



Dipartimento di Ingegneria e Scienza dell'Informazione  
Anno formativo 2022/2023

Percorso di studio: Scienze e Tecnologie Informatiche  
Attività didattica: Ingegneria del Software



Progetto: “All About Trento”  
Titolo del documento: Sviluppo Applicazione  
Gruppo: T07

Allievi: Manuel Vettori, Emanuele Nardelli, Zappacosta Lorenzo

# INDICE

Scopo del Documento	3
1. User Flows	3
2. Application Implementation and Documentation	5
2.1 Project Structure	5
2.2 Project Dependencies	7
2.3 Project Data or DB	7
2.4 Project APIs	9
2.4.1 Resources Extraction from the Class Diagram	9
2.4.2 Resources Models	10
2.5 Sviluppo API	12
2.5.1 Create User (Registrazione)	12
2.5.2 Login	12
2.5.3 Inserimento Recensione	13
2.5.4 Inserimento POI	13
2.5.5 Eliminazione POI	14
2.5.6 Modifica POI	14
3. API Documentation	15
4. FrontEnd Implementation	16
5. GitHub Repository and Deployment Info	19
6. Testing	22

## SCOPO DEL DOCUMENTO

Attraverso questo documento si riportano tutte le informazioni che sono necessarie per lo sviluppo di una parte dell' applicazione All About Trento.

In particolare, sono presenti tutti gli strumenti necessari per realizzare i servizi di gestione dei punti d'interesse (POI) dell' applicazione All About Trento.

Partendo dalla descrizione degli *User Flows* legate al ruolo del 'Gestore' (di seguito denominato 'G') dell'applicazione, il documento prosegue con la presentazione delle API necessarie (tramite l'*API Model* e il Modello delle risorse) per poter visualizzare, inserire e modificare sia i punti d'interesse che i vari utenti necessari all'applicazione All About Trento.

Le APIs fornite per interagire con l'applicazione saranno accompagnate da una descrizione in linguaggio naturale per aiutare la comprensione del loro utilizzo.

Nel documento è stata riportata la documentazione, i test effettuati e una descrizione delle funzionalità fornite per ogni API realizzata.

## 1. USER FLOWS

In questo capitolo del documento di sviluppo vengono mostrati e descritti gli "user flows" per quanto riguarda il ruolo del 'Gestore' della nostra applicazione.

La figura sottostante (Figura 1) descrive l'*user flow* relativo alla gestione delle informazioni inerenti ai punti d'interesse (POI) e ai vari utenti della nostra applicazione.

Il Gestore 'G' può, in ogni momento, accedere al sistema e consultare le liste riguardanti i vari punti d'interesse e gli utenti loggati al sito.

Il Gestore può inserire nuovi POI e utenti in caso di necessità dopo aver compilato il form di inserimento.

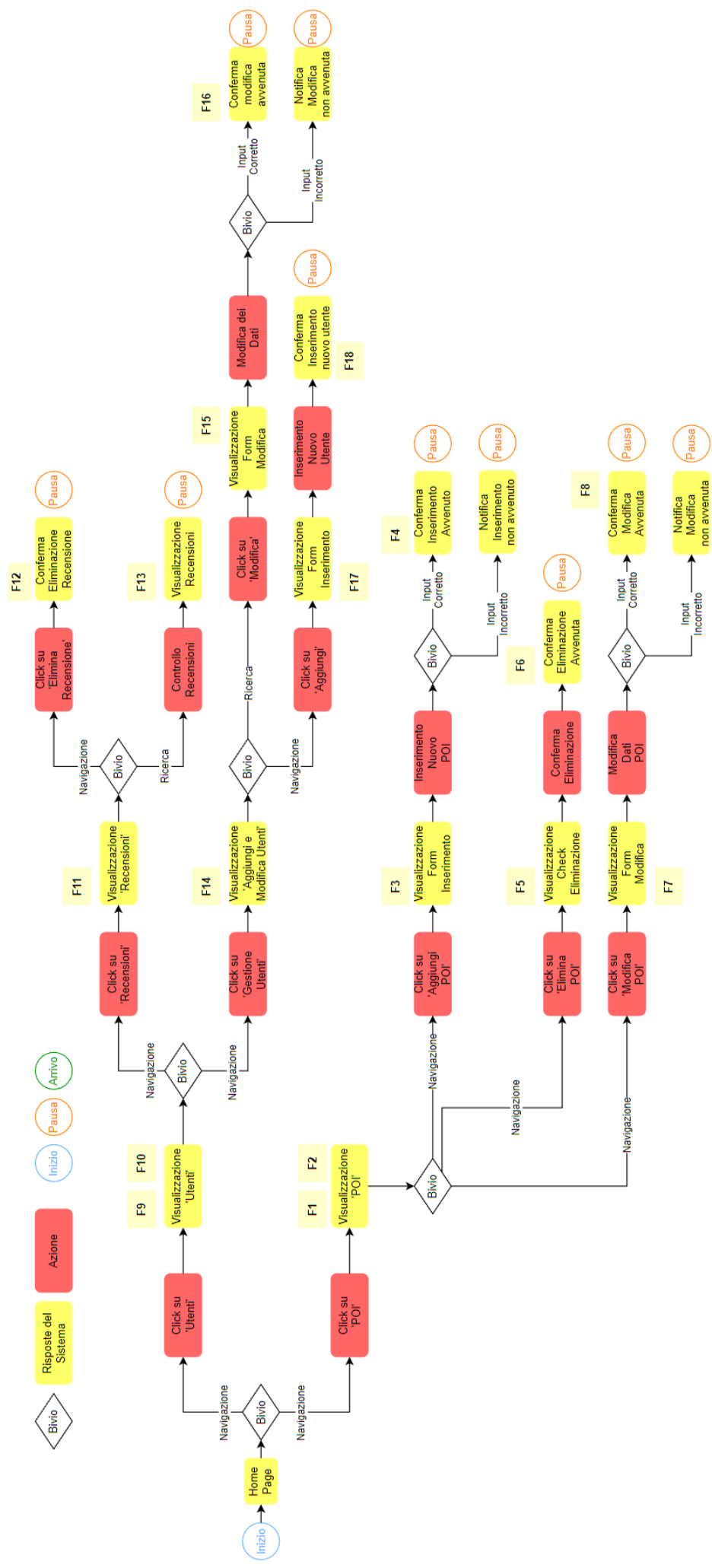
Per lo stesso principio, il Gestore può anche modificare o addirittura eliminare i POI.

