testo: req.body.testo,  
      autore: req.body.autore,  
      num_like: 0,  
      lang: req.body.lang,  
      desc: req.body.desc  
    })  
    // save this object to database  
    newSpot.save((err, data)=>{  
      if(err) return res.status(400).json({Error: err});  
      return res.status(201).json({message: "Spot creato con successo"});  
    })  
  }  
  else  
    return res.status(400).json({message: "Impossibile creare lo spot"})  
};
```



## Lista di tutti gli spot

Questa api restituisce la lista di tutti gli spot presenti nel database. Viene anche restituito l'id di ogni spot, creato da mondoDB, per poi poter lavorare più facilmente con questi nel front-end. Gli spot hanno anche un numero di like, questo dato viene modificato da un altro modulo che si occupa della relazione tra utente e mi piace messo ad un certo spot.

```
const getAllSpots = async (req, res) => {  
  let spots = await Spot.find({});  
  spots = spots.map( (spot) => {  
    return {  
      id_spot: spot._id.toString(),  
      testo: spot.testo,  
      autore: spot.autore,  
      num_like: spot.num_like,  
      desc: spot.desc,  
      lang: spot.lang,  
    };  
  });  
  res.status(200).json(spots);  
};
```

## Aggiunta di un utente alla lista amici

Questa api permette di aggiungere un utente alla lista amici. Nel database questa relazione viene rappresentata con l'username di entrambi gli utenti. Innanzitutto viene controllato che l'utente esista nel database, poi viene controllato che la relazione non sia già presente, se entrambe le condizioni sono soddisfatte viene creata una nuova entry.

```
const addFriend = async (req, res) => {
  const exists = await User.findOne({
    username: req.body.friend_username,
  });

  const existsF = await Friend.findOne({
    username: req.params.username,
    friend_username: req.body.friend_username,
  })

  if(exists && !existsF) {
    const newFriend = new Friend ({
      username: req.params.username,
      friend_username: req.body.friend_username,
    })
    newFriend.save((err, data)=>{
      if(err) return res.status(400).json({Error: err});
      return res.status(201).json(data);
    })
  }
  else if(!exists)
    return res.status(404).json({res: 'User does not exist'}).send()
  else
    return res.status(400).json({res: 'Friend already added'}).send()
};
```

## Lista amici

Questa api viene utilizzata nella frazione di ricerca utenti per mostrare tra i risultati solo quelli che sono stati aggiunti tra gli amici. L'api restituisce tutti gli amici che poi verranno filtrati nel front-end. L'api va a cercare nella tabella corrispondente la lista degli amici e prima di restituirla va a vedere per ogni utente il suo status (online o offline) e lo aggiunge alla risposta.

```
const getFriends = async (req, res) => {
  let friends;
  if(req.params.username) {
    friends = await Friend.find({
      username: req.params.username
    }).exec();
  }
  else
    friends = await Friend.find({}).exec();
  friends = friends.map((dbEntry) => {
    return {
      username: dbEntry.username,
      friend_username: dbEntry.friend_username,
    };
  });
  for(let i = 0; i < friends.length; i++) {
    let tmp = await User.findOne({username: friends[i].friend_username})
    friends[i].status = tmp.status
  }
  res.status(200).json(friends);
};
```

## Utente segue un evento

Questa api viene utilizzata quando un utente segue un evento, questa relazione viene salvata nel database in una tabella separata utilizzando il modello `User_Event`. Innanzitutto viene controllata la correttezza dell'username dell'utente e dell'id dello evento, infine, se questa già non esiste, viene aggiunta la relazione nel database.

