

Progetto Ingegneria Software

NomadBees

Gruppo T-37:

Matteo Pontalti, Matteo Bregola, Riccardo
Libanora.

Sviluppo Applicazione

Contenuti

| | |
|--------------------------------------|----|
| Scopo del Documento | 4 |
| 1. Resource Extraction Diagram | 5 |
| 2. Resource Modelling Diagram | 7 |
| 3. Struttura del Progetto | 8 |
| Servizi e piattaforme | 8 |
| Struttura | 8 |
| 4. Implementazione API | 10 |
| Introduzione | 11 |
| 1. Authenticate | 12 |
| 2. NewUser | 14 |
| 3. SearchUser | 16 |
| 4. EditUser | 18 |
| 5. FollowUser | 19 |
| 6. NewViaggio | 21 |
| 7. SearchViaggio | 23 |
| 5. Documentazione | 24 |
| 6. Testing | 26 |
| Jest | 26 |
| Postman | 29 |
| 1) Iscrizione al sito | 29 |
| 2) Autenticazione | 30 |
| 3) Pubblicazione di un viaggio | 30 |
| 4) Ricerca di un viaggio | 31 |
| 5) Ricerca di un utente | 32 |
| 6) Seguire un utente | 32 |
| 7) Modifica Profilo | 33 |

| | |
|----------------------------|----|
| 7. User Flow Diagram | 34 |
| 8. Front-end | 36 |
| Home | 37 |
| Luoghi | 38 |
| Seguiti | 39 |
| Crea | 40 |
| Profilo | 41 |
| 9. Deployment | 42 |

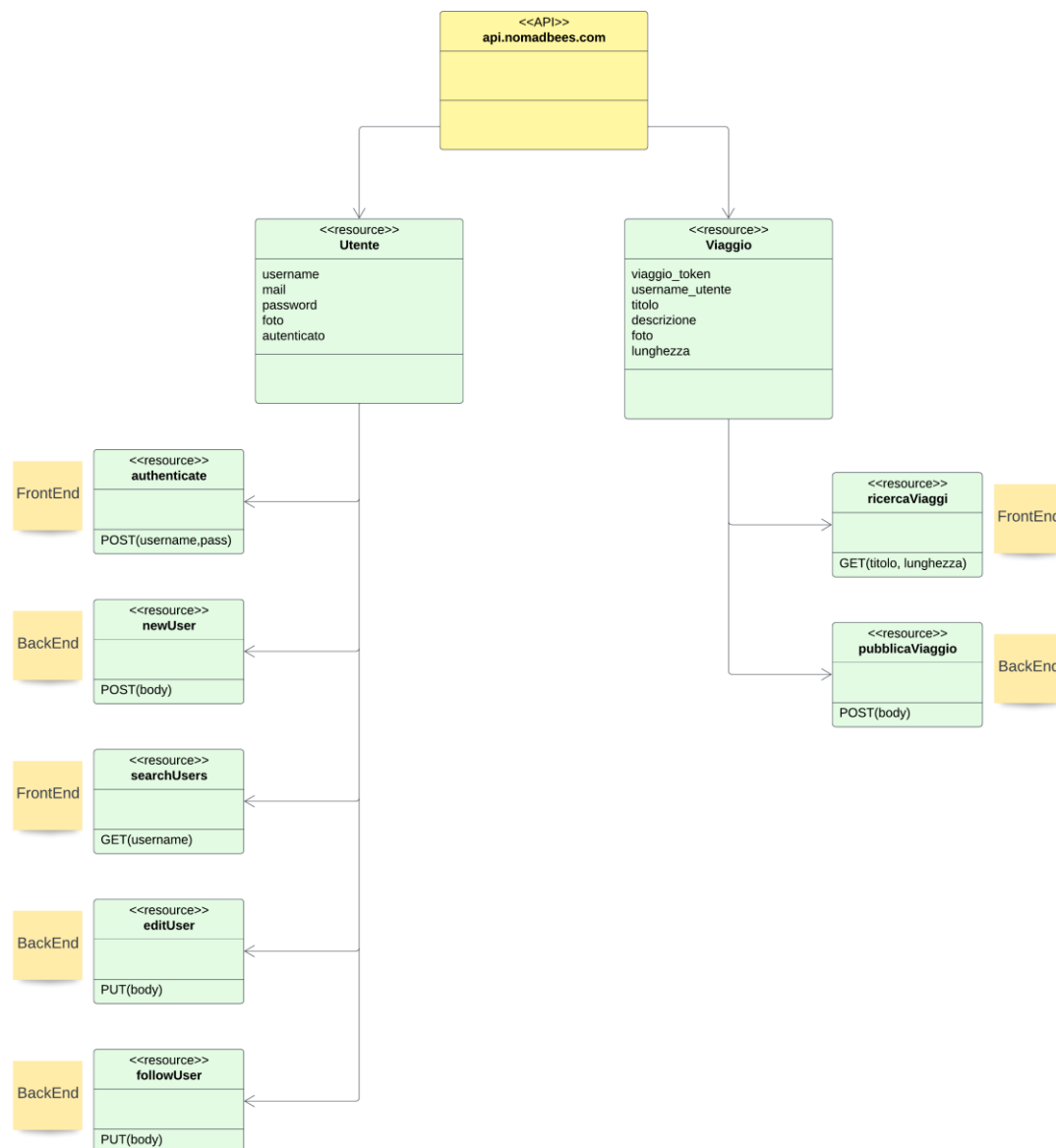
Scopo del Documento

Il seguente documento descrive come è avvenuta la parte terminale di sviluppo del progetto. Come prima cosa ci si è soffermati sull'estrazione delle risorse e successivamente sulla loro modellazione (Sezione 2 e 3 del documento). Una volta delineate le API base del sito si è proceduti con l'elaborazione della struttura del progetto per permettere un'organizzazione delle cartelle ed una miglior suddivisione del lavoro (Sezione 4). Successivamente vengono presentate in maniera concisa le implementazioni delle API (Sezione 5) e parte del codice. Nelle Sezioni 6 e 7 si trovano invece la loro documentazione ed un report sul testing effettuato. Le sezioni 8 e 9 presentano l'User Flow Diagram ed il Front-end. Infine viene esposto come è stato effettuato il Deployment del sito.

1. Resource Extraction Diagram

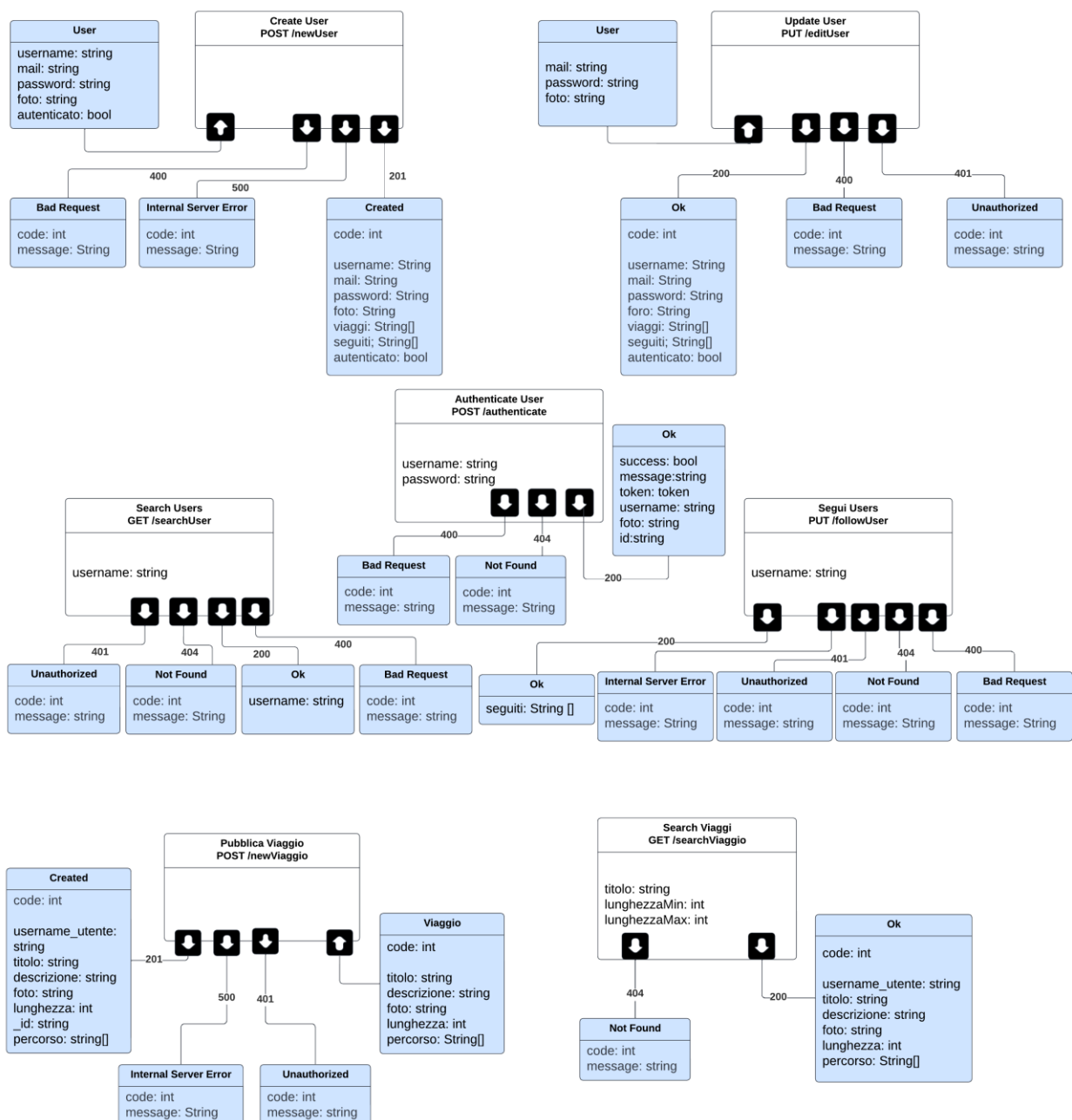
Il diagramma di estrazione delle risorse (Resource Extraction Diagram) è uno strumento utile per progettare le API di un progetto, in particolare per le API RESTfull. Esso descrive le risorse del sistema sotto forma di entità, dove ogni entità viene rappresentata come una risorsa che può essere letta, modificata o eliminata attraverso i metodi HTTP appropriati (GET, POST, PUT, DELETE...). Le risorse sono la parte fondamentale dell'API e sono rappresentate da un oggetto contenente l'URI, il tipo di richiesta http e i parametri o corpo della richiesta. Ogni risorsa API presenta quindi un servizio offerto dal sito accessibile grazie al percorso specificato. Inoltre sono state aggiunte delle indicazioni sul ruolo delle API all'interno del sito: se esse modificano dati o creano dati salvati allora sono state affiancate dall'etichetta "Back-end" mentre se utilizzavano quest'ultimi per fornire informazioni all'utente utilizzatore allora l'etichetta utilizzata è "Front-end"

Abbiamo identificato le risorse a partire dal diagramma delle classi presentato nel Documento D3.



