



UNIVERSITÀ
DI TRENTO



Documento di sviluppo dell'applicazione

Gruppo T31

- Luca Dematté
- Diego Oniarti
- Matteo Frizzera

Ingegneria del software

Anno Accademico 2022/23

Versione del documento

Versione	Data	Modifiche
0.1	5/12/2022	Creazione documento.
0.2	8/12/2022	Aggiunta struttura del documento.
0.3	21/12/2022	Aggiunti user flow con descrizione.
0.4	13/02/2023	Aggiornamento user flow.
0.5	14/02/2023	Inserimento screenshot e descrizione frontend.
0.6	15/02/2023	Aggiunto diagramma di estrazione delle risorse.
0.7	16/02/2023	Aggiunti diagrammi dei modelli delle risorse.
0.8	17/02/2023	Aggiunti screenshot del codice e descrizione API.
1.0	17/02/2023	Aggiunta descrizione frontend, deployment e testing.
1.1	17/02/2023	Aggiornamento logo e revisioni minori.

Sommario

Scopo del documento	5
User flows	5
Login e registrazione	5
Nuova prenotazione	6
Modifica profilo e logout	7
Documentazione e implementazione dell'applicazione	9
Struttura del progetto	9
BackEnd	10
FrontEnd	11
Dipendenze del progetto	12
Database del progetto	13
API del progetto	15
Risorse estratte dal diagramma delle classi	15
Modelli delle risorse	18
Utente	19
Servizi	23
Spazi	26
Prenotazioni	29
Ricorrenze	32
Sviluppo delle API	35
Utente	35
Servizi	39
Spazi	41
Prenotazioni	43
Ricorrenze	45
Documentazione delle API	47
Implementazione del FrontEnd	50
Design	50
Esempi	50



Repository GitHub e informazioni sul deployment.....	53
Repository GitHub.....	53
Deployment	53
Backend.....	53
Frontend	54
Testing	55

Scopo del documento

Il presente documento riporta tutte le informazioni necessarie per lo sviluppo di una parte del sistema. In particolare, presenta tutti i diagrammi necessari per realizzare i servizi di gestione degli utenti, spazi, servizi e prenotazioni del sistema.

Partendo dalla descrizione degli user flow legati al ruolo di un normale utente del sistema, il documento prosegue con la presentazione delle API necessarie (tramite l'API Model e il Modello delle risorse) per poter visualizzare, inserire e modificare le risorse del sistema.

Per ogni API realizzata, oltre ad una descrizione delle funzionalità fornite, il documento presenta la sua documentazione e i test effettuati. Infine, una sezione è dedicata alle informazioni del Git Repository e il deployment dell'applicazione stessa.

User flows

In questa sezione del documento di sviluppo riportiamo gli "user flows" per il ruolo dell'utente della nostra applicazione.

Login e registrazione

La Figura 1 descrive il procedimento di login o di registrazione dell'utente.

In ogni momento e da ogni pagina, un utente non autenticato può premere il relativo pulsante per effettuare il login o per registrarsi, raggiungendo la pagina per effettuare l'operazione selezionata. Dopo aver inserito i dati richiesti, viene effettuato un controllo sulla loro validità:

- Nel caso in cui i dati inseriti siano errati, l'utente viene invitato ad inserire nuovamente i dati,
- Se invece i dati inseriti sono corretti, l'operazione è conclusa e l'utente viene riportato alla pagina che stava visualizzando prima di iniziare il procedimento, ma questa volta è autenticato.

NB: abbiamo realizzato il sistema in modo che, dopo la registrazione, invii VERAMENTE un'e-mail di conferma all'indirizzo inserito. Durante i test consigliamo quindi di utilizzare un indirizzo e-mail effettivamente esistente e al quale si ha accesso.