```
const addUE = async (req, res) => {  
  const existsU = await User.findOne({  
    username: req.params.username,  
  });  
  
  var existsE;  
  if(req.body.id.length == 24) {  
    existsE = await Evento.findOne({  
      _id: req.body.id,  
    });  
  }  
  else  
    existsE = false;  
  
  const existsUE = await User_Event.findOne({  
    username: req.params.username,  
    id_evento: req.body.id  
  })  
  
  if(!existsUE) {  
    if(existsU && existsE) {  
      const newUE = new User_Event ({  
        username: req.params.username,  
        id_evento: req.body.id,  
      })  
      newUE.save((err, data)=>{  
        if(err) return res.status(400).json({Error: err});  
        return res.status(201).json(data);  
      })  
    }  
    else  
      res.status(404).json({res: 'User or event does not exist'}).send()  
  }  
  else  
    res.status(400).json({res: 'You already follow this event'}).send()  
};
```

## Like ad uno spot


Questa api viene utilizzata quando un utente mette like ad uno spot, viene utilizzato il modello User\_Like per tenere traccia dei diversi like messi agli spot messi dai diversi utenti. Innanzitutto viene controllata la correttezza dell'username dell'utente e dell'id dello spot, infine, se l'utente non aveva messo like precedentemente allo spot viene aggiunta la relazione nel database. Prima di concludere l'operazione però viene aumentato di uno il conto dei like nell'entità spot. Questa informazione viene usata nel front-end.

```
const addLike = async (req, res) => {
  const existsU = await User.findOne({
    username: req.params.username,
  });
  const existsS = await Spot.findOne({
    _id: req.body.id,
  });
  const existsL = await User_Like.findOne({
    username: req.params.username,
    id_spot: req.body.id,
  });

