**Progetto Ingegneria Software**

*NomadBees*

Gruppo T-37:

Matteo Pontalti, Matteo Bregola, Riccardo Libanora.

Sviluppo Applicazione

Contenuti

[Scopo del Documento 3](#_Toc123295596)

[1. Resource Extraction Diagram 4](#_Toc123295597)

[2. Resource Modelling Diagram 6](#_Toc123295598)

[3. Struttura del Progetto 7](#_Toc123295599)

[Servizi e piattaforme 7](#_Toc123295600)

[Struttura 7](#_Toc123295601)

[4. Implementazione API 9](#_Toc123295602)

[Introduzione 10](#_Toc123295603)

[1. Authenticate 11](#_Toc123295604)

[2. NewUser 13](#_Toc123295605)

[3. SearchUser 15](#_Toc123295606)

[4. EditUser 17](#_Toc123295607)

[5. FollowUser 18](#_Toc123295608)

[6. NewViaggio 20](#_Toc123295609)

[7. SearchViaggio 22](#_Toc123295610)

[5. Documentazione 23](#_Toc123295611)

[6. Testing 25](#_Toc123295612)

[Swagger 25](#_Toc123295613)

[Postman 28](#_Toc123295614)

[7. User Flow Diagram 28](#_Toc123295615)

[8. Front-end 29](#_Toc123295616)

[9. Deployment 30](#_Toc123295617)

# 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 coincisa 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.

# 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 contente 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.

Diagram

Description automatically generated

# 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.

A picture containing text, receipt, screenshot

Description automatically generated

# 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 E**xpress**. **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. Text

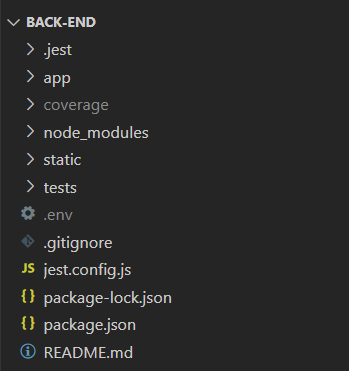
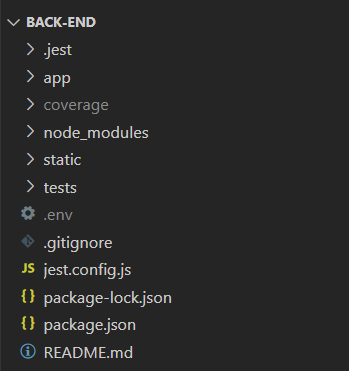
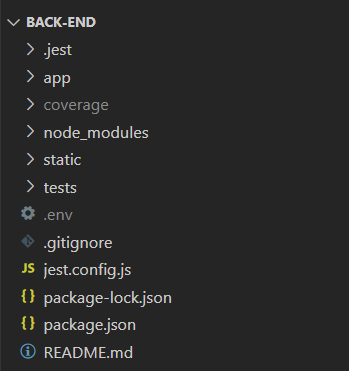
   Description automatically generated with medium confidence**App** contiene l’implementazione e gestione delle API descritte nelle sezioni precedenti (con l’aggiunta di tre API di “supporto” per il front-end), il file che gestisce la connessione al server ed il file swagger per la documentazione.
2. Text

   Description automatically generated**Static** contiene gli script html, css e javscript che gestiscono il frontend.
3. Text

   Description automatically generated**Tests** contiene i test effettuati grazie alla libreria Jest.

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.



*(Link per la repository su github:* [*https://github.com/SE-T37/Back-end*](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).*

# 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
* newViagio
* 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) e 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.*

1. 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.

## 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 visible dai file nella cartella routes.

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

## 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 no n 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.*

## 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

            };

        });

        return res.status(200).json(users);

}

Se invece uno o più utenti corrispondono a quello ricercato allora vengono ritornati gli username 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.*

## 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.*

## 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.*

## 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.*

## 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

            };

        });