Oltre a queste funzionalità, il Gestore si occupa di gestire anche la parte delle recensioni che vengono effettuate dagli utenti.

All'interno della Figura 1 presentiamo anche la relazione tra le varie azioni che l'utente 'Gestore' può fare e le features descritte in Sezione 2.

Allo scopo di poter apprendere meglio l'immagine e il suo contenuto è stata realizzata una legenda che descrive i simboli usati nell' user flow ossia presentata in Figura 1.

**Figura 1. User Flow per il Gestore 'G'**



## 2. APPLICATION IMPLEMENTATION AND DOCUMENTATION

Nei capitoli precedenti abbiamo identificato le varie features che devono essere implementate per la nostra applicazione con un'idea di come l'utente finale può utilizzarle nel suo flusso applicativo.

Queste funzionalità seguono un preciso flusso definito dallo *User Flow* e comporta diversi comportamenti secondo il quale il sistema deve adattarsi a rispondere in base all'azione eseguita

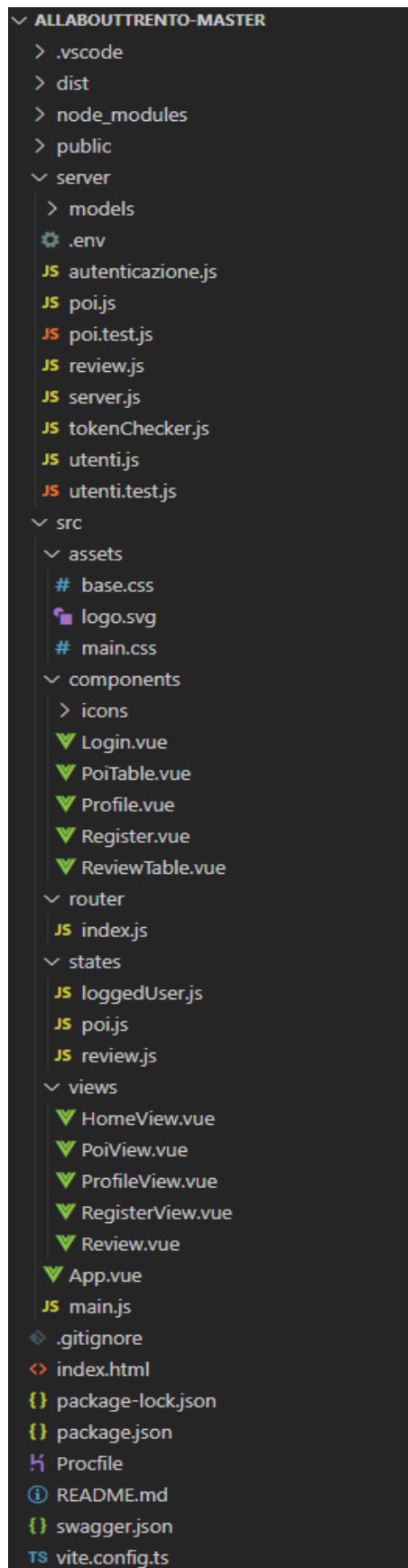
L'applicazione è stata sviluppata utilizzando *NodeJS* e *VueJS*.

Per la gestione dei dati abbiamo utilizzato *MongoDB*.

### 2.1 PROJECT STRUCTURE

La struttura del progetto è suddivisa in 2 directory principali (vedi Figura 2):

- directory "**SERVER**": all'interno della directory vengono definiti i file per la realizzazione del back-end del sito.
  - nella sotto-directory "**MODELS**" vengono definiti i modelli di Mongoose per salvare i dati all'interno di MongoDB.
- directory "**SRC**": all'interno della directory vengono definiti i file per la realizzazione del front-end del sito.
  - nella sotto-directory "**COMPONENTS**" vengono implementate i file che sono presenti all'interno della sotto-directory views.
  - nella sotto-directory "**ROUTER**" vengono definiti i percorsi alla quale l'applicazione dovrà indirizzare.
  - nella sotto-directory "**STATES**" vengono definite alcune funzioni che integrano i file che sono presenti nella sotto-directory components.
  - nella sotto-directory "**VIEWS**" vengono definite le viste alle quali l'applicazione si interfaccia.



**Figura 2. Struttura Codice Sorgente**

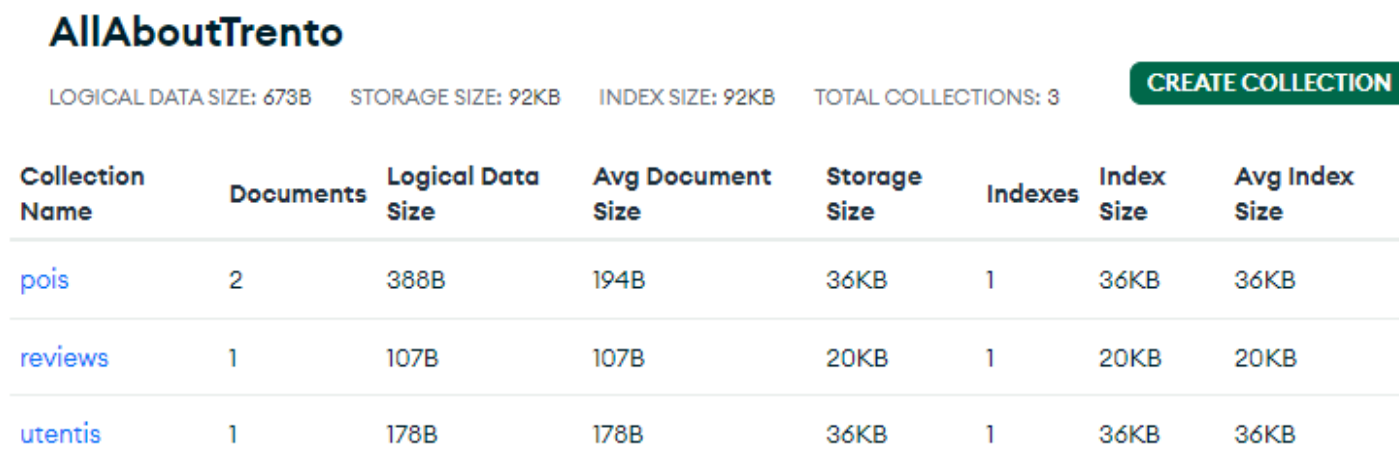
## 2.2 PROJECT DEPENDENCIES

I seguenti moduli *Node* sono stati utilizzati e aggiunti al file *Package.json*:

- MongoDB
- Cors
- Mongoose
- Dotenv
- Express
- Multer
- Supertest
- JsonWebToken
- Vue
- Vue-Router

## 2.3 PROJECT DATA OR DB

Per la gestione dei dati del sito sono state definite 3 principali strutture dati su cui operare: una collezione di “POI”, una collezione di “Utenti” e una collezione di “Recensioni” (vedi sotto Figura 3).



Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
<a href="#">pois</a>	2	388B	194B	36KB	1	36KB	36KB
<a href="#">reviews</a>	1	107B	107B	20KB	1	20KB	20KB
<a href="#">utentis</a>	1	178B	178B	36KB	1	36KB	36KB

**Figura 3. Collezione Dati usati nell'applicazione**

Rappresentiamo, dunque, gli utenti, i punti d'interesse e le recensioni definendo i tipi di dato mostrati nella Figura 4, nella Figura 5 e nella Figura 6:

```
_id: ObjectId('6399a105490ca809bc399b20')
nome: "Federico"
cognome: "Verdi"
email: "federico.verdi@studenti.unitn.it"
numTelefono: "3385695622"
password: "Federico.Verdi"
ruolo: "gestore"
__v: 0

_id: ObjectId('6395e28933e65f41c8bda54b')
nome: "Marco"
cognome: "Rossi"
email: "marco.rossi@studenti.unitn.it"
numTelefono: "3253565185"
password: "Marco.Rossi"
ruolo: "utente"
__v: 0
```

**Figura 4. Tipo di Dato “Utente”**

```
_id: ObjectId('6394ad8a19897b010dabcd67')
nome: "duomo"
tipologia: "monumento"
descrizione: "Duomo di trento"
posizione: "piazza duomo"
stato: "aperto"
orari_apertura: "10-19"
immagini: "https://upload.wikimedia.org/wikipedia/commons/9/98/Trento-Piazza_del_..."
__v: 0
```

**Figura 5. Tipo di Dato “POI”**

```
_id: ObjectId('63999c44055d491123a244cb')
titolo: "Recensione cattedrale San Vigilio"
descrizione: "Al momento della visita era in restauro l'abside (visto molti anni fa)..."
valutazione: 4
__v: 0
```

**Figura 6. Tipo di Dato “Recensione”**



## 2.4 PROJECT APIs

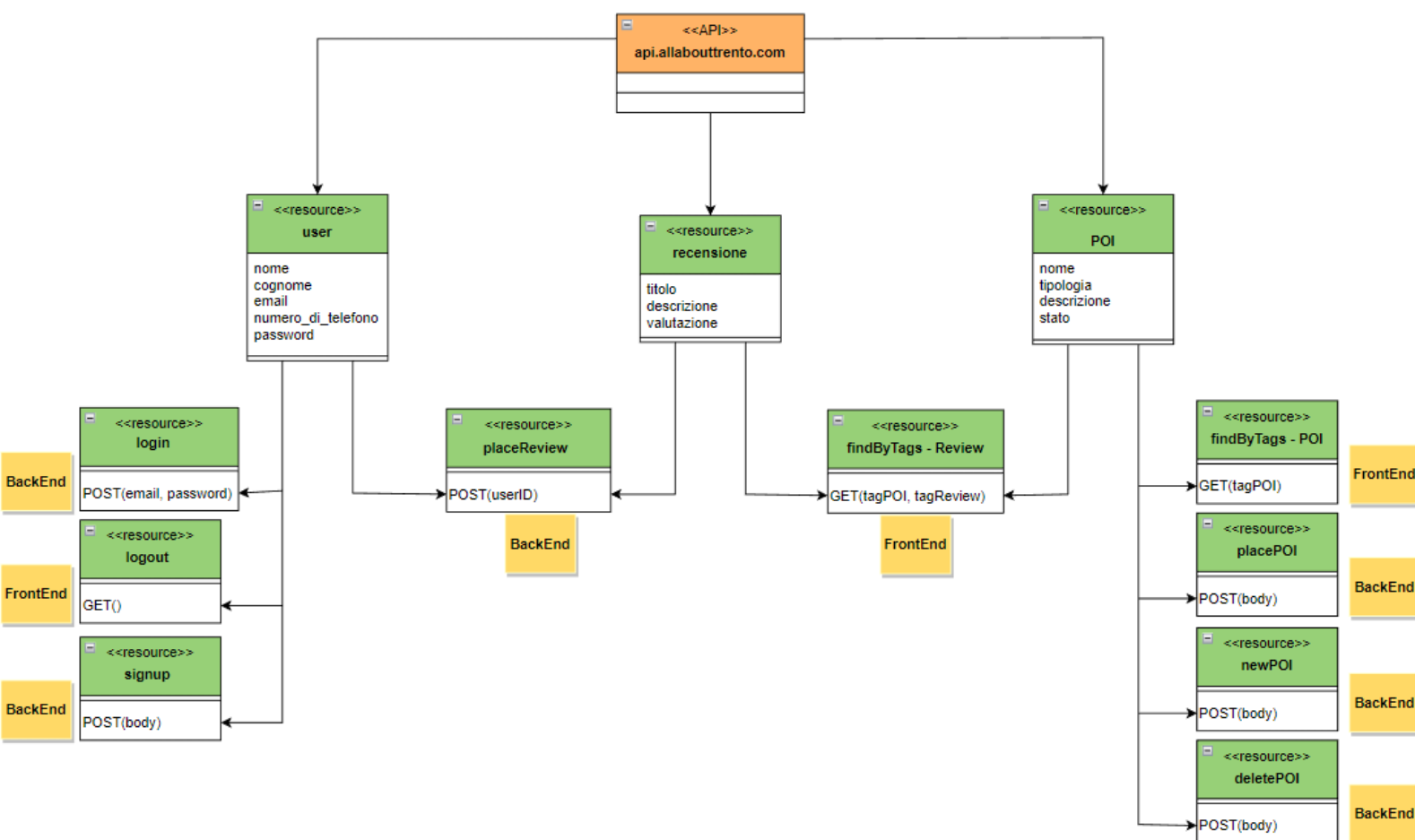
### 2.4.1 RESOURCES EXTRACTION FROM THE CLASS DIAGRAM

Dal diagramma delle classi, ricaviamo 3 risorse: lo *'user'*, la *'recensione'* e il *'POI'*.

Ognuna di queste risorse ha i propri parametri, che vanno a definire nello specifico quella determinata risorsa e che la rappresenteranno all'interno di una realtà.