  if(!existsL) {
    if(existsU && existsS) {
      const newLike = new User_Like ({
        username: req.params.username,
        id_spot: req.body.id,
      })
      var numLikes = existsS.num_like + 1;
      await existsS.updateOne({num_like: numLikes});
      newLike.save((err, data)=>{
        if(err) return res.status(400).json({Error: err});
        return res.status(201).json(data);
      })
    }
    else
      return res.status(404).json({res: 'User or spot does not exist'}).send()
  }
  else
    return res.status(400).json({res: 'Spot already liked'}).send()
};
```

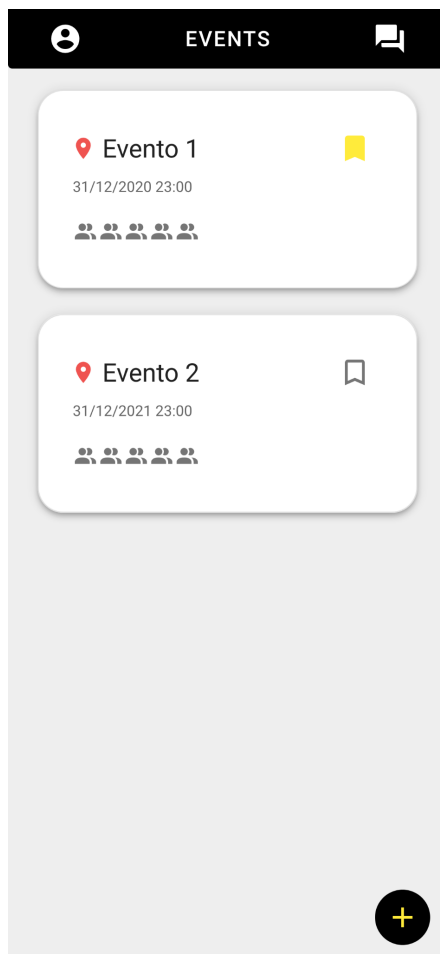
# API documentation

Le API Locali fornite dall'applicazione e descritte nella sezione precedente sono state documentate utilizzando il modulo NodeJS chiamato Swagger UI Express. In questo modo la documentazione relativa alle API è direttamente disponibile a chiunque veda il codice sorgente.

<b>Users</b> API for users in the system		^
<b>Authenticaton</b>		^
POST	/user/auth Authenticate a user	▼
<b>User</b>		^
POST	/user User registration	▼
GET	/user Get all users	▼
GET	/user/search Get the seached user	▼
GET	/user/{username} Get the logged user profile	▼ 
DELETE	/user/{username} Logout the user	▼
POST	/friend/{username} Adda a user as a friend	▼
DELETE	/friend/{username} Delete user from friends	▼
<b>Evento</b>		^
GET	/evento Get all events on the database	▼
<b>Friend</b>		^
POST	/friend/{username} Adda a user as a friend	▼
DELETE	/friend/{username} Delete user from friends	▼
GET	/friend/{username} Get all friends of the user	▼
<b>Spot</b>		^
POST	/spot Create a new spot, generic or specific	▼
GET	/spot Get all the spots on the database	▼
GET	/spot/id Get on spot specifying its id	▼
<b>User_Like</b>		^
GET	/spot/like/{username} Get all the spots liked by the user	▼
POST	/spot/like/{username} User adds a like to a specific spot	▼
DELETE	/spot/like/{username} User removes a like to a specific spot	▼
<b>User_Event</b>		^
POST	/evento/seguì/{username} User follow a specific event	▼
GET	/evento/seguì/{username} Get events followed by user	▼
DELETE	/evento/seguì/{username} Users unfollows a specific event	▼

## Front-End Implementation

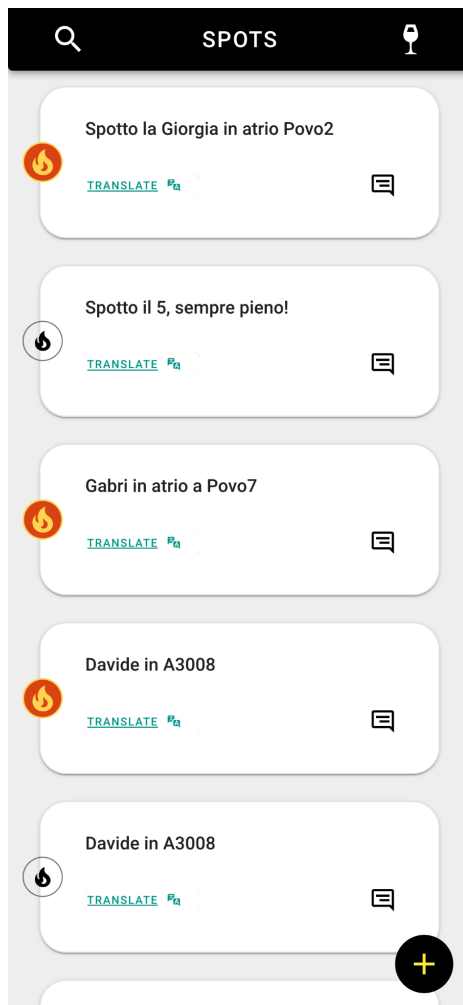