2. Resource Modelling Diagram

Nel diagramma qui riportato vengono presentate più in dettaglio le risorse API sopra indicate. Nello specifico per ogni risorsa vengono indicati: nome, metodo http, URI, corpo della richiesta (se presente) o i parametri ed uno o più corpi di risposta. Mentre i metodi erano già presenti nel diagramma precedente questo risulta particolarmente utile per capire come funzionano le API, che dati necessitano come input e cosa offrono come output. Le risposte forniscono anche i codici presenti nello standard http RFC 2616.



3. Struttura del Progetto

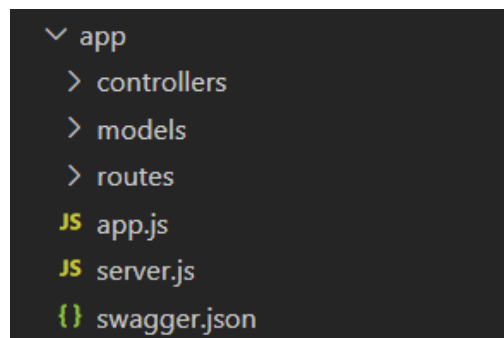
Servizi e piattaforme

Nel progetto è stato utilizzato **Node.js**, una piattaforma per lo sviluppo di applicazioni web basate su JavaScript integrato con **Express**. **MongoDB**, un database NoSQL basato su documenti per la memorizzazione dei dati del tuo progetto. **JSON Web Token (JWT)** è stato utilizzato per gestire l'autenticazione e l'autorizzazione delle richieste delle applicazioni attraverso l'utilizzo di token di accesso sicuri. **Swagger** è stato utilizzato per documentare le API web del progetto in modo semplice e veloce. **Jest**, un framework di test automatizzato per JavaScript, è stato sfruttato per scrivere e eseguire i test del tuo progetto.

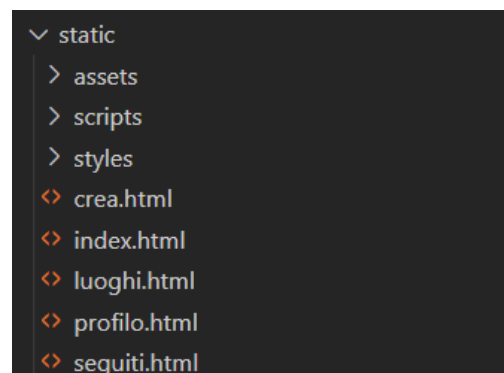
Struttura

Si è deciso di suddividere il progetto in tre cartelle di lavoro principali:

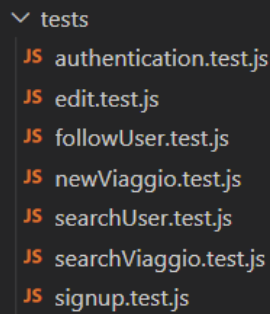
1. **App** contiene l'implementazione e gestione delle API descritte nelle sezioni precedenti (con l'aggiunta di tre API di "supporto" per il front-end), il file che gestisce la connessione al server ed il file swagger per la documentazione.



2. **Static** contiene gli script html, css e javascript che gestiscono il frontend.



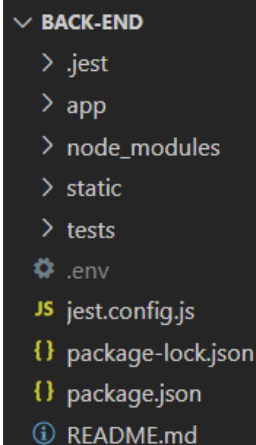
3. **Tests** contiene i test effettuati grazie alla libreria Jest.



```
▼ tests
JS authentication.test.js
JS edit.test.js
JS followUser.test.js
JS newViaggio.test.js
JS searchUser.test.js
JS searchViaggio.test.js
JS signup.test.js
```

In aggiunta alle cartelle principali vi sono dei file e cartelle di supporto/configurazione:

- Nel file **.env** vi sono le informazioni private per la gestione del database, dei token e del deployment che non sarà pubblicato.
- All'interno dei file **package.json** vi sono le informazioni sulle dipendenze delle librerie usate per lo sviluppo, informazioni sul progetto Node.js come autori e la definizione di script di avvio, build e testing
- Il file denominato **jest.config.js** insieme quello all'interno della cartella **.jest** configura dei parametri specifici per la gestione del testing.
- Vi è poi la cartella **node_modules** (la cui pubblicazione può essere omessa) che contiene i moduli importati per la realizzazione del progetto.
- Infine è presente un file **readme** che contiene alcune informazioni sul progetto in generale e le informazioni su come provarlo.



```
▼ BACK-END
> .jest
> app
> node_modules
> static
> tests
⚙ .env
JS jest.config.js
{} package-lock.json
{} package.json
📖 README.md
```

(Link per la repository su github: <https://github.com/SE-T37/Back-end>).

(Nota: la repository di Front-end è stata utilizzata solo in fase di progettazione e non risulta per tanto più aggiornata).

4. Implementazione API

In questa sezione viene riportato parte del codice scritto per sviluppare le API ed una breve spiegazione. Di seguito la lista completa:

- authenticate
- newUser
- searchUser
- editUser
- followUser
- newViaggio
- searchViaggio

Queste sono le API effettive che implementano le funzioni presentate nei documenti precedenti; inoltre vi sono tre API di “supporto” che sono state realizzate successivamente per gestire la visualizzazione dei dati nel front-end:

- getUsers
- getViaggi
- getViaggiAmici

Per queste ultime non è stato presentato il codice (che si può trovare all’interno del progetto) né la documentazione in Swagger, tuttavia si è provveduto a fare delle basilari funzioni di testing.

Introduzione

La cartella app contiene, come spiegato nella sezione precedente l'implementazione del back-end. Essa contiene due file denominati server.js ed app.js, la prima sfrutta la libreria Mongoose per connettersi al database mentre nella secondo vengono importati i moduli dalla cartella routes

```
const user = require('./routes/user');
```

e vengono registrate le rispettive funzioni middleware per la routes specificate.

```
app.use('/newUser', user);
```

Come specificato nei documenti precedenti alcuni servizi del sito richiedono che l'utente sia autenticato, per implementare la seguente funzionalità si è deciso di utilizzare la libreria JWT e la funzione tokenChecker. Le API che necessitano di questi metodi sfruttano oltre al codice sopra riportato il tokenchecker: (`app.use('/searchUser', tokenChecker)`).

La gestione delle API si sviluppa quindi attraverso 3 passaggi principali:

1. Il file **app.js** si occupa di reindirizzare il controllo delle azioni al corretto file di route in base all'URI .
2. Il **file route** specifico (presente nella cartella routes) indica quali funzioni controller sono accessibili da quel URI e i rispettivi di metodi http . *Es file user.js nella cartella routes:*

```
const UserController = require('../controllers/user');  
router.post('', upload.none(), UserController.newUser);
```

Indica che è possibile utilizzare la funzione newUser (implementata nella cartella controllers) se utilizza il presente URI (in quanto nella funzione post il primo parametro è una stringa vuota). Inoltre indica che la funzione newUser si baserà su una richiesta di tipo POST.

3. Nella cartella controller è possibile trovare l'implementazione delle funzioni controller.

Inoltre è presente una cartella "models" che contiene i modelli Mongoose dell'oggetto utente e viaggio. Questi saranno importati ed utilizzati dai file controllers.

1. Authenticate

Accesso locale: <http://localhost:8080/authenticate>

Metodo: **POST**.

Parametri richiesta: **username, password**.

Risposta attesa: creazione token o rifiuto autenticazione.

L'API di autenticazione si basa sull'utilizzo della libreria JSON Web Token (JWT) la quale permette di identificare gli utenti attraverso un token crittografato. Questa libreria viene importata all'inizio del file insieme al modello User, necessario per accedere ai vari dati dell'utente, in particolare username e password.

```
const authenticate = async function(req, res, next) {  
  let user = await User.findOne({  
    username: req.body.username  
  }).exec();
```

La prima parte della funzione authenticate è di ricercare nel database un Utente con l'username uguale a quello passato nella richiesta (*Nota: si è deciso in fase di progettazione di rendere gli username unici*).

```
if (!user) {  
  res.status(404).json({ success: false, message: 'Authentication  
failed. User not found.' });  
}
```

Se l'utente non è presente nel database non è necessario controllare che la password sia corretta e si ritorna l'errore 404.

```
else{  
  if (user.password !== req.body.password) {  
    res.status(400).json({ success: false, message:  
'Authentication failed. Wrong password.' });  
  }
```

Se l'utente è presente ma la password non corrisponde l'autenticazione fallisce.

```
else{  
    user.autenticato=true;  
    var payload = {  
        username: user.username,  
        id: user._id  
    }  
    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,  
        username: user.username,  
        foto: user.foto,  
        id: user._id,  
    });  
}
```

Se la password corrisponde allora viene creato il token inserendo nelle informazioni criptate il suo username, l'ID e il tempo di scadenza del token. La funzione di creazione del token richiede l'utilizzo di una chiave segreta che viene adoperata nell'algoritmo di codifica; questa stringa è presente nel file .env per una questione di sicurezza. Una volta creato il token viene restituito come risposta insieme all'username e all'ID.

```
module.exports = { authenticate };
```

La funzione viene poi esportata per essere visibile dai file nella cartella routes.

Codice completo presente in: back-end/app/controllers/authentication.js.

2. NewUser

Accesso locale: <http://localhost:8080/newUser>

Metodo: **POST**.

Parametri richiesta: **username**, **mail**, **password**, **foto**.

