

Progetto Ingegneria Software

NomadBees

Gruppo T-37:

Matteo Pontalti, Matteo Bregola, Riccardo
Libanora.

Sviluppo Applicazione

Contenuti

Scopo del Documento	3
---------------------------	---

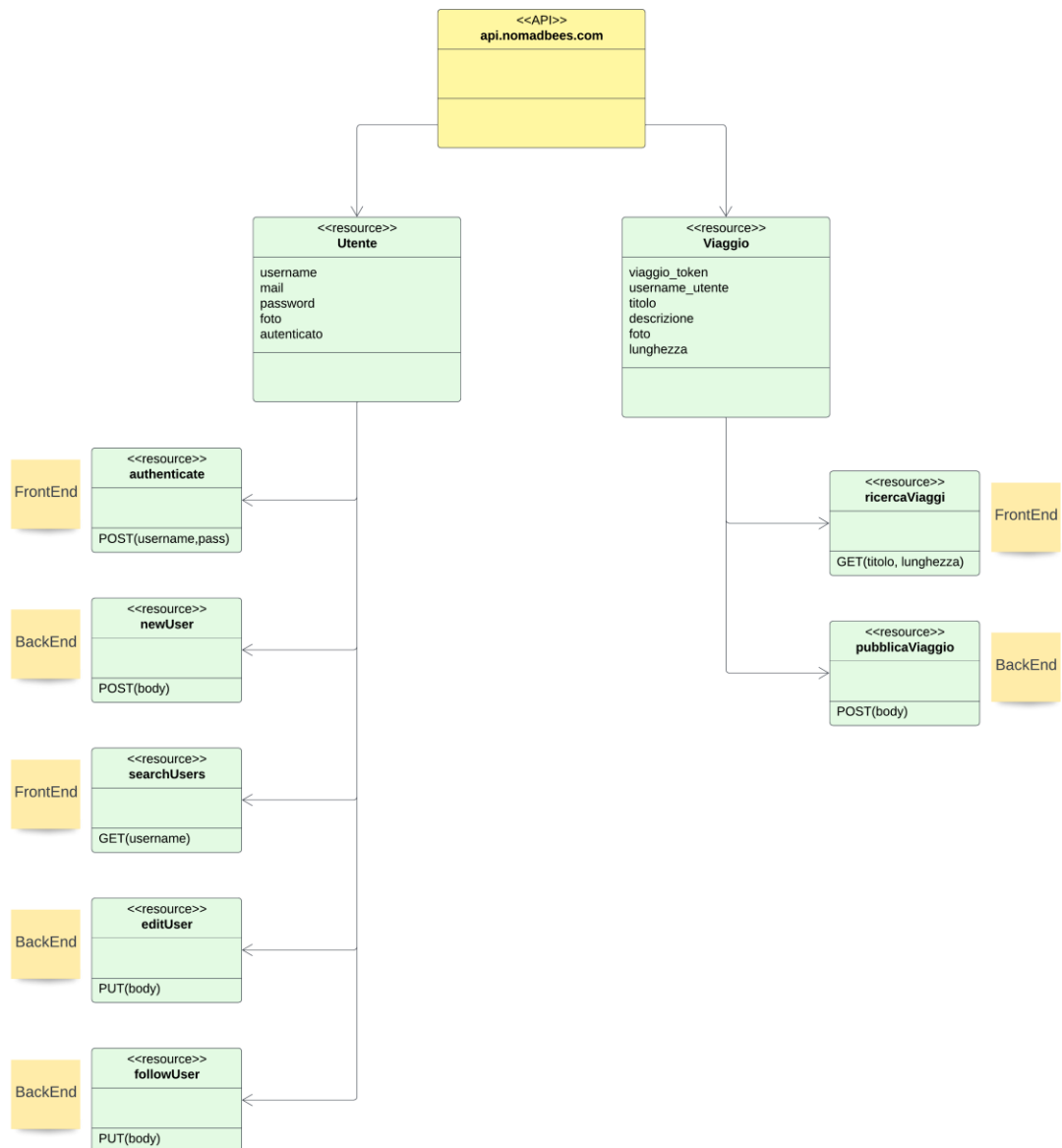
Scopo del Documento

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.