Da queste risorse otteniamo, inoltre, le funzionalità collegate ad esse, definite anche loro come risorse.

Si noti che le risorse *'placeReview'* e *'findByTags - Review'* sono collegate da più risorse; entrambe condividono la risorsa *'recensione'*, la prima utilizza anche la risorsa *'user'* mentre la seconda è associata alla risorsa *'POI'*.



**Figura 7. Diagramma delle Risorse**

### 2.4.2 RESOURCE MODELS

Una volta estratte le risorse dal diagramma delle classi, andiamo a costruire il *Resources Model*.

Dalle 3 risorse principali estratte ('*user*', '*recensione*' e '*POI*') creiamo le API collegate ad esse.

Ogni API rappresenta una funzionalità collegata a quella specifica risorsa.

Una volta associate alla loro risorsa, le API creano ulteriori risorse che rappresentano cosa fa nello specifico quella specifica funzione.

Dallo schema si possono notare 2 risorse specifiche: '*ERROR*' e '*UNAUTHORIZED*'.

Queste 2 risorse sono associate a quasi tutte le API in quanto rappresentano delle determinate situazioni (in questo caso, errore o azione non autorizzata) che possono accadere durante l'utilizzo di una funzione.

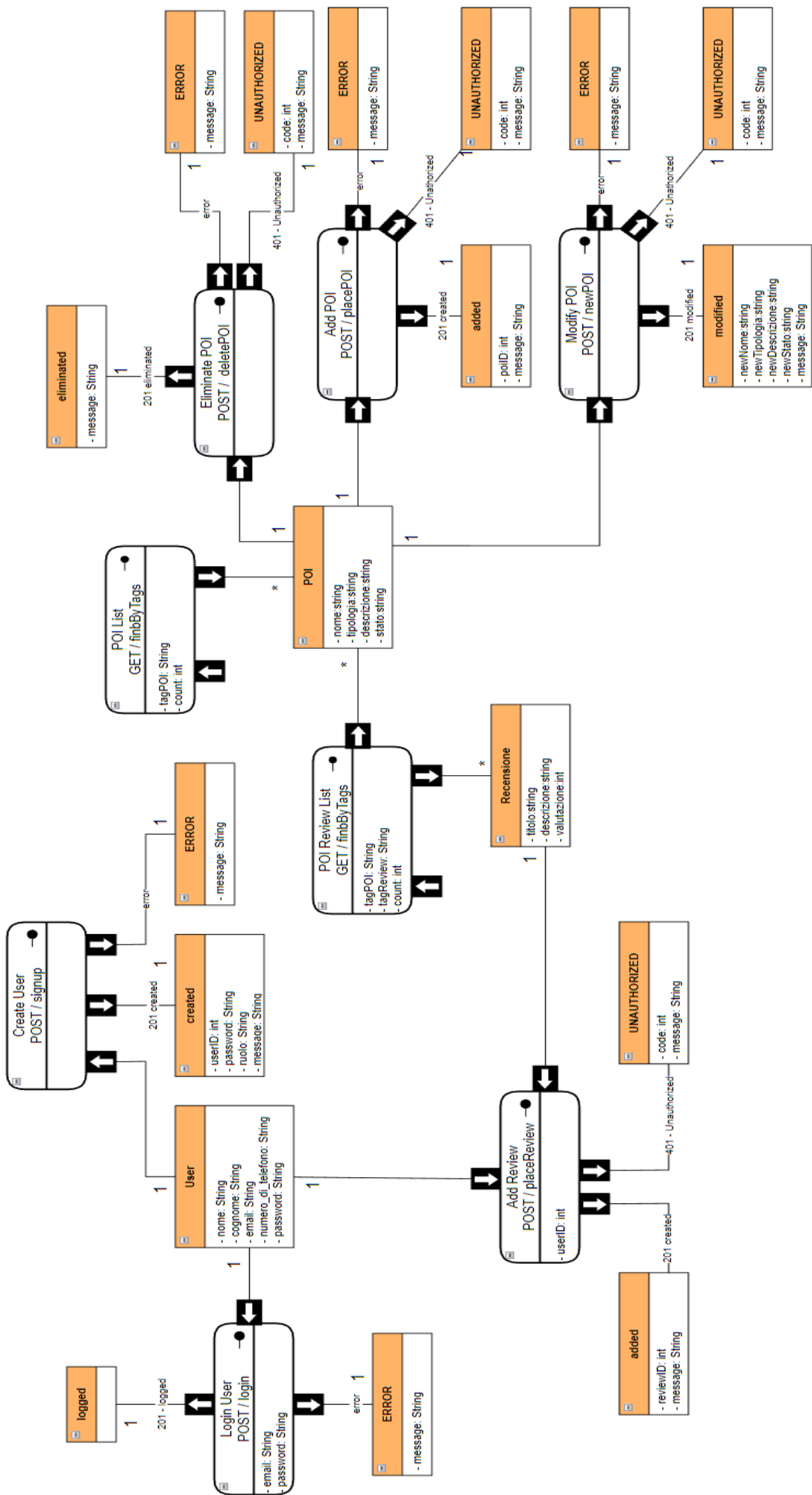
All'interno del nostro schema, vengono presentate le seguenti API:

- **CREATE USER** (*signup*)
- **LOGIN USER** (*login*)
- **POI LIST** (*findByTags - POI*)
- **ADD POI** (*placePOI*)
- **MODIFY POI** (*newPOI*)
- **ELIMINATE POI** (*deletePOI*)
- **POI REVIEW LIST** (*findByTags - Review*)
- **ADD REVIEW** (*placeReview*)

Come spiegato nel punto precedente (vedi 2.4.1 *RESOURCES EXTRACTION FROM THE CLASS DIAGRAM*), le risorse '*placeReview*' e '*findByTags - Review*' condividevano più risorse.

Questo è stato riportato anche durante la creazione del *Resources Model*; difatti, alle 2 API definite per queste risorse, vengono associate più risorse che servono al corretto funzionamento della funzione.

**Figura 8. Diagramma delle API (Resources Model)**



## 2.5 SVILUPPO API

Vengono elencate le varie API implementate per soddisfare i vari requisiti del sistema dal punto di vista dell'utente.

### 2.5.1 CREATE USER (Registrazione)

Questa API permette al sistema di registrare nuovi utenti all'interno del database.

Viene creato un nuovo oggetto di tipo 'Utente' e, tramite il form di inserimento, vengono inserite tutti i dati riguardanti quello specifico utente.