Risposta attesa: creazione profilo.

```
const newUser = (req, res, next) => {  
  let username= req.body.username;  
  let password= req.body.password;  
  let mail= req.body.mail;  
  let foto= req.body.foto;  
  if(username== null || password== null || mail== null)  
    return res.status(400).json({message: "Missing credentials"});  
}
```

La prima azione eseguita è il recupero e controllo delle credenziali passate alla richiesta, quest'operazione sarebbe sovrabbondante in quanto viene effettuato un esame di esse precedentemente nel front-end ma garantisce una maggior sicurezza dell'API.

```
User.findOne({ username:username}, (err, data) => {
```

Successivamente si controlla l'eventuale presenza nel database di un utente con lo stesso username

```
  if (err) return  
    res.status(500).json('Something went wrong, please try again.  
    ${err}');
```

Viene controllato il primo scenario: la ricerca non va a buon termine a causa di un errore del database.

```
if(!data){  
  const newUser = new User({  
    username: username,  
    mail: mail,  
    password: password,
```

```
        foto: foto,  
    })  
    newUser.save( (err,data) => {  
        if (err)  
            return res.status(500).json({message:"Error saving user"});  
        else  
            return res.status(201).json(data);  
    })
```

Se invece non occorrono errori e non vengono trovati utenti con lo stesso username se i crea un utente con i valori della richiesta e si salva nel database. Se il salvataggio va a buon fine si ritorna il codice 201 altrimenti viene restituito il 500.

```
}else{  
    return res.status(400).json({message: "User already exists"});  
}
```

Se l'username è già presente viene notificato il problema.

Codice completo presente in: back-end/app/controllers/user.js.

3. SearchUser

Accesso locale: <http://localhost:8080/searchUser>

Metodo: GET.

Parametri richiesta: username.

Risposta attesa: Username utenti contenenti la stringa ricercata.

```
let username= req.query.username;
  if(username==null){

    return res.status(400).json({message: "Bad request"});

  }
```

Come prima azione viene controllata la correttezza della richiesta, assicurandosi che siano stati passati i parametri corretti.

```
const searchUsers = async function(req, res, next){
  let users = await User.find({ username: { $regex: username ,
$options: 'i' } });
```

Viene fatta una richiesta al database di trovare tutti gli username che contengono quello presente nella richiesta.

```
if(users.length==0){

  return res.status(404).json({message: "User not found"});

}
```

Se non è presente nessun username corrispondente viene ritornato l'errore 404 e specificato che non è stato trovato nessun utente.

```
else{

  users = users.map((user) => {
    return {
      username: user.username
      foto: user.foto,
    };
  });
  return res.status(200).json(users);
}
```


Se invece uno o più utenti corrispondono a quello ricercato allora vengono ritornati username, foto e il codice 200. Per questa, come per tutte le API che sfruttano il tokenChecker, l'errore 401 di "utente non autorizzato" viene gestito dalla funzione di tokenChecker.

Codice completo presente in: back-end/app/controllers/search.js.

4. EditUser

Accesso locale: <http://localhost:8080/editUser>

Metodo: **PUT**.

Parametri richiesta: [foto/mail/password](#).

Risposta attesa: Cambio dati profilo utente.

```
const editUser = async function(req, res, next){
  User.findByIdAndUpdate(req.loggedUser.id, req.body, {new:true},
(err,user) =>{
    if(err) return res.status(400).json({Error: err});
    else{
      return res.status(200).json(user);
    }
  });
};
```

La funzione edit user sfrutta la funzione `findByIdAndUpdate` di Mongoose la quale richiede come primo parametro l'id dell'utente che si vuole modificare e come secondo i dati del profilo che si vogliono modificare. Il primo parametro è reso disponibile dalla funzione di `tokenChecker` mentre il secondo è parte della richiesta stessa. Se l'operazione si conclude correttamente viene ritornato il codice 200 e l'utente,

Codice completo presente in: `back-end/app/controllers/edit.js`.

5. FollowUser

Accesso locale: <http://localhost:8080/followUser>

Metodo: **PUT**.

Parametri richiesta: **username**.

Risposta attesa: Aggiunta dell'username indicato alla lista dei seguiti.

```
const seguiUser = async function(req, res, next){
  const id_richiedente=req.loggedUser.id;
  let user_to_follow = await User.findOne({ username: req.body.username
});
  let user_richiedente= await User.findOne({ _id: id_richiedente});
```

La prima parte della funzione si occupa di controllare la presenza e recuperare l'oggetto utente per l'utente che si vuole seguire dato l'username. Viene fatta la stessa operazione per l'utente che sta utilizzando la funzione (anche se in questo caso sicuramente si otterrà un riscontro dato che è stato precedentemente eseguito il tokenChecker).

```
if(user_to_follow==null){
  return res.status(404).json({message: "User not found"});
}
```

Se l'utente che si è richiesto di seguire non viene trovato si restituisce un errore e si termina l'esecuzione.

```
else{
  if(user_richiedente.seguiti.includes(req.body.username)){
    return res.status(400).json({ message: "Already following"});
  }
```

Successivamente si controlla che l'utente che si vuole aggiungere non sia già presente tra la lista di quelli seguiti. In caso affermativo si segnala l'errore.

```
else{
  user_richiedente.seguiti.push(user_to_follow.username);
  try{
    await user_richiedente.save();
    return res.status(200).json(user_richiedente.seguiti)
```

```
}catch(e){return res.status(500)};
```

Se l'utente esiste nel database e non è tra quelli seguiti dal richiedente allora si prova ad aggiungerlo alla lista, se l'operazione fallisce viene ritornato il codice 500 altrimenti viene ritornato il codice 200 e la lista degli utenti seguiti.

Codice completo presente in: back-end/app/controllers/segui.js.

6. NewViaggio

Accesso locale: <http://localhost:8080/newViaggio>

Metodo: **POST**.

Parametri richiesta: **titolo**, **descrizione**, **foto**, **lunghezza**, **tappe**.

Risposta attesa: Pubblicazione di un viaggio da parte di un Utente.

```
const newViaggio = (req, res, next) => {  
  const newViaggio = new Viaggio({  
    username_utente: req.loggedUser.username,  
    titolo: req.body.titolo,  
    descrizione: req.body.descrizione,  
    foto: req.body.foto,  
    lunghezza: req.body.lunghezza,  
  })
```

La prima parte della funzione si occupa di creare un oggetto newViaggio (utilizzando il modello Viaggio importato). Vengono impostate le informazioni fondamentali quali nome, titolo, descrizione, foto e lunghezza del viaggio.

```
for(let i=1; i<=10; i++){  
  eval('var fotoForm' + ' = ' + 'req.body.foto' + i + ';' );  
  eval('var descrizioneForm' + ' = ' + 'req.body.descrizione' + i + ';' );  
  eval('var latitudineForm' + ' = ' + 'req.body.latitudine' + i + ';' );  
  eval('var longitudineForm' + ' = ' + 'req.body.longitudine' + i +  
    ';' );  
  newViaggio.percorso.push({  
    foto:fotoForm,  
    descrizione:descrizioneForm,  
    latitudine:latitudineForm,  
    longitudine:longitudineForm,  
  })  
}
```

Nella seconda parte vengono aggiunte le tappe al viaggio utilizzando un form per tappa (fino ad un massimo di 10 come specificato durante la progettazione).

```
newViaggio.save( (err,data) => {  
  if (err) {  
    return res.status(500).json({Error: err});  
  }  
  else{  
    user_richiedente.viaggi.push(newViaggio._id);  
    user_richiedente.save();  
    return res.status(201).json(data);  
  }  
})
```

Una volta impostate tutte le informazioni necessarie per pubblicare il viaggio si prova a salvarlo nel database. Se l'operazione va a buon fine si salva l'id tra i viaggi dell'utente che lo sta pubblicando, si ritorna il codice 201 e il viaggio, altrimenti l'errore 500.

Codice completo presente in: back-end/app/controllers/viaggio.js.

7. SearchViaggio

Accesso locale: <http://localhost:8080/searchViaggio>

Metodo: GET.

Parametri richiesta: luogo, lunghezza minima, lunghezza massima.

Risposta attesa: viaggi che rispettano i filtri.

```
const searchViaggio = async function(req, res, next){
  let viaggi = await Viaggio.find({
    luogo: { $regex: req.query.luogo , $options: 'i' },
    lunghezza: { $gte: req.query.lunghezzaMin, $lte:
      req.query.lunghezzaMax}
```

La funzione si occupa di ricercare un viaggio che abbia come titolo parte di quello passato come parametro della richiesta e come lunghezza, una lunghezza tra il valore minimo e quello massimo.

```
if (viaggi.length==0){
  return res.status(404).json({message: "Nessun viaggio trovato"});
}
```

Se non sono stati trovati viaggi con questi parametri viene ritornato il codice 404.

```
else{
  viaggi = viaggi.map((viaggio) => {
    return {
      username_utente: viaggio.username_utente,
      titolo: viaggio.titolo,
      descrizione: viaggio.descrizione,
      foto: viaggio.foto,
      lunghezza: viaggio.lunghezza,
      percorso: viaggio.percorso
    };
  });
  return res.status(200).json(viaggi);
}
```

Se sono stati trovati viaggi allora vengono ritornati.

Codice completo presente in: *back-end/app/controllers/searchViaggio.js*

5. Documentazione

La documentazione delle API è stata realizzata tramite la libreria Swagger. Il file contenente il codice che la implementa è denominato come *swagger.json* mentre la documentazione è accessibile al URL *api-docs* (es in locale: <http://localhost:8080/api-docs>).

| | |
|---|---|
| Authentication | ^ |
| POST /authenticate User login | v |
| Signup | ^ |
| POST /newUser New user signup | v |
| searchUser | ^ |
| GET /searchUser Search users | v |
| editUser | ^ |
| PUT /editUser Edit a user profile | v |
| followUser | ^ |
| PUT /followUser Follow another user profile | v |
| newViaggio | ^ |
| POST /newViaggio Post a new viaggio | v |
| searchViaggio | ^ |
| GET /searchViaggio Search a viaggio | v |

Per ogni API si possono ottenere le seguenti informazioni:

- Il metodo http utilizzato;
- L'endpoint da utilizzare per accedervi;
- I parametri richiesti e come vengono passati (tutti grazie ad un body tranne per i metodi GET poiché come specificato nella specifica RFC 2616 essi non accettano un body);
- Le risposte possibili con i codici dello standard e in alcuni casi il modello di risposta.