Il FrontEnd fornisce le funzionalità di visualizzazione, inserimento e cancellazione dei dati dell'applicazione. In particolare, l'applicazione è composta da una pagina eventi, una pagina spot, una pagina per gestire il proprio profilo ed una pagina per visualizzare gli altri utenti con i rispettivi amici. In tutte le schermate dell'applicazione la NavBar è un componente dinamico che permette di accedere a diverse pagine basandosi su che schermata è mostrata.



Sia un utente che ha effettuato l'accesso che un utente *anonimo* verrà accolto con questa schermata dove sono mostrati i vari eventi.

Dalla barra di navigazione è possibile andare nella schermata del proprio profilo o vedere gli spot postati.

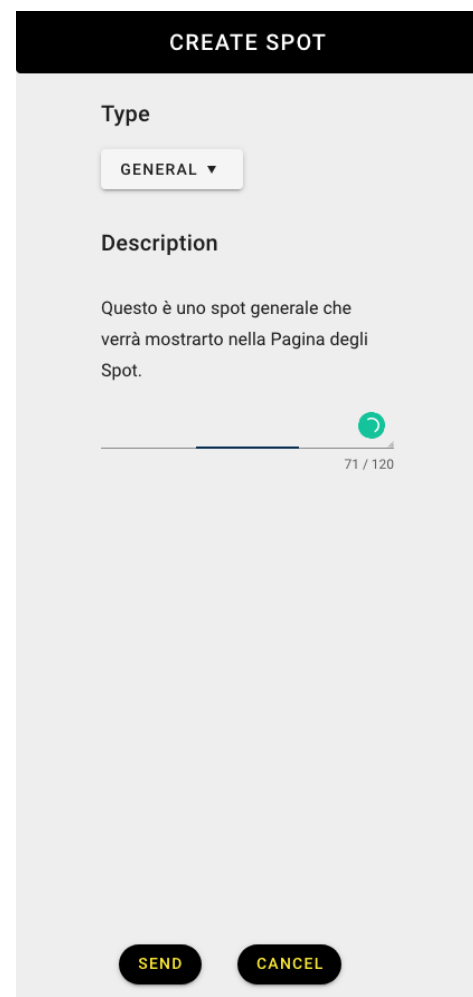
In basso a destra, è possibile accedere alla pagina per creare uno spot.



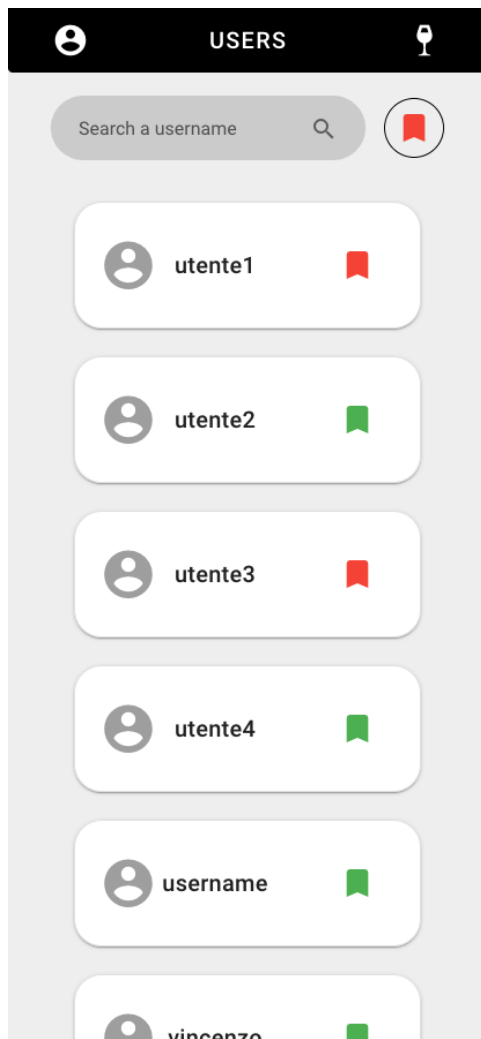
Dalla pagina degli spot è possibile mettere *like* ad uno specifico. Il numero di like di uno spot ne decreta l'ordine in cui, al prossimo caricamento della pagina (da parte di qualsiasi utente), i vari spot vengono mostrati. Si può tradurre uno spot dalla lingua italiana a quella inglese (il processo inverso non è ancora supportato). Altra feature non ancora disponibile è quella di contattare l'autore di uno spot.

Dalla barra di navigazione è possibile andare alla pagina contenente informazioni riguardanti altri utenti oppure tornare alla home page, dove è mostrata la lista di eventi.

Questa semplice pagina permette la creazione di uno spot **generale** e uno **specifico** (non ancora supportato). Quando uno spot viene creato avrà un numero di like pari a 0 e di conseguenza verrà mostrato verso la fine nella Pagina degli Spot (in ordine di *popolarità/likes*). Attraverso il pulsante **cancel** si può tornare alla schermata visitata in precedenza annullando l'operazione di creazione di uno spot.

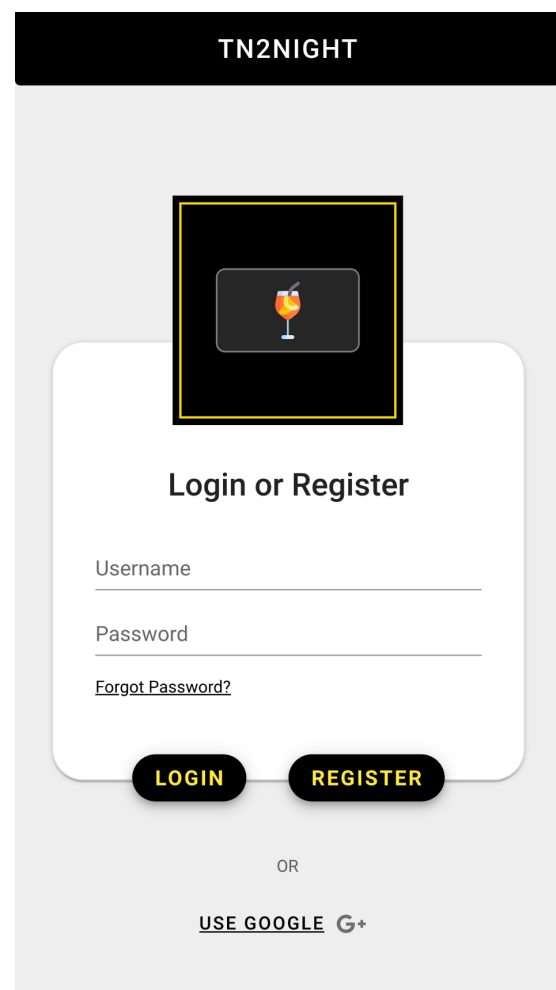


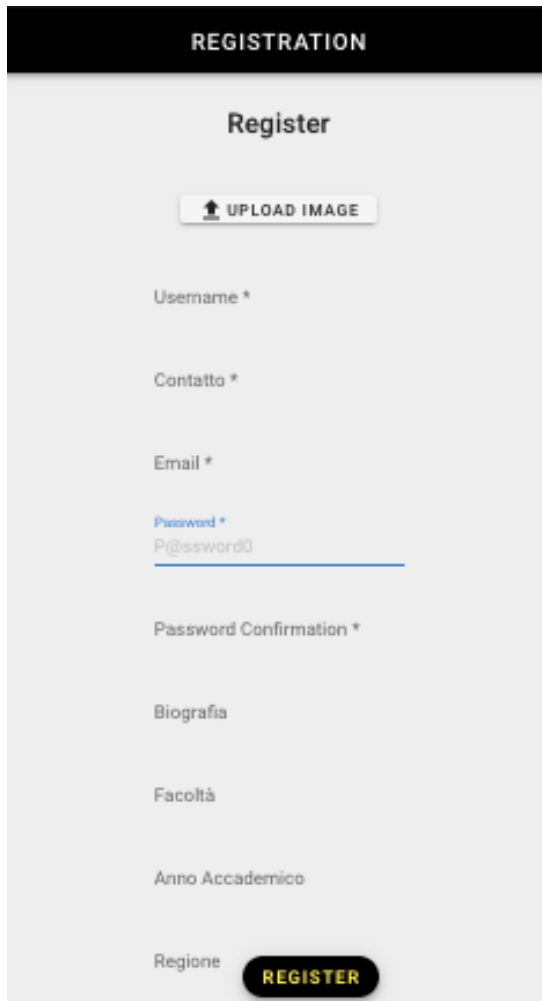




In questa schermata è possibile interagire con gli altri utenti iscritti alla piattaforma. Sarà possibile salvarli come amici, toglierli dalla lista personale degli amici, cercare un utente specifico conoscendo il suo username ed, infine, permettere un filtraggio di tutti gli amici tramite il bottone posizionato accanto alla barra di ricerca.

Ogni qualvolta si voglia interagire con l'applicazione (es. seguire un evento, accedere alla propria pagina profilo, cercare altri utenti, mettere like ad uno spot oppure creare uno spot) se non si ha effettuato l'accesso, verrà mostrata la pagina per effettuare l'autenticazione così da verificare l'utente. Da questa pagina è anche possibile accedere alla pagina per creare un nuovo account.