Ritorna un messaggio di avvenuta registrazione altrimenti un messaggio di errore qualora qualcosa fosse andato storto.

```
//post a new utente
router.post('/', async (req, res) => {
  let utenti = new Utenti({
    nome : req.body.nome,
    cognome : req.body.cognome,
    email : req.body.email,
    numTelefono : req.body.numTelefono,
    password : req.body.password,
    ruolo : req.body.ruolo
  });

  utenti = await utenti.save();
  let utentiId = utenti.id;
  console.log('Utente saved successfully');
  res.location("utentis/" + utentiId).status(201).send();
});
```

**Figura 9. Inserimento di un nuovo Utente**

### 2.5.2 LOGIN

Questa API viene utilizzata per gestire il Login degli utenti all'interno del sito.

Viene effettuato il controllo sulle credenziali inserite (email e password), ritorna lo status di 'logged' se le credenziali sono corrette altrimenti un messaggio di errore qualora qualcosa andasse storto.

```
function login() {
  fetch(HOST+"/autenticazione", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ email: email.value, password: password.value }),
  })
  .then((resp) => resp.json())
  .then(function (data) {
    setLoggedUser(data);
    emit("login", loggedUser);
    return;
  })
  .catch((error) => console.error(error));
}

function logout() {
  clearLoggedUser();
}
```

**Figura 10. Login (e Logout)**

### 2.5.3 INSERIMENTO RECENSIONE

L'API qui riportata permette l'inserimento di una recensione da parte dell'utente registrato.

Viene creato un nuovo oggetto di tipo 'Review' e vengono salvate all'interno del database tutte le informazioni immesse nel form di inserimento.

Ritorna un messaggio di avvenuto inserimento altrimenti un messaggio di errore qualora qualcosa fosse andato storto.

```
//post a new review
router.post('/', async (req, res) => {
  let review = new Review({
    titolo : req.body.titolo,
    descrizione : req.body.descrizione,
    valutazione : req.body.valutazione
  });

  review = await review.save();
  let reviewId = review.id;
  console.log('Review saved successfully');
  res.location("reviews/" + reviewId).status(201).send();
});
```

*Figura 11. Inserimento di una Recensione*

### 2.5.4 INSERIMENTO POI

Questa API permette l'inserimento di un nuovo POI all'interno del sistema.

Viene creato un nuovo oggetto di tipo 'POI' e vengono salvate all'interno del database tutte le informazioni immesse nel form di inserimento.

Ritorna un messaggio di avvenuto inserimento altrimenti un messaggio di errore qualora qualcosa fosse andato storto.

```
//post a new poi
router.post('/', async (req, res) => {
  let poi = new Poi({
    nome : req.body.nome,
    tipologia : req.body.tipologia,
    descrizione : req.body.descrizione,
    posizione: req.body.posizione,
    stato : req.body.stato,
    orari_apertura : req.body.orari_apertura,
    immagine : req.body.immagine
  });

  poi = await poi.save();
  let poiId = poi.id;
  console.log('Poi saved successfully');
  res.location("pois/" + poiId).status(201).send();
});
```

*Figura 12. Inserimento di un POI*

### 2.5.5 ELIMINAZIONE POI

L'API riportata rappresenta la funzione di eliminazione di un POI all'interno del sistema.

Viene richiesto quale POI eliminare e, qualora il POI esistesse, viene cancellato e ritorna un messaggio di avvenuta eliminazione.

Se il POI non venisse trovato ritorna un messaggio di errore che enuncia il mancato ritrovamento del POI all'interno del sistema.

```
//delete a poi
router.delete('/:id', async (req, res) => {
  let poi = await Poi.findById(req.params.id).exec();
  if (!poi) {
    res.status(404).send()
    console.log('Poi non trovato')
    return;
  }
  await poi.deleteOne()
  console.log('Poi rimosso')
  res.status(204).send()
});
```

**Figura 13. Eliminazione di un POI**

### 2.5.6 MODIFICA POI

Questa API permette al gestore di modificare (eventualmente) un POI.

Inizialmente viene richiesto quale POI modificare e, successivamente, viene fornito un form in cui si possono inserire le modifiche dei vari campi della classe POI.

Ritorna un messaggio di errore nel caso in cui la modifica non andasse a buon fine.

```
//modifica poi
router.put('/:id', async (req, res) => {
  let poi = await Poi.findById(req.params.id);
  if(poi){
    let poiUpdate = await Poi.findByIdAndUpdate(req.params.id,{ $set:{nome:nome , descrizione:descrizione
    , posizione:pos , stato:stato , orari:orari_apertura}}, {new:true,runValidators:true});
    res.status(201).send();
  }
  else{
    res.status(204).send();
  }
});
```

**Figura 14. Modifica di un POI**

### 3. API DOCUMENTATION

Le API descritte nel capitolo precedente e fornite dall'applicazione "All About Trento" vengono documentate utilizzando il modulo *NodeJS*, chiamato *Swagger UI Express*. Facendo così, la documentazione delle API principali del sistema è disponibile per chiunque voglia accedere al codice sorgente.

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

Nell'immagine sottostante viene mostrata la pagina web relativa alla documentazione che presenta le 3 API per la gestione dei dati della nostra applicazione, ovvero GET, POST e DELETE.