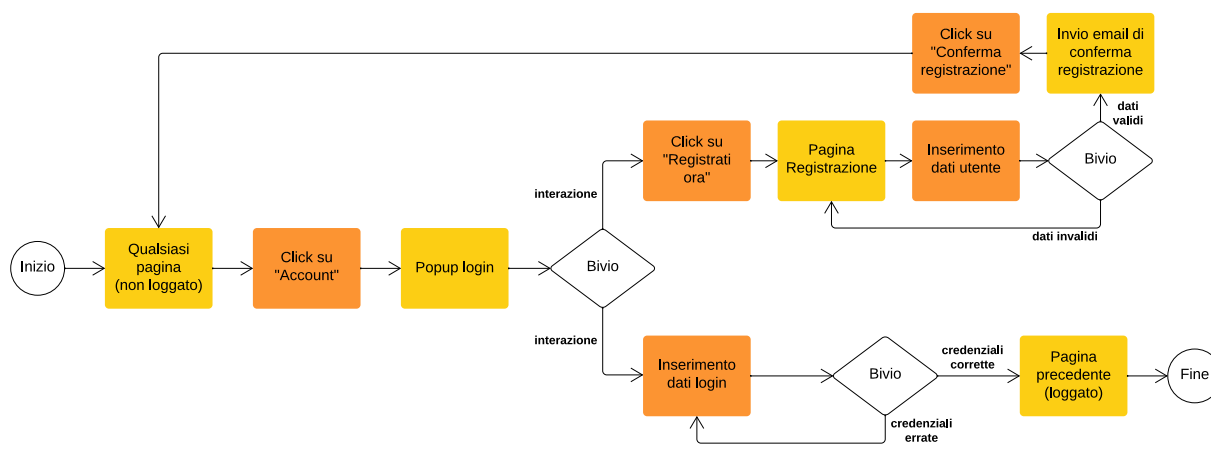


Figura 1 – User flow di login e registrazione

Nuova prenotazione

La Figura 2 mostra un esempio di inserimento di prenotazione da parte di un utente autenticato.

Dopo aver selezionato l'opzione "Nuova prenotazione", viene richiesto di inserire i dati relativi alla prenotazione.

Successivamente, l'utente deve indicare se la prenotazione è per un evento pubblico (che quindi diventerà visibile sul sito) oppure se si tratta di un evento privato. Nel caso in cui l'evento sia pubblico, vengono richiesti i dati relativi all'evento per la visualizzazione sul sito.

In seguito, viene richiesto di indicare come si desidera effettuare il pagamento:

- Se l'utente seleziona il pagamento online, viene avviata la procedura di pagamento tramite il sistema esterno Nexi,
- Se invece l'utente decide di pagare in contanti (o non dispone di un metodo di pagamento online) viene informato su come ed entro quando recarsi in segreteria per pagare la prenotazione.

Al termine di entrambi i casi, la prenotazione viene registrata e viene mostrato un avviso di conferma.