The screenshot shows a registration form titled 'REGISTRATION' with a sub-header 'Register'. It includes an 'UPLOAD IMAGE' button, followed by input fields for 'Username \*', 'Contatto \*', 'Email \*', 'Password \*' (with a placeholder 'P@ssword0'), 'Password Confirmation \*', 'Biografia', 'Facoltà', 'Anno Accademico', and 'Regione'. A yellow 'REGISTER' button is at the bottom right.

Se è la prima volta che si utilizza la web app è possibile creare un account da zero con tale schermata. Non è ancora possibile aggiungere una foto, ma i campi che sono obbligatoriamente richiesti hanno un asterisco (\*) accanto all'etichetta. Una volta effettuata la registrazione bisognerà fare l'accesso e si potrà finalmente interagire con la web app.

In questa pagina, accessibile dalla barra di navigazione della home page, permette di avere un resoconto delle informazioni inserite durante la fase di registrazione. Da qua è possibile, eventualmente modificare tale informazioni (non implementato ancora) oppure effettuare il logout.



The screenshot shows a user profile page titled 'PROFILE' with a back arrow icon. It features a circular profile picture placeholder with a drink icon. Below the picture is the username 'utente2'. The page lists user details: 'email' (utenteprova2@gmail.com), 'contatto' (@utente2), 'bio' (Ciao!), 'facoltà' (Informatica), and 'anno' (partially visible). At the bottom are 'EDIT' and 'GO BACK' buttons, and a page number '2'.

# GitHub Repository & Deployment Guide

La repository di GitHub è disponibile al link: [GitHub Repository](#)

I contenuti dell'applicazione sono disponibili nelle repositories back-end, front-end. All'interno di tali cartelle è presente il codice sviluppato per le rispettive parti dell'applicazione.

L'applicazione è deployata al seguente link: [TN2night.com](http://TN2night.com)

nel caso in cui lo si desiderasse è possibile eseguire il server in locale con il seguente comando:

## **npm install && npm start**

Sarà poi necessario cambiare la variabile

**baseURL = "<http://localhost:3000>";**

contenuta dentro il file **src > api > init.js**

avviare l'applicazione con

## **npm install && npm run serve**

la web app sarà attiva all'indirizzo <http://localhost:8080/>