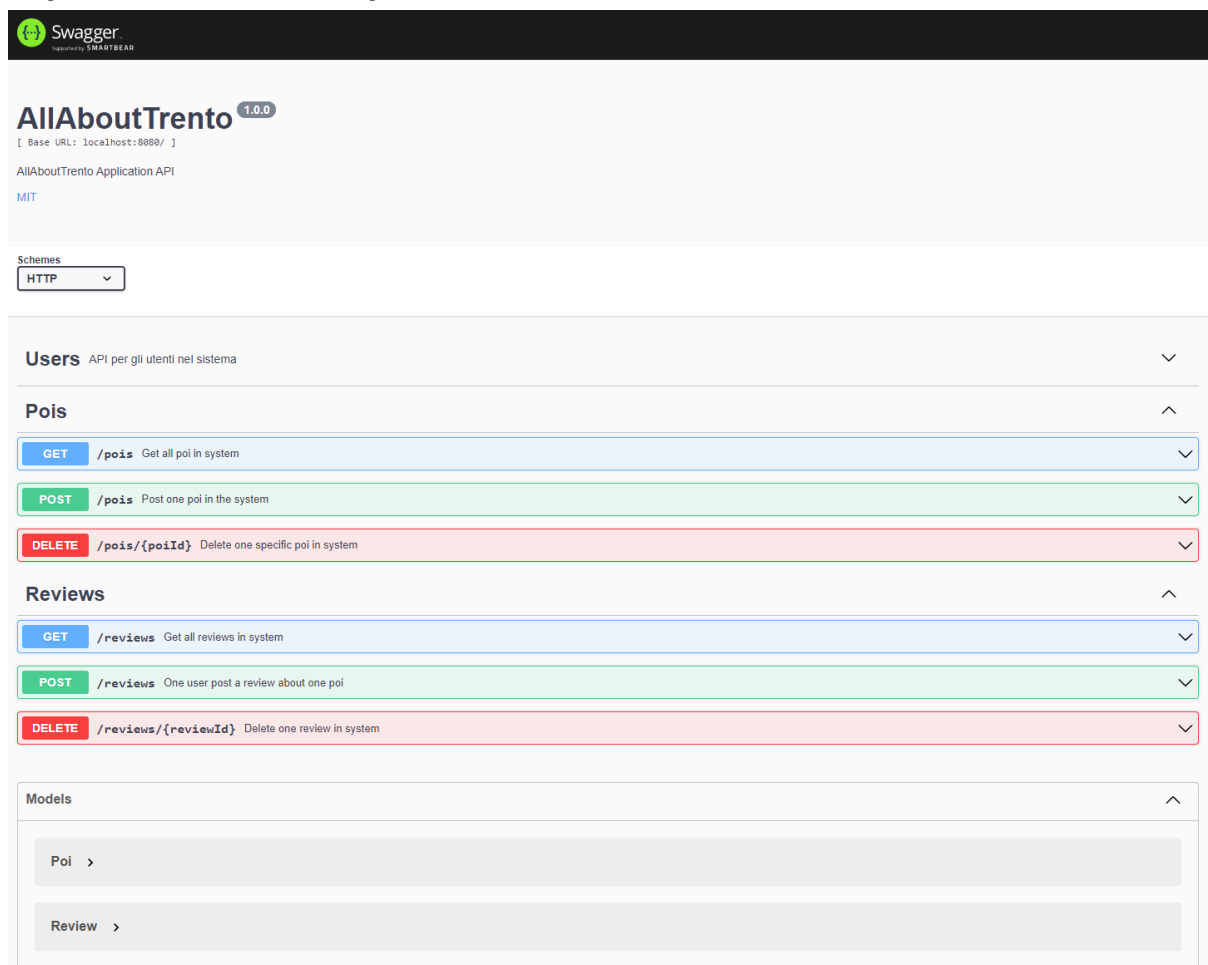
La GET viene utilizzata per la visualizzazione dei dati all'interno di una pagina HTML.

La POST viene utilizzata per l'inserimento di un nuovo dato all'interno del sistema.

La DELETE viene utilizzata per cancellare un dato.

L'*endpoint* da invocare per raggiungere la seguente documentazione è:

***http://localhost:8080/api-docs***



**Figura 15. Documentazione delle API di "All About Trento"**

## 4. FRONTEND IMPLEMENTATION

Il FrontEnd fornisce le funzionalità di visualizzazione, inserimento e cancellazione dei dati dell'applicazione "All About Trento".

Il sito è composto da una Home Page, da una pagina per la visualizzazione dei POI, da una pagina per l'inserimento delle recensioni (solo per gli utenti registrati) e da una pagina per l'inserimento, la modifica e la cancellazione dei POI (solo per i gestori).

Nella Home Page sono presenti vari pulsanti che guidano la navigazione all'interno del sito per accedere alle pagine di servizio, come ad esempio la pagina di 'Visualizzazione POI'.

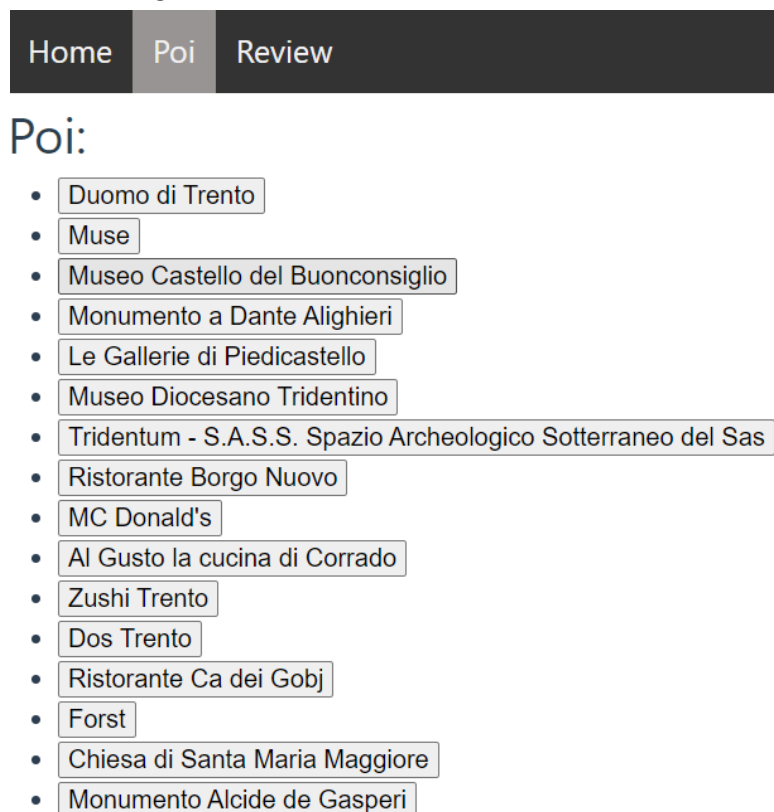


Il sito ha come obiettivo quello di favorire il turismo culturale di una città d'arte di medie dimensioni come Trento, offrendo ai visitatori un servizio per la fruizione di contenuti multimediali che descrivono i "punti di interesse" (POI) di tipo monumentale, artistico e culinario.

***Figura 16. Home Page di "All About Trento"***



La pagina di ‘Visualizzazione POI’ viene utilizzata dall’utente per poter esaminare i vari POI registrati all’interno del sistema.