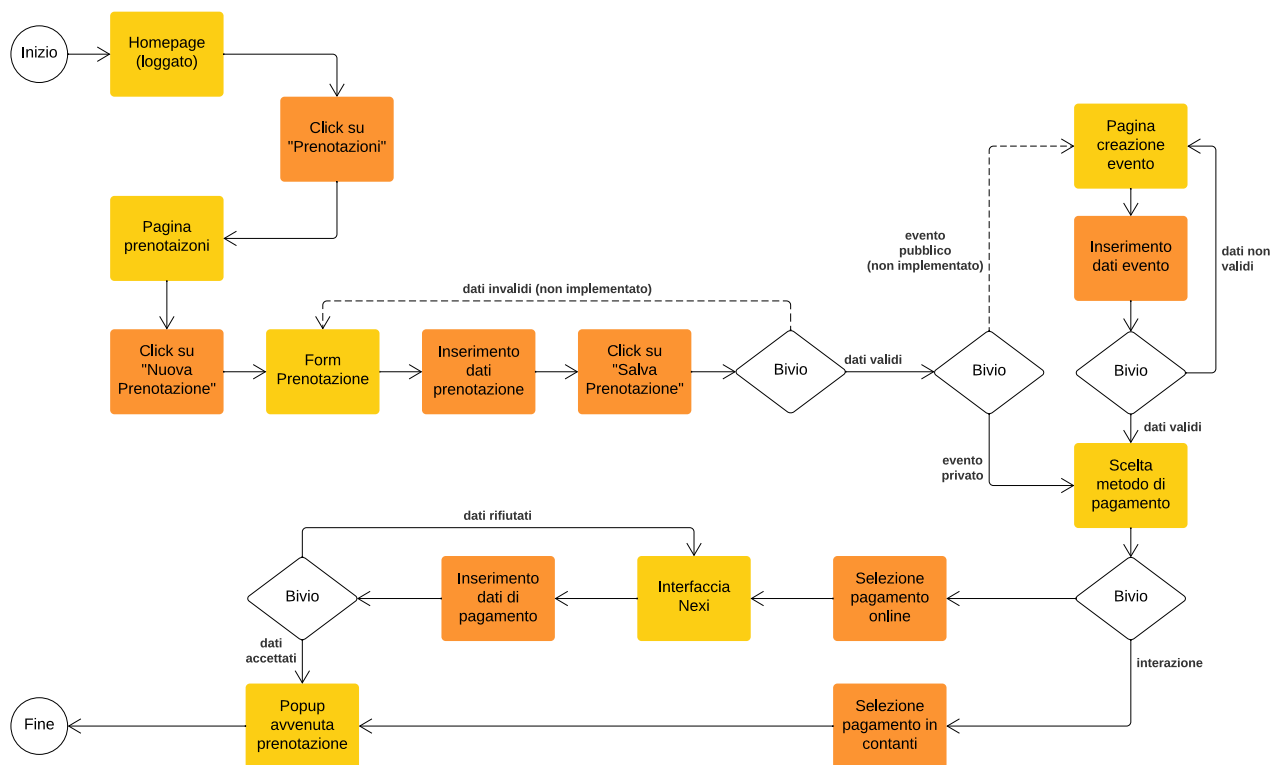


Figura 2 – User flow di inserimento di una prenotazione

Modifica profilo e logout

La Figura 3 mostra i passaggi che l'utente deve svolgere per vedere e modificare il proprio profilo o per poter effettuare il logout.

Nel primo caso, dopo aver cliccato sulla propria immagine, viene portato alla pagina del profilo, dove può opzionalmente modificare le proprie informazioni personali attivando la modifica del form con il pulsante "Modifica profilo". Dopo aver effettuato tutte le modifiche che desidera, premendo su "Salva" esse vengono verificate e, se sono valide, viene visualizzata nuovamente la pagina del profilo con i dati aggiornati.

Nel caso del logout, dopo l'operazione l'utente viene riportato alla pagina che stava visualizzando in precedenza oppure, se essa è disponibile solo per gli utenti autenticati, alla homepage.