Nota: come indicato all'interno della documentazione bisogna prestare attenzione ai parametri da passare quando si vuole provare le API. Questo poiché le API dalla seconda in poi (esclusa la searchViaggio) necessitano di un token e quindi se non si provvede ad inserirlo nell'apposito campo allora tutte le esecuzioni ritorneranno il codice 401 "non autorizzato". Si consiglia quindi di utilizzare l'autenticazione per recuperare il token da passare alle altre API. Non è stato possibile settare un token valido di default in quanto si è deciso di impostare il tempo di vita del token pari a 24 ore, ciò avrebbe portato il token di default ad essere inutilizzabile.

Riportiamo un esempio di come è stata documentata l'API di Signup.

Signup

POST /newUser New user signup

Parameters

Try it out

| Name | Description |
|--------------------|--|
| Data object (body) | <p>Data of the new user profile</p> <p>Example Value Model</p> <pre>{ "username": "newUser1", "password": "randompassword", "mail": "test@gmail.com", "foto": "string" }</pre> <p>Parameter content type</p> <p>application/json</p> |

I parametri da passare alla richiesta vengono in questo caso inseriti in un body e come indicato dalla descrizione sono i dati dell'utente necessari per l'iscrizione. In questo caso vi sono dei valori di default come "newUser1" per testare l'API ma sono modificabili a piacere.

Responses

Response content type application/json

| Code | Description |
|------|---|
| 201 | <p>Created</p> <p>Example Value Model</p> <pre>{ "username": "string", "password": "string", "mail": "string", "foto": "string", "viaggio": [], "seguiti": [], "autenticato": true, "_id": "string", "_v": "string" }</pre> |
| 400 | Bad request |
| 401 | Unauthorized |
| 500 | Internal Server Error |

Vi sono infine le possibili risposte coerenti con quanto riportato nell'Resource Modelling Diagram (Sezione 3) e nell'implementazione (Sezione 5).

In questo caso sono 4 e la prima riporta anche l'informazione che si aspetta una risposta che contenga il modello User. (I modelli definiti sono User e Viaggio e sono accessibili nella parte terminale della documentazione).

6. Testing

E' stato effettuato il grazie all'utilizzo della libreria Jest e la piattaforma Postman.

Jest

Sono state sviluppate **9 Test Suites** per un totale di **30 Tests**. Al momento del deployment viene i risultati dei test sono:

- Test superati **30/30**
- Linee analizzate: **99.62%**
- Funzioni analizzate: **100%**

| File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s |
|-------------------|---------|----------|---------|---------|-------------------|
| All files | 99.62 | 98.14 | 100 | 99.62 | |
| app | 100 | 100 | 100 | 100 | |
| app.js | 100 | 100 | 100 | 100 | |
| app/controllers | 99.34 | 98.07 | 100 | 99.34 | |
| authentication.js | 100 | 100 | 100 | 100 | |
| edit.js | 100 | 100 | 100 | 100 | |
| getUsers.js | 100 | 100 | 100 | 100 | |
| getViaggi.js | 100 | 100 | 100 | 100 | |
| getViaggiAmici.js | 100 | 100 | 100 | 100 | |
| search.js | 100 | 100 | 100 | 100 | |
| searchViaggio.js | 100 | 100 | 100 | 100 | |
| segui.js | 100 | 100 | 100 | 100 | |
| tokenChecker.js | 100 | 100 | 100 | 100 | |
| user.js | 100 | 100 | 100 | 100 | |
| viaggio.js | 95.91 | 75 | 100 | 95.91 | 36-37 |
| app/models | 100 | 100 | 100 | 100 | |
| user.js | 100 | 100 | 100 | 100 | |
| viaggio.js | 100 | 100 | 100 | 100 | |
| app/routes | 100 | 100 | 100 | 100 | |
| authentication.js | 100 | 100 | 100 | 100 | |
| edit.js | 100 | 100 | 100 | 100 | |
| getUsers.js | 100 | 100 | 100 | 100 | |
| getViaggi.js | 100 | 100 | 100 | 100 | |
| getViaggiAmici.js | 100 | 100 | 100 | 100 | |
| search.js | 100 | 100 | 100 | 100 | |
| searchViaggio.js | 100 | 100 | 100 | 100 | |
| segui.js | 100 | 100 | 100 | 100 | |
| user.js | 100 | 100 | 100 | 100 | |
| viaggio.js | 100 | 100 | 100 | 100 | |

(Provare ad eseguire npm test)

Il risultato del testing risulta quindi coprire la maggior parte del codice scritto, si nota esclusivamente la mancanza di analisi dello scenario di un caso di errore del server nella funzione di salvataggio dell'API per la pubblicazione dei viaggi.

La maggior parte delle funzioni di testing eseguono effettivamente il codice connettendosi al database e leggendo/modificando dati, riportiamo un esempio da

back-end/tests/authentication.test.js:

```
const request= require("supertest");
const app = require("../app/app");
const mongoose = require("mongoose");
mongoose.set('strictQuery',false);

beforeAll(async () => {
    jest.setTimeout(8000);
    app.locals.db = await mongoose.connect(process.env.MONGODB_URI);
});
afterAll(() => {
    mongoose.connection.close(true);
});
```

Come prima cosa vengono importate le librerie necessarie per eseguire il test, successivamente la funzione `beforeAll` si occupa di impostare un timer al test e di impostare la connessione al database; `afterAll` la chiude dopo ogni test.

```
const body1={
    username: 'NotExistingUsername',
    password: 'NotExistingPassword',
}

test('Authentication, wrong username ', ()=>{
    return request(app).post('/authenticate')
        .set('Accept', 'application/json')
        .send(body1)
        .expect(404);
})
```

In questo esempio di testing si vuole testare cosa accade se venisse passato all'API di autenticazione un username non presente nel database (ed una password casuale). La funzione di testing chiamata "Authentication, wrong username" si occupa di fare una richiesta di tipo post all'endpoint `/authenticate` passando alla richiesta il body sopra descritto ed aspettandosi uno status 404 (Errore: non trovato).

Come sopra annunciato alcune funzioni di testing necessitano l'utilizzo di mock per simulare dei particolari scenari, ciò è tipico quando all'interno di un API si controlla più volte il funzionamento di un'azione del database. Difatti si può utilizzare un `"mongoose.connection.close(true)"` per testare un API simulando l'errata connessione con il database. Questa azione ha un effetto però su tutto il codice dell'API, quindi se si vuole simulare un errore in un specifico punto è utile "creare" artificialmente l'errore. Riportiamo un esempio da *back-end/tests/followUser.test.js*:

```
test('Follow user, not following but connection fail' , ()=>{
  const mockSave = jest.fn(()=>{throw new Error('Internal server
error')}));
  User.prototype.save = mockSave;
  return request(app).put('/followUser')
    .set('Accept', 'application/json')
    .send(body3)
    .expect(500);
})
```

La funzione di mock in questo caso vuole fare fallire la funzione di save presente nell'API followUser (da *back-end/tests/segui.js*), qui riportata:

```
try{
  await user_richiedente.save();
  return res.status(200).json(user_richiedente.seguiti)
}catch(e){return res.status(500).json({message:"Internal server
error"})};
```

Senza però causare errori nelle precedenti funzioni che interagiscono con il database come:

```
let user_to_follow = await User.findOne({ username: req.body.username});
```

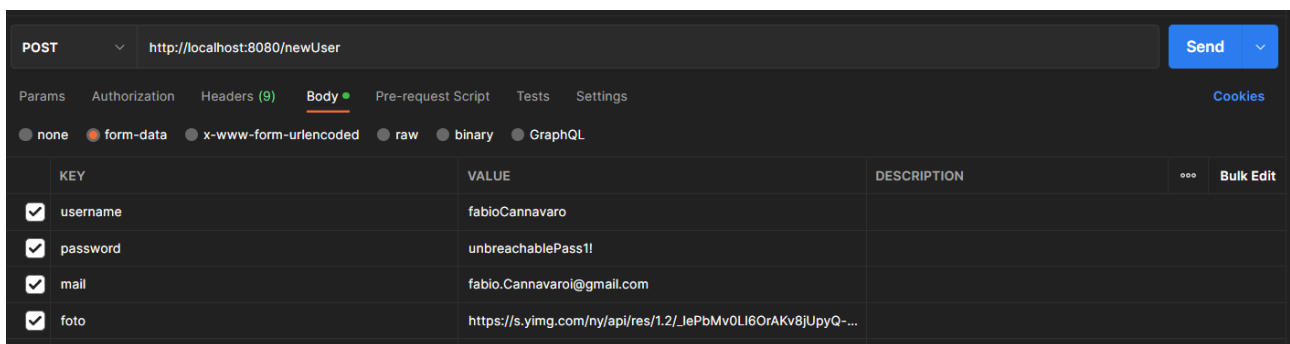
Codice Testing completo: *back-end/tests*

Postman

L'interazione tra Postman e il monitoring delle collections fornito da MongoDB è stato un passo fondamentale del testing. Esso difatti permette il testing diretto delle API specificando esclusivamente l'URI e i parametri da passare alla richiesta. Riportiamo un esempio di un possibile User Flow testato con Postman e mongoDB.

1) Iscrizione al sito

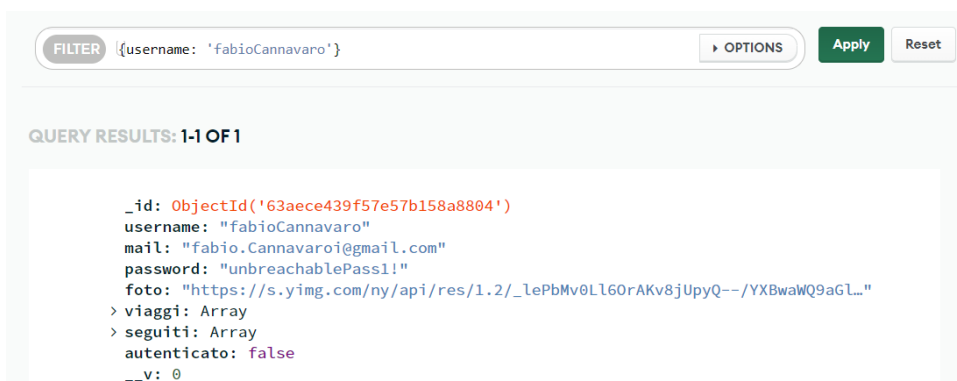
Richiesta all'endpoint corretto con i parametri specificati nel body.