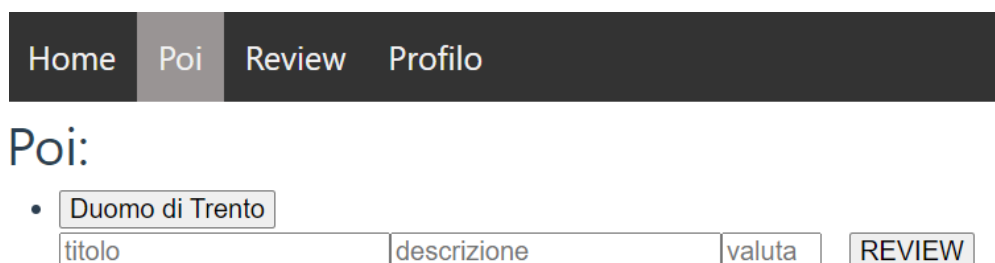


**Figura 17. Pagina ‘Visualizzazione POI’**

La pagina ‘Inserimento Recensioni’ viene utilizzata per inserire delle recensioni sul sito oppure su un POI specifico.

La pagina è composta da varie text-box dove l’utente può inserire il titolo e la descrizione della recensione e, inoltre, può aggiungere una valutazione (da 1 a 5) sul POI scelto oppure sul sito.

Una volta inseriti i campi necessari, è presente un pulsante che aggiunge la recensione all’interno del sistema.



**Figura 18. Pagina ‘Inserimento Recensioni’**

La pagina ‘Inserimento-Modifica-Cancellazione POI’ viene utilizzata per la gestione dei POI, ovvero per la visualizzazione, l’inserimento e, eventualmente, la modifica e la cancellazione.

Per ognuna di queste operazioni è previsto un pulsante specifico.

[Home](#) [Poi](#) [Review](#) [Profilo](#)

### Inserimento di un nuovo Poi

nome	Seleziona una Tipologia ▼	descrizione	posizione	Aperto o Chiuso ▼	orari_apertura	immagine	Inserisci nuovo Poi
------	---------------------------	-------------	-----------	-------------------	----------------	----------	---------------------

Poi:

- Duomo di Trento [DELETE](#)
- Muse [DELETE](#)
- Castello del Buonconsiglio [DELETE](#)
- Monumento a Dante Alighieri [DELETE](#)
- Le Gallerie di Piedicastello [DELETE](#)
- Museo Diocesano Tridentino [DELETE](#)
- Tridentum - S.A.S.S. Spazio Archeologico Sotterraneo del Sas [DELETE](#)
- Ristorante Borgo Nuovo [DELETE](#)
- MC Donald's [DELETE](#)
- Al Gusto la cucina di Corrado [DELETE](#)
- Zushi Trento [DELETE](#)
- Dos Trento [DELETE](#)
- Ristorante Ca del Gobj [DELETE](#)
- Forst [DELETE](#)
- Chiesa di Santa Maria Maggiore [DELETE](#)
- Monumento Alcide de Gasperi [DELETE](#)

[Home](#) [Poi](#) [Review](#) [Profilo](#)

## Duomo di Trento : monumento

### Duomo di Trento

[Update](#)

**Figure 19-20. Pagine di ‘Inserimento-Modifica-Cancellazione POI’**

## 5. GITHUB REPOSITORY AND DEPLOYMENT INFO

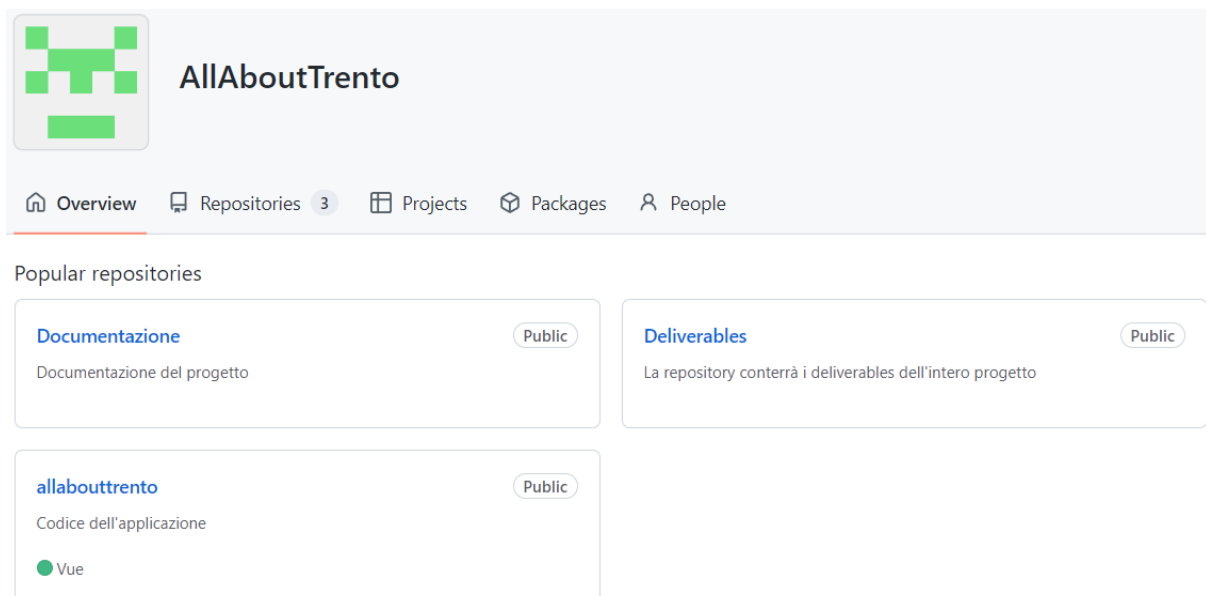
L'intero progetto di "All About Trento" è stato caricato al seguente link:

<https://github.com/All-About-Trento>

All'interno della Home Page sono state realizzate diverse repository, ognuna con delle funzionalità precise (vedi sotto Figura 20).
















Le repository realizzate sono le seguenti:

- *Documentazione*: viene riportata la documentazione del progetto
- *Deliverables*: vengono riportati tutti i deliverables del progetto
- *allabouttrento*: viene riportato il codice del progetto



**Figura 21. Home Page Git del progetto "All About Trento"**

Come spiegato precedentemente, all'interno della repository “*allabouttrento*” viene riportato il codice dell'intero progetto (vedi Figura 21). Sono stati caricati tutti i file necessari per poter avviare il progetto e alcuni di essi sono stati caricati all'interno di cartelle apposite per alcune funzionalità specifiche.