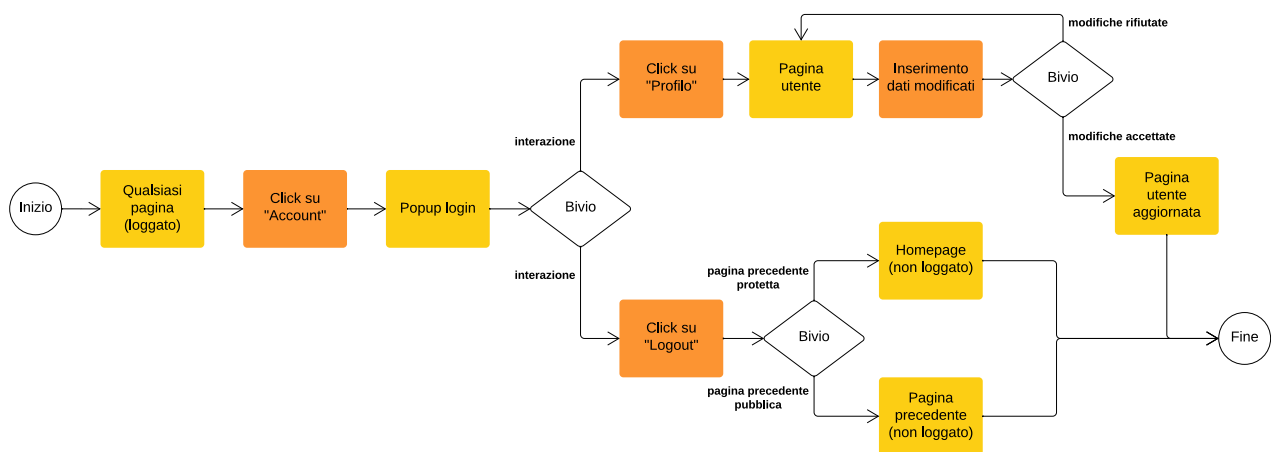


Figura 3 – User flow di modifica profilo e logout

Documentazione e implementazione dell'applicazione

Nelle sezioni precedenti abbiamo identificato le varie features che devono essere implementate per la nostra applicazione con un'idea di come il nostro utente può utilizzarle nel suo flusso applicativo.

L'applicazione è stata sviluppata utilizzando NodeJS e VueJS. Per la gestione dei dati abbiamo utilizzato MongoDB.

A causa del grande numero e della complessità delle risorse che verranno descritte nei paragrafi successivi, abbiamo deciso di non implementare la parte relativa agli eventi della nostra applicazione. Nonostante questo, riteniamo che il core business del sistema ideato, ovvero il poter prenotare spazi, rimanga sufficientemente completo e variegato per mettere in pratica le nostre conoscenze apprese durante le lezioni.

Un'altra parte del sistema che abbiamo deciso di non sviluppare è quella relativa al pagamento della prenotazione. Questa scelta è derivata dal fatto che l'utilizzo di sistemi di pagamento online (nel nostro caso era stato scelto Nexi) richiede una profonda conoscenza delle complesse API fornite dal sistema.

Inoltre, dato che il sistema realizzato è una simulazione e viene usato solo per casi di test, è naturale l'esclusione di spostamenti di denaro dal flusso di utilizzo.

Struttura del progetto

Il progetto è stato strutturato in due repository: una per il BackEnd e una per il FrontEnd. Nelle pagine successive ne descriviamo l'organizzazione con l'aiuto di screenshot dell'albero delle cartelle.

BackEnd

Il repository del Backend è strutturato come è visibile nella Figura 4. Sono presenti le seguenti cartelle:

- **controllers**
Qui risiede la business logic dell'applicazione, ovvero i metodi che vengono chiamati quando si riceve una richiesta alle API.
- **images**
Questa cartella contiene le immagini caricate dal FrontEnd relative a utenti, spazi e servizi.
Per fare in modo che il FrontEnd possa recuperare queste immagini, questa cartella è esposta da express sul path /images.
- **models**
Vengono raggruppate qui le definizioni dei modelli utilizzati da mongoose per salvare informazioni su MongoDB.
- **routes**
I file di questa cartella si occupano di chiamare il giusto controller in base al tipo di richiesta ricevuta.
- **scripts**
In questa cartella sono stati salvati vari file con funzioni utili in varie parti dell'applicazione, come ad esempio il sistema di invio di e-mail o quello per verificare l'autenticazione di un utente quando viene richiesta un'API.