Risposta del sito



Controllo salvataggio nel database:



2) Autenticazione

POST

http://localhost:8080/authenticate

Send

Params

Authorization

Headers (9)

Body

Pre-request Script

Tests

Settings

Cookies

☐ none

☒ form-data

☐ x-www-form-urlencoded

☐ raw

☐ binary

☐ GraphQL

| | KEY | VALUE | DESCRIPTION | ... | Bulk Edit |
|-------------------------------------|----------|-------------------|-------------|-----|-----------|
| <input checked="" type="checkbox"/> | username | fabioCannavaro | | | |
| <input checked="" type="checkbox"/> | password | unbreachablePass1 | | | |

Body

CookiesHeaders (7)Test Results

Status: 200 OKTime: 37 msSize: 746 BSave Response ▾

PrettyRawPreviewVisualizeJSON ↕

```
1{"success": true,  
2 "message": "Enjoy your token!",  
3 "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
4   eyJ1c2VybmFtZSI6ImZhbnVlY2FubmF2YXJvIiwiaWQiOiIyM2FhY2U0MzlmNTdNTDIyMTU0YTg4MDQwImlCPCJpYXNjOjE2NmZlODMA3MzAsImVhY2Ci6MTY3MjQ4NzEzMH8.  
    LaaxzNNAwTa_xQ1USIT1h5_RV2_Qq9qrGg3GpsyKw8Q",  
5 "username": "fabioCannavaro",  
6 "foto": "https://s.yimg.com/ny/api/res/1.2\_  
    notizie\_it\_154/89192de49d7b3a4a668660033fd51f83",  
7 "id": "63aece439f57e57b158a8804"  
8}
```

3) Pubblicazione di un viaggio

POST

http://localhost:8080/newViaggio

Send

Params

Authorization

Headers (9)

Body

Pre-request Script

Tests

Settings

Cookies

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

| | | | |
|-------------------------------------|--------------|---|--|
| <input checked="" type="checkbox"/> | descrizione1 | Visita guidata allo stadio di Berlino, costo 20€ | |
| <input checked="" type="checkbox"/> | latitudine1 | 52.514 | |
| <input checked="" type="checkbox"/> | longitudine1 | 13.239 | |
| <input checked="" type="checkbox"/> | foto2 | https://upload.wikimedia.org/wikipedia/commons/a/a9/Jewis... | |
| <input checked="" type="checkbox"/> | descrizione2 | Visita del museo ebraico | |
| <input checked="" type="checkbox"/> | latitudine2 | 52.300 | |
| <input checked="" type="checkbox"/> | longitudine2 | 13.234 | |

Risposta (parziale):

BodyCookiesHeaders (7)Test Results

Status: 201 CreatedTime: 78 msSize: 1.81 KBSave Response

```
1  {
2    "username_utente": "fabioCannavaro",
3    "titolo": "Gita a Berlino",
4    "descrizione": "Tour di Berlino: primo giorno visita agli stadi, secondo giorno visita ai musei, terzo giorno degustazione culinaria di
      specialità tipiche",
5    "foto": "https://www.travel365.it/foto/brandenburgertor-berlino-2.jpg",
6    "lunghezza": 15800,
7    "_id": "63aed2379f57e57b158a8818",
8    "percorso": [
9      {
10         "foto": "https://upload.wikimedia.org/wikipedia/commons/thumb/c/cd/Olympiastadion_Berlin_2015.jpg/1200px-Olympiastadion_Berlin_2015.jpg",
11         "descrizione": "Visita guidata allo stadio di Berlino, costo 20€",
12         "latitudine": "52.514",
13         "longitudine": "13.239",
14         "_id": "63aed2379f57e57b158a8819"
15       },
16       {
17         "foto": "https://upload.wikimedia.org/wikipedia/commons/a/a9/JewishMuseumBerlinAerial.jpg",
18         "descrizione": "Visita del museo ebraico",
19         "latitudine": "52.300",
20         "longitudine": "13.234",
21         "_id": "63aed2379f57e57b158a881a"
22       }
23     ]
24   }
```

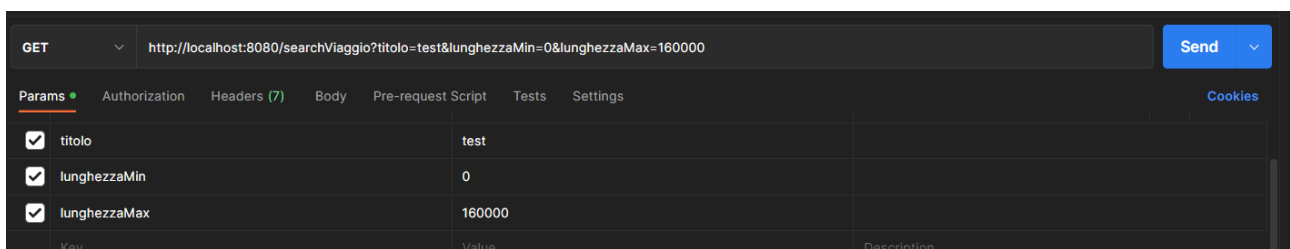
Controllo presenza viaggio nell'utente:

```
_id: ObjectId('63aece439f57e57b158a8804')
username: "fabioCannavaro"
mail: "fabio.Cannavaro@gmail.com"
password: "unbreachablePass1!"
foto: "https://s.yimg.com/ny/api/res/1.2/_lePbMv0Ll6OrAKv8jUpyQ--/YXBwaWQ9aG..."
viaggi: Array
  0: "63aed2379f57e57b158a8818"
> seguiti: Array
autenticato: false
```

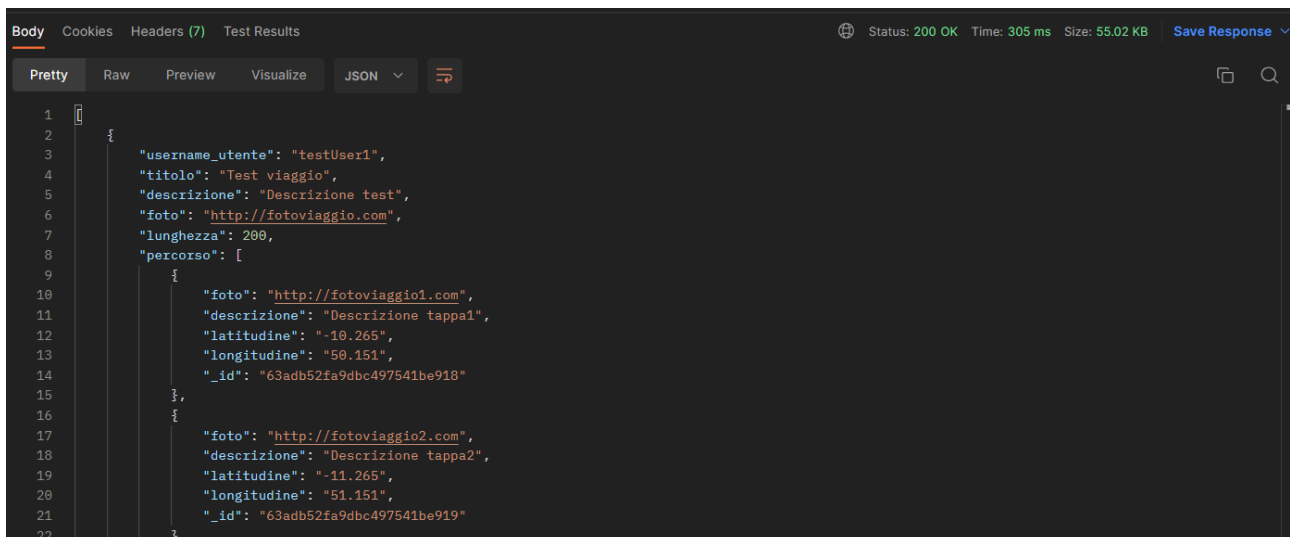
Controllo presenza viaggio:

```
_id: ObjectId('63aed2379f57e57b158a8818')
username_utente: "fabioCannavaro"
titolo: "Gita a Berlino"
descrizione: "Tour di Berlino: primo giorno visita agli stadi, secondo giorno
visita..."
foto: "https://www.travel365.it/foto/brandenburgertor-berlino-2.jpg"
lunghezza: 15000
> percorso: Array
__v: 0
```

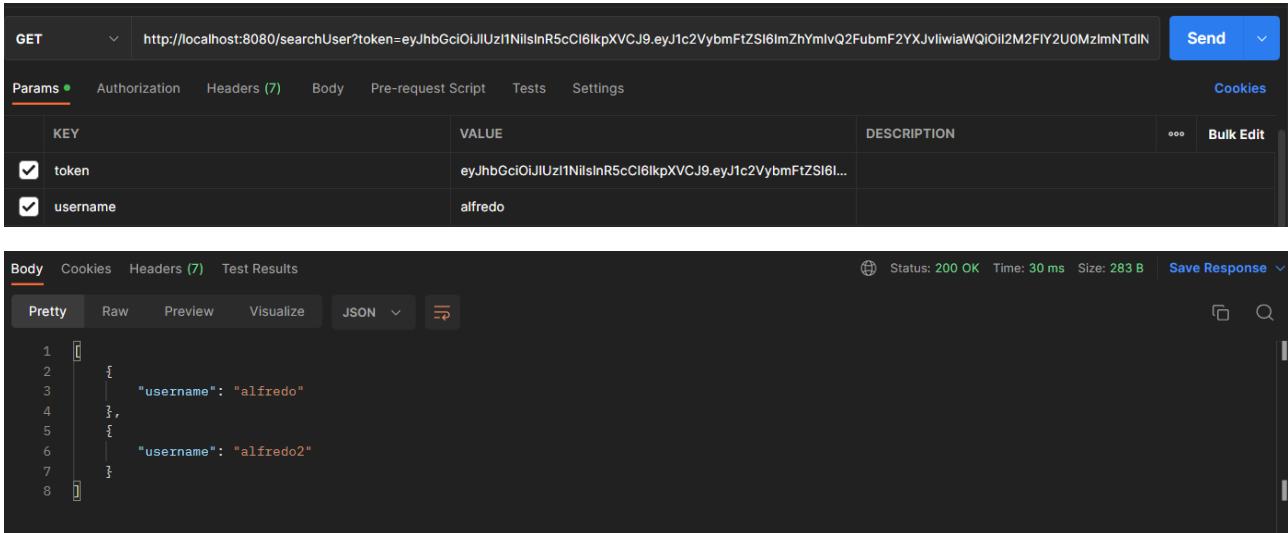
4) Ricerca di un viaggio



Risposta (parziale):