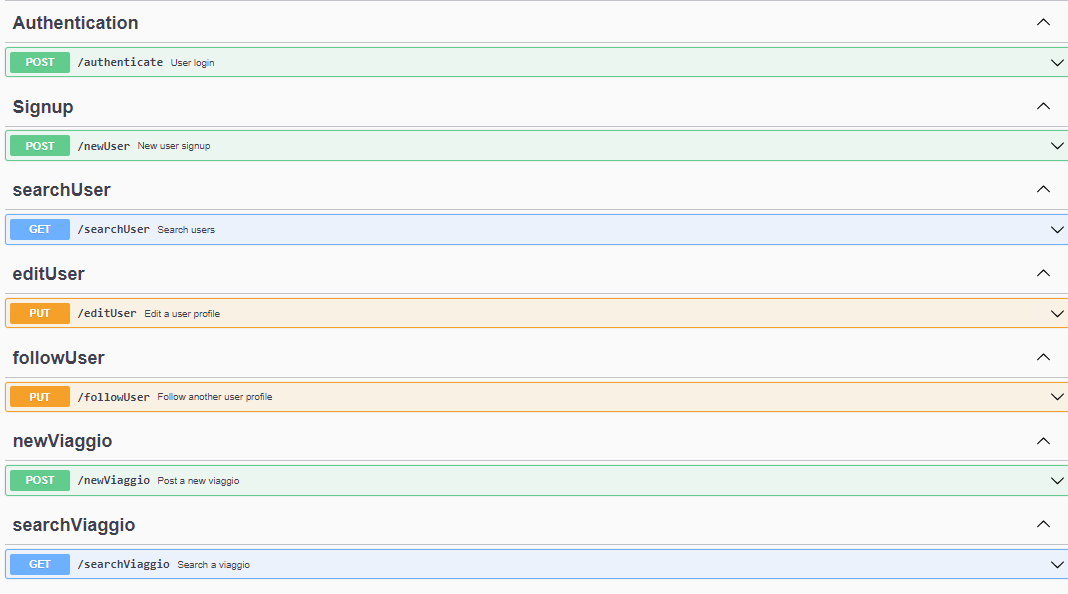
        return res.status(200).json(viaggi);

    }

Se sono stati trovati viaggi allora vengono ritornati.

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

# Documentazione

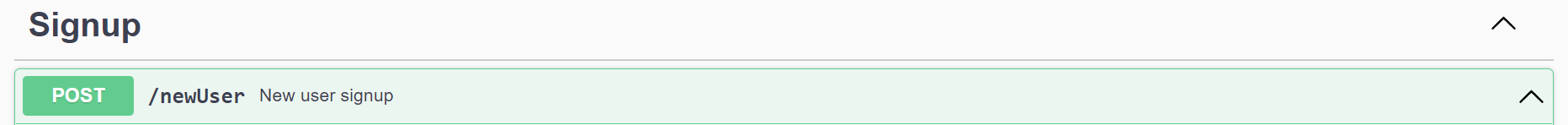
La documentazione delle API è stata realizzata tramite la libreria Swagger. Il file contente il codice che la implementa è denominato come *swagger.json* mentre la documentazione è accessibile al URL *api-docs (es in locale:* [*http://localhost:8080/api-docs*](http://localhost:8080/api-docs)*).*

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.



L’API che permette di creare un profilo all’interno del sito è di tipo POST ed è accessibile tramite l’endpoint “/newUser”.

Graphical user interface, application

Description automatically generated

Background pattern, rectangle

Description automatically generatedI 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.

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).

# Testing

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

## Swagger

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%**
* Graphical user interface

  Description automatically generated with medium confidenceFunzioni analizzate: **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 “/autenticate” 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 del’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.

### **Iscrizione al sito**

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

Text

Description automatically generated

Risposta del sito A screenshot of a computer

Description automatically generated with medium confidence

Graphical user interface, text, application, email

Description automatically generatedControllo salvataggio nel database:

### **Autenticazione**

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

### **Pubblicazione di un viaggio**

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generatedRisposta (parziale):

Graphical user interface, text, application

Description automatically generatedControllo presenza viaggio nell’utente:

Controllo presenza viaggio:

Text

Description automatically generated

### **Ricerca di un viaggio**

Graphical user interface, text, application

Description automatically generated

Risposta (parziale):

Text

Description automatically generated

### **Ricerca di un utente**

A screenshot of a computer screen

Description automatically generated with low confidence

Shape

Description automatically generated with medium confidence

### **Seguire un utente**

A screenshot of a computer

Description automatically generated with medium confidence

A picture containing shape

Description automatically generated

Controllo salvataggio: tra i seguiti

Graphical user interface, text, application

Description automatically generated

### **Modifica Profilo**

A screenshot of a computer

Description automatically generated with low confidence

Text

Description automatically generated

Controllo avvenuta modifica password:

Graphical user interface, text, application, email

Description automatically generated

# User Flow Diagram

# Front-end

Si può provare a loggarsi come diversi utenti già presenti nel sito, tuttavia si consiglia di utilizzare:

SteveJobs password: Apple1955!

# Deployment