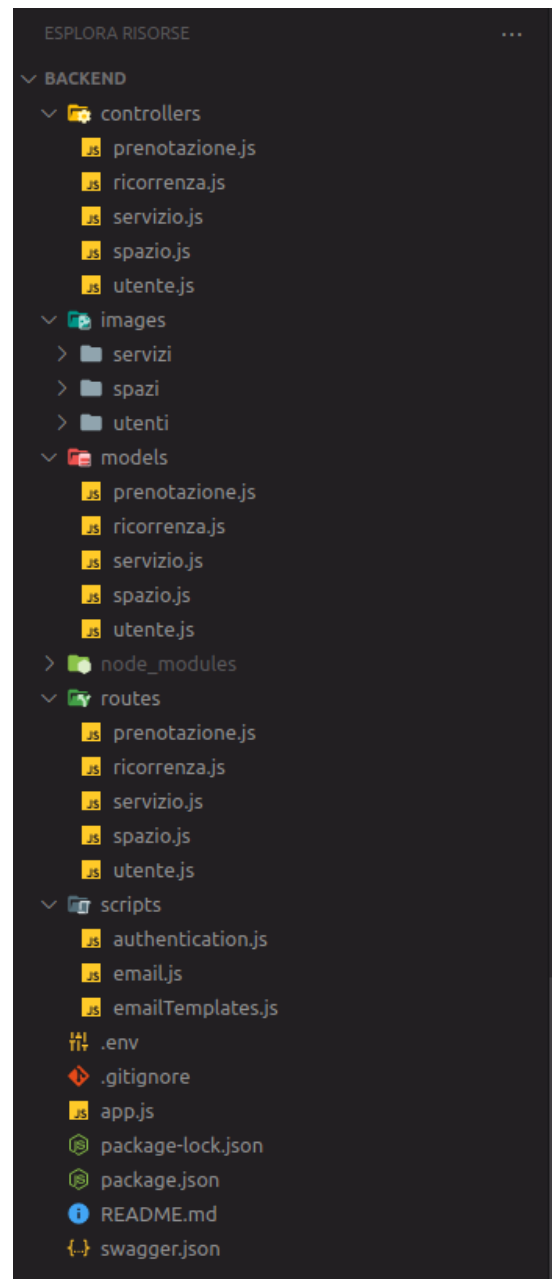


Figura 4 – Struttura Backend

Nella root sono presenti il file **app.js** (punto di avvio dell'applicazione), il file **swagger.json** (file JSON contenente le informazioni per visualizzare la pagina di documentazione) e altri file relativi alla gestione di Git, Node o dei suoi pacchetti.

FrontEnd

Nella Figura 5 riportiamo la struttura del repository del FrontEnd, composto dalle seguenti cartelle:

- **dist**
Qui viene creato il progetto buildato pronto per essere hostato.
- **public**
I file statici accessibili dal browser sono memorizzati qui, come ad esempio la pagina conferma.html che viene visualizzata quando si apre il link presente nell'email di conferma registrazione.
- **src**
In questa cartella risiede tutta l'applicazione Vue, divisa in varie sottocartelle:
 - **assets**
Immagini e fogli di stile
 - **components**
Componenti Vue utilizzati dalle varie views
 - **router**
Contiene il router che si occupa di mostrare le varie viste in base a cosa si clicca sulla navbar del sito
 - **states**
Viene salvato qui il file che si occupa di settare lo stato di un utente quando effettua il login
 - **views**
Questa cartella contiene tutte le pagine dell'applicazione.