5) Ricerca di un utente





GET `http://localhost:8080/searchUser?token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImZhYmVmlvQ2FubmF2YXJvliwiaWQiOiI2M2FYI2U0MzlmNTdlIn` **Send**

Params | Authorization | Headers (7) | Body | Pre-request Script | Tests | Settings | Cookies

| KEY | VALUE | DESCRIPTION |
|--|---|-------------|
| <input checked="" type="checkbox"/> token | eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImZhYmVmlvQ2FubmF2YXJvliwiaWQiOiI2M2FYI2U0MzlmNTdlIn | |
| <input checked="" type="checkbox"/> username | alfredo | |

Body | Cookies | Headers (7) | Test Results | Status: 200 OK | Time: 30 ms | Size: 283 B | Save Response

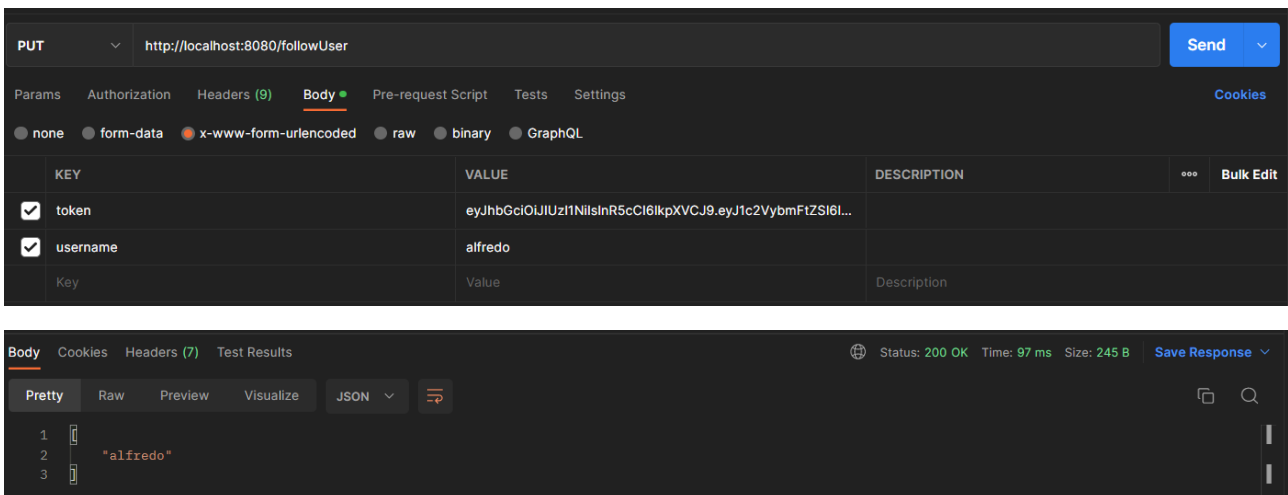
Pretty | Raw | Preview | Visualize | JSON |  

```

1  [
2    {
3      "username": "alfredo"
4    },
5    {
6      "username": "alfredo2"
7    }
8  ]

```

6) Seguire un utente





PUT `http://localhost:8080/followUser` **Send**

Params | Authorization | Headers (9) | **Body** | Pre-request Script | Tests | Settings | Cookies

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

| KEY | VALUE | DESCRIPTION |
|--|---|-------------|
| <input checked="" type="checkbox"/> token | eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImZhYmVmlvQ2FubmF2YXJvliwiaWQiOiI2M2FYI2U0MzlmNTdlIn | |
| <input checked="" type="checkbox"/> username | alfredo | |
| Key | Value | Description |

Body | Cookies | Headers (7) | Test Results | Status: 200 OK | Time: 97 ms | Size: 245 B | Save Response

Pretty | Raw | Preview | Visualize | JSON |  

```

1  [
2    "alfredo"
3  ]

```

Controllo salvataggio: tra i seguiti

```

_id: ObjectId('63aece439f57e57b158a8804')
username: "fabioCannavaro"
mail: "fabio.Cannavaro@gmail.com"
password: "unbreachablePass1!"
foto: "https://s.yimg.com/ny/api/res/1.2/_lePbMv0Ll60rAKv8jUpyQ--/YXBwaWQ9aG..."
> viaggi: Array
< seguiti: Array
  0: "alfredo"
autenticato: false

```


7) Modifica Profilo

The screenshot shows a REST client interface. The top bar indicates a PUT request to `http://localhost:8080/editUser`. The 'Body' tab is selected, showing form data with the following key-value pairs:

| KEY | VALUE | DESCRIPTION |
|----------|---|-------------|
| token | eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybWZtZSI6I... | |
| password | passWORD23! | |

The response body is shown in the bottom panel, displaying a JSON object with the following structure:

```

1 {
2   "_id": "63aece439f57e57b158a8804",
3   "username": "fabioCannavaro",
4   "mail": "fabio.Cannavaro@gmail.com",
5   "password": "passWORD23!",
6   "foto": "https://s.yimg.com/ny/api/res/1.2/_lePbMv0Ll6OrAKv8jUpYQ--/YXBwaWQ9aGlnaGxhbmRlcjR3PTY0MDtoPTQyNw--/https://media.zenfs.com/it/notizie_it_154/89192de49d7b3a4a668660033fd51f83",
7   "viaggi": [
8     "63aed2379f57e57b158a8818"
9   ],
10  "seguiti": [
11    "alfredo"
12  ],
13  "autenticato": false,
14  "__v": 2
15 }
```

Controllo avvenuta modifica password:

```

_id: ObjectId('63aece439f57e57b158a8804')
username: "fabioCannavaro"
mail: "fabio.Cannavaro@gmail.com"
password: "passWORD23!"
foto: "https://s.yimg.com/ny/api/res/1.2/_lePbMv0Ll6OrAKv8jUpYQ--/YXBwaWQ9aGlnaGxhbmRlcjR3PTY0MDtoPTQyNw--/https://media.zenfs.com/it/notizie_it_154/89192de49d7b3a4a668660033fd51f83"
> viaggi: Array
> seguiti: Array
autenticato: false
__v: 2
```

7. User Flow Diagram

In questa sezione del documento di sviluppo riportiamo gli “user flows” degli utenti che utilizzano la nostra applicazione. Come possiamo notare dal diagramma sottostante, l’utente ha molteplici azioni da poter compiere nella nostra applicazione. In particolare, non appena l’utente accede al sito, può navigare in 5 differenti pagine del sito: “Profilo”, “Seguiti”, “Crea”, “Registrazione”, “Luoghi”.

Per poter accedere alle pagine “Profilo”, “Seguiti” e “Crea” è prima necessario essersi autenticati mediante username e password; in caso contrario non sarà possibile visualizzare tali pagine.

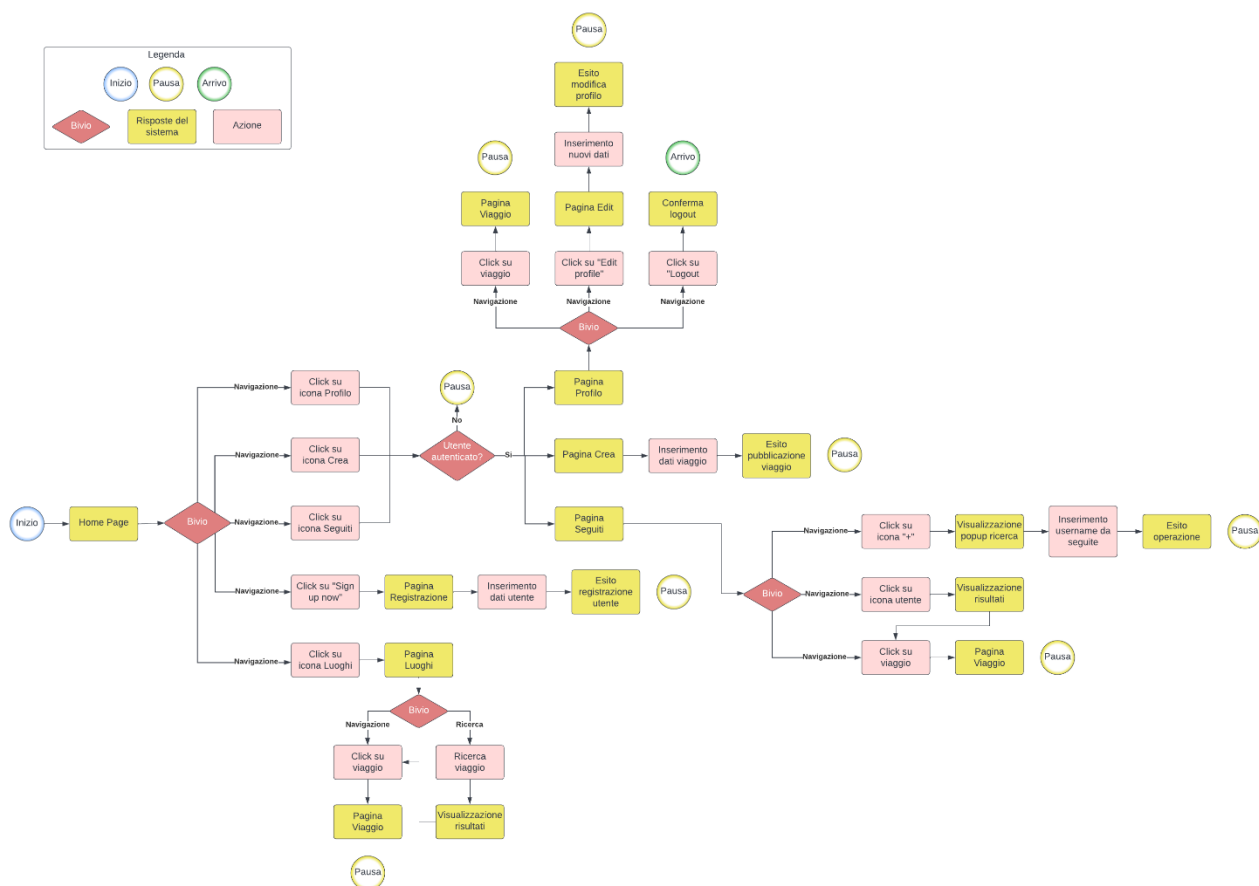
Qualora non si fosse registrati al sito bisogna recarsi nella pagina “Registrazione”; nella quale si andrà a compilare con i propri dati un breve form di registrazione.