	Manuvet02 Modifica inserimento	4e661e8 now	 16 commits
	.vscode	primo commit	2 weeks ago
	dist	Modifica inserimento	now
	public	primo commit	2 weeks ago
	server	Modifiche al testing	20 hours ago
	src	Modifica inserimento	now
	.gitignore	.env	last week
	Procfile	primo commit	2 weeks ago
	README.md	Update README.md	last week
	index.html	primo commit	2 weeks ago
	package-lock.json	Commit progetto	last week
	package.json	Modifiche al testing	20 hours ago
	swagger.json	primo commit	2 weeks ago
	vite.config.ts	primo commit	2 weeks ago

**Figura 22. Repository Git del progetto “All About Trento”**

Vengono riportate, inoltre, tutte le istruzioni per poter avviare correttamente il progetto (vedi Figura 22).

In particolare, vengono riportate le procedure per poter avviare l'applicazione e per avviare il server collegata ad essa.

## AllAboutTrento

---

### Installare npm

---

```
npm install
```

Per avviare l'applicazione e il server in contemporanea dovremo utilizzare due terminali (creare due powershell su VisualStudio Code)

Nel primo terminale bisogna avviare l'applicazione

```
npm run start
```

Nel secondo terminale , per avviare il server dobbiamo spostarci nella cartella /server

```
cd .\server\
```

E invocare il comando node per avviare effettivamente il server

```
node .\server.js
```

***Figura 23. Istruzioni per avviare il progetto “All About Trento”***

## 6. TESTING

In questa sezione ci soffermeremo sulle funzionalità non ancora descritte e analizzate precedentemente.

Per poter effettuare al meglio la fase di verifica e validazione abbiamo impostato dei punti fondamentali da rispettare per poter ottenere un risultato corretto e coerente alle aspettative.

I punti principali da analizzare sono:

- identificazione descrizione test case
- test data
- eventuali precondizioni
- dipendenze
- risultato atteso
- risultato ottenuto

La fase di testing è stata svolta attraverso un'analisi statica e dinamica dei test case, i quali sono descritti successivamente.

### **REGISTRAZIONE**

La funzione di registrazione permette all'utente di registrarsi all'interno del sistema ed avere la possibilità di accedere a funzionalità più specifiche e interattive.

```
//post a new utente
router.post('/', async (req, res) => {
  let utenti = new Utenti({
    nome : req.body.nome,
    cognome : req.body.cognome,
    email : req.body.email,
    numTelefono : req.body.numTelefono,
    password : req.body.password,
    ruolo : req.body.ruolo
  });

  utenti = await utenti.save();
  let utentiId = utenti.id;
  console.log('Utente saved successfully');
  res.location("/utentis/" + utentiId).status(201).send();
});
```

**Figura 24. Codice Funzione REGISTRAZIONE**

Attraverso l'analisi statica del codice precedentemente riportato non abbiamo riscontrato nessuna anomalia che potesse causare problemi durante l'esecuzione del programma.

```
PASS server/utenti.test.js
```

#### Testing Login

- ✓ GET /utentis/me with no token should return 401 (21 ms)
- ✓ GET /utentis/me?token=<invalid> should return 401 (4 ms)
- ✓ GET /utentis/me?token=<valid> should return 200 (4 ms)

#### Testing Registrazione

- ✓ should respond with 201 (745 ms)

**Figura 25. Testing REGISTRAZIONE**

Per svolgere al meglio questa fase abbiamo svolto un'analisi dinamica della funzione, attraverso l'esecuzione del codice stesso fornendo come dati di ingresso i dati personali di un utente non registrato, i quali sono stati accettati correttamente.

## VALUTAZIONE DELL'UTENTE

Questa funzione controlla che tipologia di utente è durante la fase di login. Come affermato nei precedenti documenti, all'interno del nostro sistema sono presenti 3 tipi di utenti: l'utente non registrato, l'utente registrato e il gestore. L'utente non registrato può decidere di registrarsi per poi effettuare il login all'interno del sistema.

La funzione in questione va a cercare che tipologia di utente è e, in base a ciò, cambiano le funzionalità una volta effettuato l'accesso, ovvero le funzionalità che sono proposte all'utente registrato non sono le stesse di quelle del gestore.

```
if($cookies.get("ruolo")==="gestore"){  
    ruoloGestore = true;  
}  
else if($cookies.get("ruolo")==="utente"){  
    ruoloUtente = true;  
}
```

**Figura 26. Codice Funzione VALUTAZIONE DELL'UTENTE**

Attraverso l'analisi statica del codice precedentemente riportato non abbiamo riscontrato nessuna anomalia che potesse causare problemi durante l'esecuzione del programma.

```
PASS server/utenti.test.js
```

#### Testing Ruolo Utente

- ✓ GET /utentis/63e374453fb7a44d700ef04a should respond with "utente" (853 ms)

#### Testing Ruolo Gestore

- ✓ GET /utentis/6399a501490ca809bc4879f9 should respond with "gestore" (28 ms)

**Figura 27. Testing VALUTAZIONE DELL'UTENTE**

Per svolgere al meglio questa fase abbiamo svolto un'analisi dinamica della funzione, attraverso l'esecuzione del codice stesso fornendo opportuni dati di ingresso i quali sono stati accettati correttamente.

### **CAMBIO PASSWORD**

Questa funzione permette a tutti gli utenti registrati al sito di poter cambiare la propria password.

Quando un'utente decide di cambiare password, per ragioni ovvie, la nuova password dovrà essere differente rispetto alla vecchia, e deve rispettare i vincoli di validità predisposti.

```
function cambioPassword(id){  
  
  fetch(HOST+'/utentis/'+id, {  
    method: 'PUT',  
    headers: {  
      'Content-Type': 'application/json',  
      'x-access-token': loggedUser.token  
    },  
    body: JSON.stringify( {  
      oPassword: oldPassword.value ,  
      nPassword : newPassword.value  
    } )  
  })  
  .then((resp) => {  
    if(resp.status==204){  
      psMessage.value="Vecchia Password Sbagliata";  
      return;  
    }  
    else{  
      psMessage.value='';  
      location.reload();  
      return;  
    }  
  })  
  .catch( error => console.error(error) );  
}
```

**Figura 28. Codice Funzione CAMBIO PASSWORD**

Attraverso l'analisi statica del codice precedentemente riportato non abbiamo riscontrato nessuna anomalia che potesse causare problemi durante l'esecuzione del programma.

```
PASS server/utenti.test.js  
Testing Cambio Password  
✓ should respond with 201 (555 ms)
```

**Figura 29. Testing CAMBIO PASSWORD**

Per svolgere al meglio questa fase abbiamo svolto un'analisi dinamica della funzione, attraverso l'esecuzione del codice stesso fornendo opportuni dati di ingresso, ossia la nuova password inserita due volte come metodo di verifica, i quali sono stati accettati correttamente.