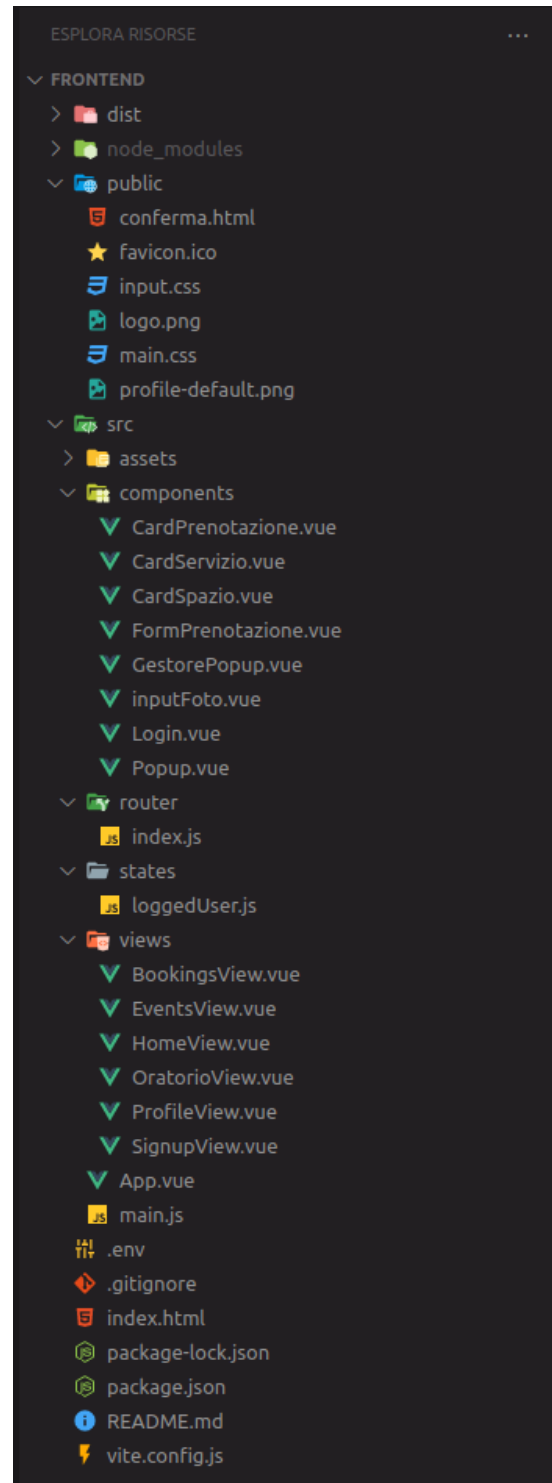


Figura 5 – Struttura FrontEnd

Sono poi presenti il file `index.html` (dove viene montata l'applicazione Vue) e altri file nella root utilizzati dal server Node per far funzionare il progetto Vue.

Dipendenze del progetto

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

- **express** (v4.18.2)
Ambiente di esecuzione del server web.
- **mongoose** (v6.8.0)
Collegamento a MongoDB.
- **dotenv** (v16.0.3)
Utilizzo di variabili d'ambiente di configurazione accessibili da ogni file.
- **jsonwebtoken** (v9.0.0)
Generazione e gestione di token per l'autenticazione alle API.
- **swagger-ui-express** (v4.6.0)
Documentazione delle API basata sulle specifiche OpenAPI.
- **body-parser** (v1.20.1)
Middleware che permette a express di ricevere il body dalle richieste di tipo POST in un formato facile da utilizzare.
- **cookie-parser** (v1.4.6)
Middleware che permette a express di leggere i cookies presenti negli headers di una richiesta.
- **cors** (v2.8.5)
Aggiunge a tutte le risposte inviate dal server i parametri CORS necessari a soddisfare le CORS policies sul client.
- **multer** (v1.4.5-lts.1)
Middleware che permette a express di ricevere file inviati tramite form dal frontend (usato per le immagini).
- **nodemailer** (v6.8.0)
Sistema di invio e-mail per comunicazioni agli utenti.
- **nodemon** (v2.0.20)
Modulo per il development che riavvia il server ad ogni modifica ai file.
- **jest** (v29.4.3)
Framework utilizzato per il testing.
- **node-fetch** (v2.6.9)
Modulo per simulare richieste fetch provenienti dal frontend.