Nella pagina “Profilo” è possibile avere una panoramica del proprio profilo e dei propri viaggi pubblicati. In questa pagina, qualora lo si desiderasse, si hanno inoltre le possibilità di modificare il proprio profilo e/o di effettuare il logout.

Nella pagina “Seguiti” è possibile vedere gli utenti che si seguono ed una “preview” dei viaggi pubblicati da essi. Cliccando sull'icona di un utente seguito è possibile visualizzare i viaggi che ha pubblicato. In questa pagina è inoltre possibile seguire un nuovo utente, andando a compilare un apposito form.

La pagina “Crea” è stata sviluppata per permettere all’utente di pubblicare un nuovo viaggio. In questa pagina è infatti possibile compilare un form con i dati del viaggio e delle relative tappe.

Nella pagina “Luoghi” viene mostrata una “preview” di alcuni viaggi che sono stati pubblicati dagli utenti. Qualora lo si volesse, cliccando su un viaggio, è possibile visualizzare interamente i viaggi. È inoltre possibile ricercare un viaggio specifico andandone a specificare il titolo ed un range di lunghezza.



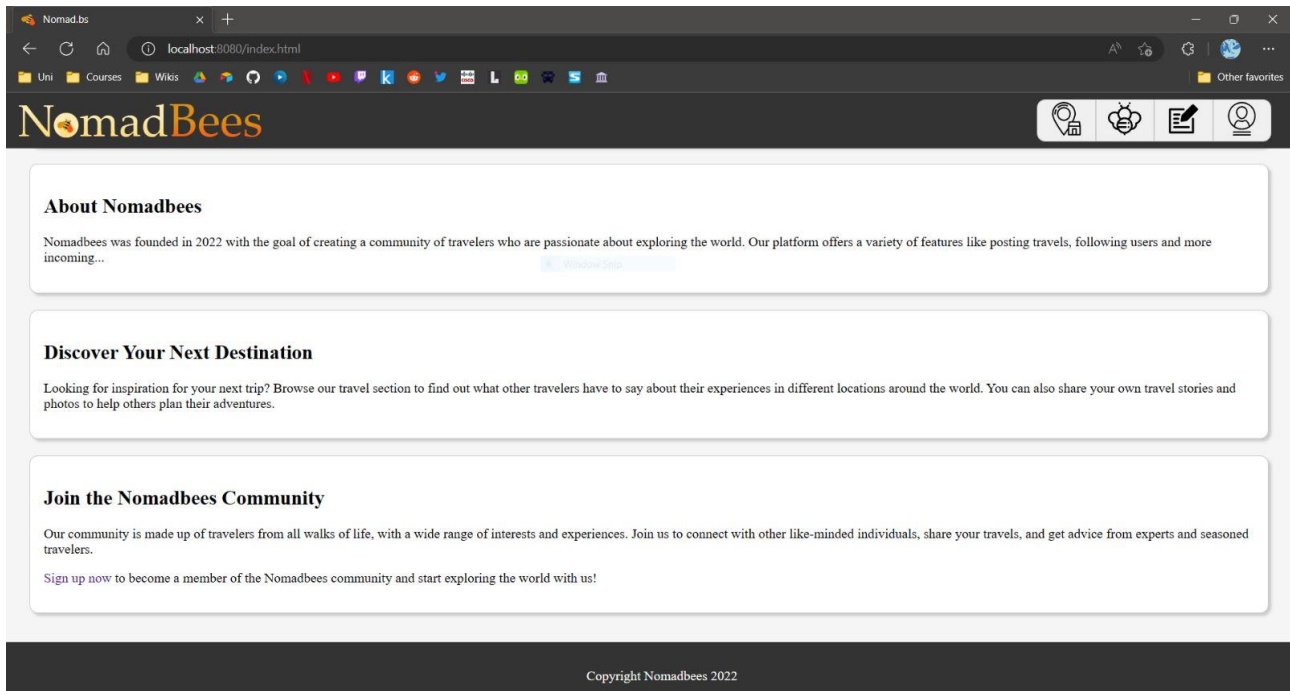
8. Front-end

L'implementazione di frontend sviluppata si pone di seguire le linee guida riportate nel documento 1; pertanto è stato scelto di utilizzare 5 distinti documenti .html relativi alle diverse pagine del sito

- Home
- Luoghi
- Seguiti
- Crea
- Profilo

A ciascuna di queste è associato un file .css dallo stesso nome, che ne gestisce lo stile, nonché alcuni script .js che aggiungono funzionalità sia grafiche, quali popup, sia funzionali, quali le chiamate al backend per ottenere i dati necessari al sito. Poiché non c'è stato tempo di testare il front-end con risoluzioni diverse, si consiglia di visualizzarlo su uno schermo 1920x1080 (si possono avere problemi in crea.html soprattutto legati alla mappa di Google).

Home

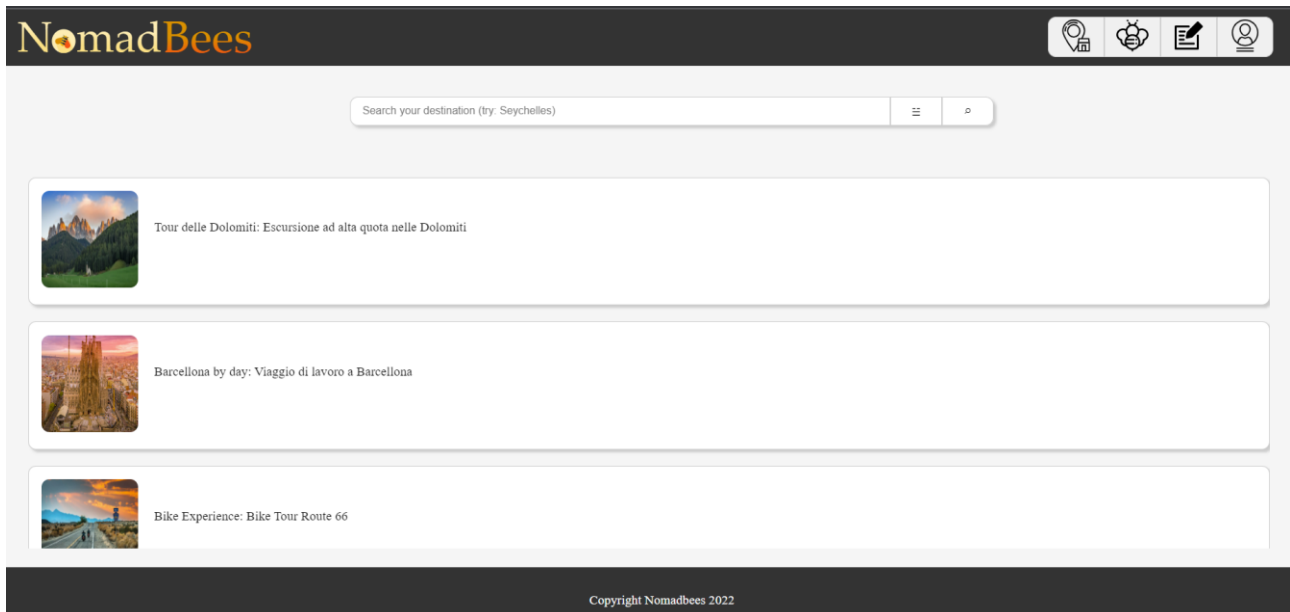


La pagina principale del sito consiste in una semplice lista delle proprie informazioni principali. Essa contiene tuttavia, in alto a destra, i pulsanti di navigazione principali, che permettono all'utente di spostarsi nelle 5 pagine e che rimangono anche nel layout delle rimanenti 4.

Notare come l'accesso alle pagine 'Seguiti' e 'Crea' sia negato all'utente qualora esso non sia autenticato; in caso un utente del sito che non abbia effettuato l'accesso clicchi su uno dei due pulsanti corrispondenti, esso sarà invece reindirizzato alla pagina 'Profilo', dove sarà presente un form per il login. Nell'ultimo paragrafo nella schermata iniziale è presente un link che ti porta alla pagina profilo.

Inoltre quando si accede alle diverse pagine è possibile tornare alla schermata Home cliccando sul logo del sito.

Luoghi

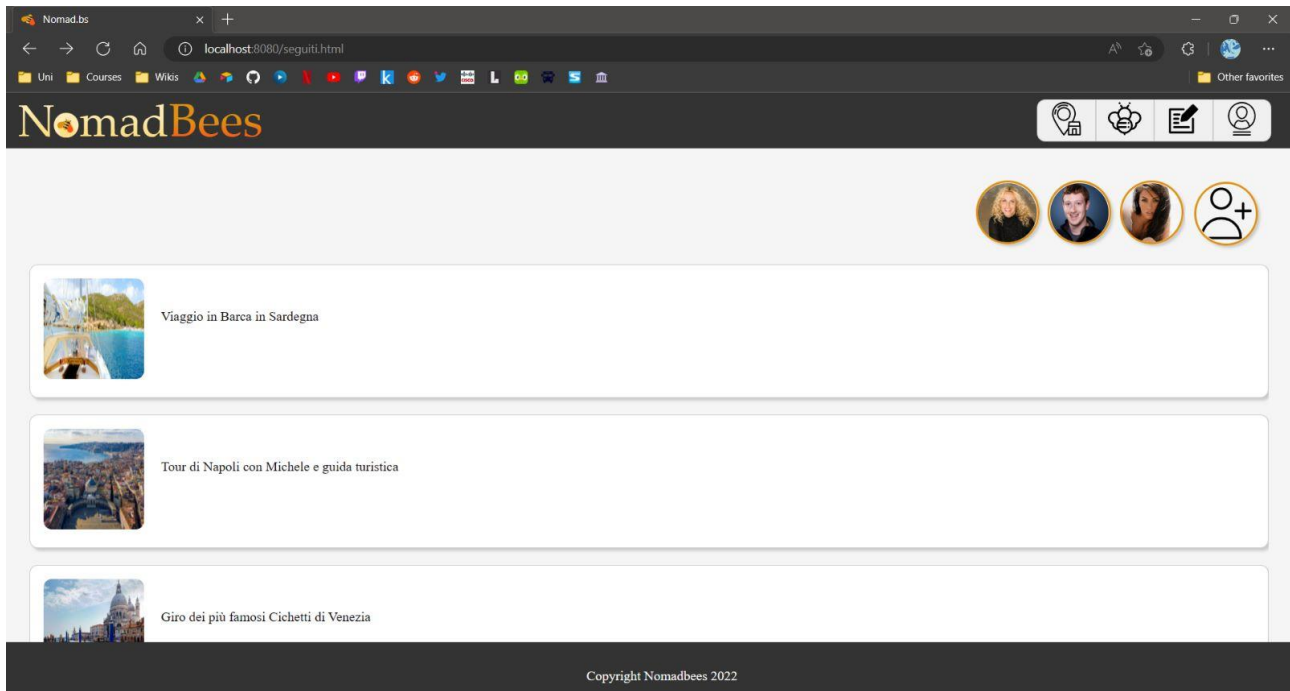


La pagina Luoghi presenta inizialmente 6 viaggi casuali estratti dal database accompagnati da titolo, descrizione e foto.