2. 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 RESTful. Esso rappresenta 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.



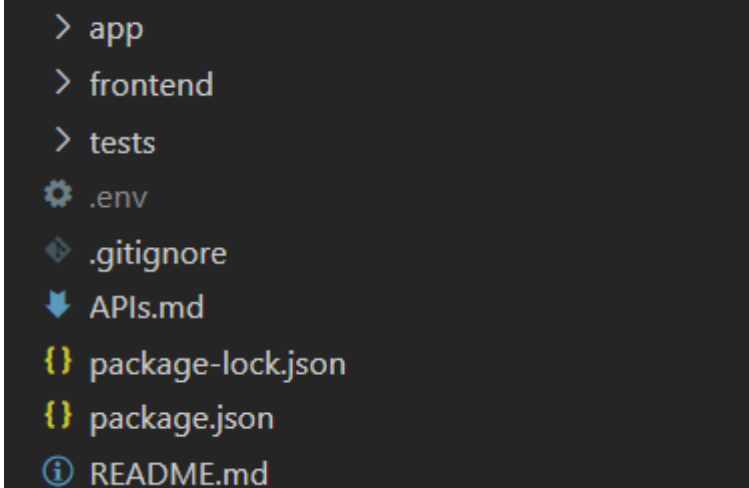
3. Resource Modeling Diagram

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.

INSERIRE QUI IL DIAGRAMMA

4. Struttura del Progetto

Si è deciso di suddividere il progetto in tre cartelle di lavoro: app contiene l'implementazione delle API descritte nelle sezioni precedenti, frontend contiene gli script html, css e javascript che gestiscono il frontend ed una cartella tests che contiene i test effettuati grazie a Jest. Nel file .env vi sono le informazioni private per la gestione del database, dei token e del deployment. All'interno dei file json vi sono le informazioni sulle dipendenze delle librerie usate per lo sviluppo ed altre informazioni sul progetto Node.js come autori script di avvio e build. Il file APIs contiene gli URI delle API sviluppate.



```
> app
> frontend
> tests
⚙ .env
💎 .gitignore
⬇ APIs.md
{} package-lock.json
{} package.json
📘 README.md
```

5. 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

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>

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,
      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>

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

Risposta attesa: creazione profilo.

```
const newUser = (req, res, next) => {;  
  User.findOne({ username: req.body.username}, (err, data) => {
```

La prima azione eseguita è il controllo della presenza di un utente già registrato con lo stesso username di quello che si vuole creare.

```
if(!data){  
  const newUser = new User({  
    username: req.body.username,  
    mail: req.body.mail,  
    password: req.body.password,  
    foto: req.body.foto,  
  })  
  newUser.save( (err,data) => {  
    if (err) return res.status(500).json({Error: err});  
    else  
      return res.status(201).json(data);  
  })  
}
```

Se 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{  
    if (err) return res.status(500).json('Something went wrong,  
please try again. ${err}');  
    return res.status(400).json({message: "User already  
exists"});  
}
```

Se l'username è già presente viene notificato il problema mentre se si occorre in altri problemi durante l'operazione viene ritornato il codice 500.

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

3. SearchUser

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

Parametri richiesta: **username**.

Risposta attesa: Username utenti contenenti la stringa ricercata.

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

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

```
if(!users){  
    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.

4. EditUser

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

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>

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){
  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);
  return res.status(200).json(user_richiedente.seguiti);
}
```

Se l'utente esiste nel database e non è tra quelli seguiti dal richiedente allora viene aggiunto alla lista, viene ritornato il codice 200 e la lista degli utenti seguiti.

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

6. NewViaggio

DA COMPLETARE

7. SearchViaggio

DA COMPLETARE

6. Documentazione

Nota: non è possibile utilizzare swagger direttamente per testare le API che richiedono l'autenticazione

7. Testing

8. User Flow Diagram

9. Front-end

10. Deployment