Database del progetto

Per la gestione dei dati utili all'applicazione abbiamo definito le strutture dati come illustrato nella Figura 6. In seguito, le strutture dati vengono descritte brevemente e ne viene presentato un esempio.

LOGICAL DATA SIZE: 6.67KB STORAGE SIZE: 208KB INDEX SIZE: 208KB TOTAL COLLECTIONS: 7 **CREATE COLLECTION**

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
eventi	0	0B	0B	4KB	1	4KB	4KB
metodoPagamento	2	274B	137B	36KB	1	36KB	36KB
prenotazioni	1	132B	132B	36KB	1	36KB	36KB
ricorrenze	0	0B	0B	24KB	1	24KB	24KB
servizi	14	3.02KB	222B	36KB	1	36KB	36KB
spazi	7	2.31KB	338B	36KB	1	36KB	36KB
utenti	3	963B	321B	36KB	1	36KB	36KB

Figura 6 – Collections del progetto su MongoDB

eventi:

Qui vengono memorizzati gli eventi associati alle prenotazioni inserite dagli utenti.

(Come già specificato all'inizio del documento, la parte del sistema relativa agli eventi non è stata sviluppata ulteriormente)

metodoPagamento:

Contiene le informazioni necessarie ad effettuare un pagamento, ovvero i dati della carta di credito inserita dall'utente.

(Come già specificato all'inizio del documento, la parte del sistema relativa al pagamento non è stata sviluppata ulteriormente)

```
_id: ObjectId('63a3a189898364bac0118187')
nome: "MARIO ROSSI"
circuito: "Mastercard"
IBAN: "IT3100300203280114128671598"
scadenza: "04/27"
CVV: "925"
```

prenotazioni:

Questa collection contiene i dati generali della prenotazione ed un vettore che contiene gli ID delle ricorrenze ad essa associate.

```
_id: ObjectId('63ef83f17021c04f64242254')
proprietario: ObjectId('63ed4e1f52146304e2d7557b')
statusPagamento: 0
importoPagamento: 158.5
✓ ricorrenze: Array
  0: ObjectId('63ef83f17021c04f64242250')
  __v: 0
```

ricorrenze:

Una ricorrenza rappresenta un periodo di tempo nel quale vengono utilizzati determinati spazi e servizi. Le ricorrenze devono sempre essere associate a una (e una sola) prenotazione.

```
_id: ObjectId('63ef83f17021c04f64242250')
inizio: 2023-02-17T13:40:00.000+00:00
fine: 2023-02-17T14:40:00.000+00:00
✓ spaziPrenotati: Array
  0: ObjectId('63a326d39c88f0007ca92dfa')
✓ serviziPrenotati: Array
  0: ObjectId('63a387cc898364bac011817a')
  1: ObjectId('63a39bb7898364bac0118181')
  __v: 0
```

servizi:

I servizi sono caratteristiche aggiuntive di certi spazi che possono essere inclusi nella prenotazione. Un servizio può essere associato a più spazi, nel caso in cui non sia legato a uno spazio in particolare (ad esempio tavoli, sedie, ...).

```
_id: ObjectId('63a387cc898364bac011817a')
nome: "Casse audio con microfono"
descrizione: "Impianto audio con due casse audio cablate e supporto microfono"
tipologia: "Attrezzatura specifica"
prezzoIniziale: 30
prezzoOra: 5
URLfoto: "/images/servizi/default.png"
```

spazi:

Qui vengono salvate le informazioni relative agli spazi dell'oratorio e i servizi che possono essere utilizzati in un determinato spazio.

```
_id: ObjectId('63a326d39c88f0007ca92dfa')
nome: "Sala polifunzionale"
descrizione: "Grande sala con palco e proiettore per proiezioni, eventi teatrali, bu..."
tipologia: "Locale interno"
prezzoIniziale: 10
prezzoOra: 3.5
URLfoto: "/images/spazi/salaPolifunzionale.png"
v servizi: Array
  0: "63a387cc898364bac011817a"
  1: "63a39bb7898364bac0118181"
  2: "63a39c49898364bac0118182"
  3: "63a3a0d4898364bac0118185"
__v: 0
```

utenti:

Questa collection contiene i dati dell'utente, tra cui la password che è stata cifrata e resa più sicura grazie al salt.