## API Testing

Per testare il nostro progetto abbiamo creato la cartella "testing" con all'interno tre file che andranno a testare le funzioni dell'api che la nostra applicazione necessita. I file effettuano dei test per quanto riguarda le operazioni effettuate sugli utenti, spot ed eventi. I test ripropongono gran parte delle situazioni che possono verificarsi nell'utilizzo quotidiano da parte dell'applicazione per dimostrarne la sua robustezza.

### Operazioni testate

#### **Utente (user.test.js):**

- Get di tutti gli utenti
- Get di un singolo utente
- Registrazione di un utente correttamente
- Errore nel caso in cui avvenga la registrazione di un utente già registrato

- Errore nel caso di inserimento di un utente senza specifico username
- Aggiunta di un amico
- Errore se viene aggiunto un amico nonostante ci sia già una relazione di amicizia tra i due utenti
- Rimozione di un amico
- Get di tutti gli amici di un determinato utente

**Evento (event.test.js):**

- Get di tutti gli eventi
- Inserimento di un nuovo evento
- Registrazione di un utente ad un evento
- Get di tutti gli eventi seguiti da un utente
- Errore nel caso di inserimento di un utente senza specifico username
- Aggiunta di un amico
- Rimozione della registrazione di un utente ad un evento

**Spot (spot.test.js):**

- Get di tutti gli spot
- Inserimento di un nuovo spot
- Aggiunta di un like ad uno spot
- Errore nel inserire un doppio like allo stesso spot
- Rimozione di un like ad uno spot

```
veenca@archveenca ~/S/U/I/Project (main) [1]> npm test -- --coverage
```

```
> tn2night@1.0.0 test
```

```
> jest --coverage
```

```
PASS testing/event.test.js
```

- Console

```
console.log
  localhost
```

```
    at Object.log (app.js:3:9)
```

```
console.log
  root
```

```
    at Object.log (app.js:4:9)
```

```
PASS testing/spot.test.js
```

- Console

```
console.log
  localhost
```

```
    at Object.log (app.js:3:9)
```

```
console.log
  root
```

```
    at Object.log (app.js:4:9)
```

```
PASS testing/user.test.js (5.228 s)
```

- Console

```
console.log
  localhost
```

```
    at Object.log (app.js:3:9)
```

```
console.log
  root
```

```
    at Object.log (app.js:4:9)
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
app.js	100	100	100	100	

```
Test Suites: 3 passed, 3 total
```

```
Tests: 19 passed, 19 total
```

```
Snapshots: 0 total
```

```
Time: 5.888 s
```

```
Ran all test suites.
```

```
veenca@archveenca ~/S/U/I/Project (main)> █
```