Utilizzando la barra di ricerca, che rimane perennemente visibile durante lo scorrimento della pagina, è possibile cercare il viaggio desiderato in base al titolo. Inoltre è possibile filtrare la ricerca in base alla lunghezza del viaggio, cliccando il pulsante apposito sulla barra di ricerca che mostrerà un popup dove selezionare un range di distanze.

Per maggiori informazioni sulla lista dei viaggi si faccia riferimento all'api `/searchViaggio` nella documentazione del backend.

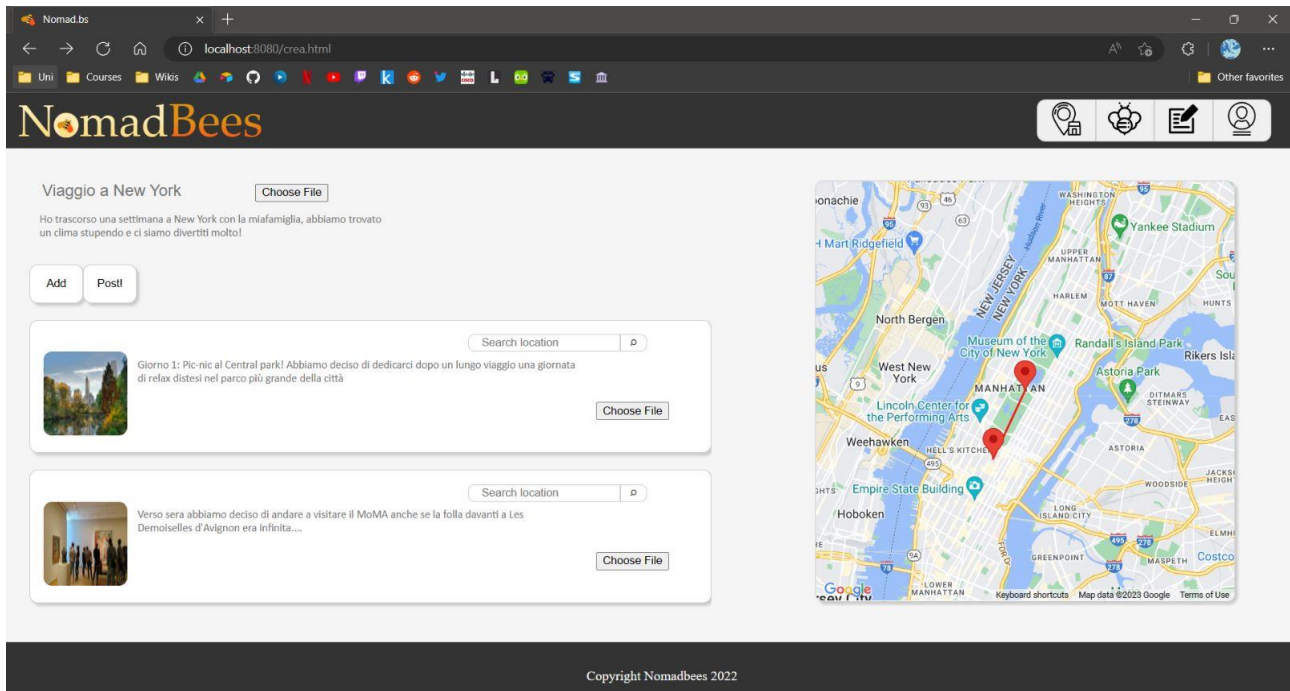
Seguiti



La pagina, come la precedente, presenta una lista di viaggi nello stesso formato, ma i viaggi in questa pagina appartengono esclusivamente ai profili seguiti dall'utente del sito; per ottenere questo risultato è utilizzata l'api `/getViaggiAmici`.

Questi ultimi compaiono inoltre come icona in alto nella pagina, in modo che l'utente, nell'implementazione finale, possa filtrare in base al profilo di cui desidera visualizzare i viaggi. Per ottenere dal database la lista di amici è utilizzata l'api `/getUsers`.

Crea



Questa pagina permette all'utente di creare il proprio post, costituito da un titolo e descrizione generali seguito da massimo 10 tappe con la loro foto e descrizione.

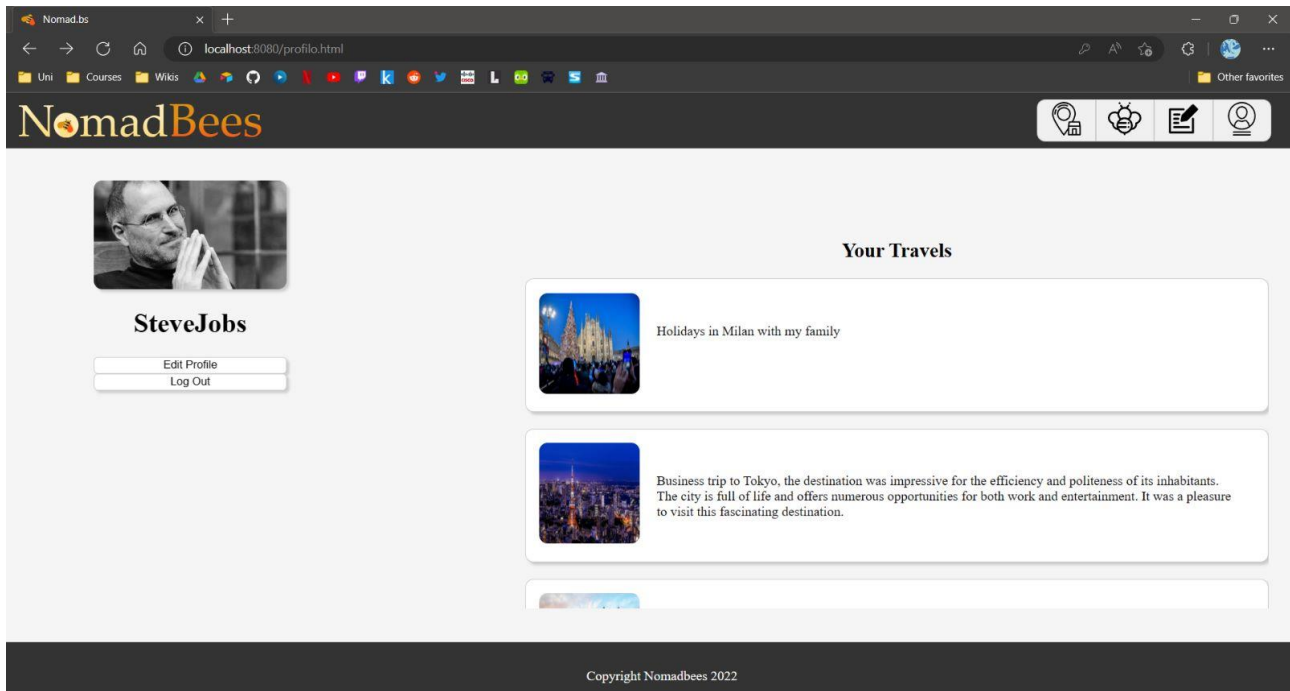
Nella pagina è presente, in alto a sinistra, un form per il titolo, la descrizione e l'immagine del viaggio, seguita dalla lista di tappe selezionate.

è possibile aggiungere fino ad un massimo di 10 tappe con il pulsante 'Add'; qui, nell'implementazione finale, sarà possibile cercare il luogo tramite ricerca grazie all'api di google maps, nonché caricare la propria foto con l'apposito pulsante. Nella stessa implementazione sarà infine possibile postare il viaggio finito utilizzando il pulsante 'Post!', che chiamerà l'api /newViaggio per salvarlo nel database.

A destra, nella pagina, è inoltre mostrata una mappa che presenta un'indicazione del percorso svolto, tappa per tappa, dall'utente.

Notare che, data lo stato incompleto della pagina, sono già state inserite due tappe di esempio sia sulla mappa che nella lista.

Profilo



Nella pagina 'Profilo' è possibile gestire tutto ciò che riguarda l'utente, ovvero:

- Login
- Modifica Profilo
- Logout

è stato deciso di tralasciare la funzione di signup, tramite l'api `/newUser`, e di invece inserire un account standard, con credenziali:

Username: SteveJobs

Password: Apple1955!

in modo da poter avere utenti seguiti e viaggi personali già inseriti.

Questi ultimi sono presentati a destra nella pagina, qualora si effettui il login.

9. Deployment

L'implementazione del sito è pubblicata su github: <https://github.com/SE-T37/Back-end>.

La repository contiene nel main una versione senza i node_modules, se necessari, una volta clonata la repo si può fare un checkout al branch with_modules per avere la versione con i moduli importati.

1. Una volta clonata la repository è **necessario creare un file .env** con le seguenti credenziali (presenti anche nel file readme.md della repository):

```
SUPER_SECRET='NomadBees-Key'
```

```
DB_HOST=localhost
```

```
DB_USER=nomadbees
```

```
DB_PASS=IGWcyG7y372jOu14
```

```
PORT = 8080
```

```
MONGODB_URI="mongodb+srv://nomadbees:IGWcyG7y372jOu14@cluster0.r7axnea.mongodb.net/NomadBeesDB"
```

2. Per avviare il server utilizzare il comando **npm start**
3. Aprire `http://localhost:8080` sul browser