Il parametro confermaAccount viene inizialmente settato a false, e quando l'utente conferma la sua registrazione viene messa a true.

Il livello rappresenta i privilegi assegnati all'utente: il livello 1 rappresenta i normali utenti del sistema, il livello 2 viene dato solo agli utenti di segreteria.

```
_id: ObjectId('63ebb74eb675169ffb61a27d')
nome: "Luca"
cognome: "Dematté"
email: "luca.dematte-2@studenti.unitn.it"
password: "629db87643fd016bcc1e323119c49df70e89476024f5c5c8a3e0eb921edbb5b5"
salt: "aec14908b7acb03dba2d126686a874db"
confermaAccount: true
telefono: "1234567890"
indirizzo: "via Roma 1, Trento"
livello: 1
URLfoto: "/images/utenti/68bb962daa85f778f0924e60cf2e581a.png"
__v: 0
```

API del progetto

In questa sezione esplicitiamo le risorse da realizzare tramite il diagramma di estrazione delle risorse e il diagramma del modello delle risorse.

Risorse estratte dal diagramma delle classi

Dal diagramma delle classi realizzato nel documento di design architetture abbiamo estratto 33 risorse, divise nelle 5 entità trattate dal nostro sistema. Le risorse sono caratterizzate da un nome e da un metodo di fruizione della risorsa (GET, POST, PATCH o DELETE), e vengono separate tra risorse di tipo Backend (che quindi operano sui dati o svolgono operazioni con sistemi esterni) e Frontend (operazioni di recupero di dati).

Di seguito vengono descritte brevemente le risorse individuate (verranno poi approfondite nelle sezioni [Modelli delle risorse](#) e [Sviluppo delle API](#)):

- Risorsa utente:
 - **lista**

Permette di ottenere una lista di utenti. È possibile specificare il numero di elementi da restituire e da quale elemento partire per permettere un caricamento progressivo.
 - **login**

Tramite questa risorsa viene eseguito l'accesso al sistema da parte di un utente.
 - **registrazione**

Un utente utilizza questa risorsa per inserire i propri dati nel sistema.
 - **conferma**

Questa risorsa permette di attivare l'account dell'utente dopo che si è registrato.
 - **email, token e ID**

Queste risorse permettono di prelevare le informazioni di un singolo utente, fornendo il rispettivo parametro univoco.
 - **modifica**

Permette di modificare i dati di un utente già presente nel sistema.
 - **elimina**

Tramite questa risorsa è possibile rimuovere i dati di un utente dal sistema.
- Risorsa servizio e risorsa spazio:
 - **lista**

Permette di ottenere una lista di elementi. È possibile specificare il numero di elementi da restituire e da quale elemento partire per permettere un caricamento progressivo.
 - **ID**

Permette di prelevare le informazioni di un singolo spazio o servizio fornendo il suo ID.
 - **disponibilita**

Questa risorsa, specificando un ID e un periodo di tempo delimitato da due date, verifica se lo spazio o servizio è disponibile per essere prenotato oppure no.
 - **crea**

Tramite questa risorsa possono essere inseriti nuovi spazi e servizi nel sistema.
 - **modifica**

Questa risorsa permette di modificare i dati di un elemento già inserito.
 - **elimina**

Per rimuovere spazi o servizi nel sistema viene utilizzata questa risorsa.

- Risorsa prenotazione:
 - **lista e listaPerUtente**

Queste risorse permettono di ottenere un elenco di prenotazioni. È possibile specificare il numero di elementi da restituire e da quale elemento partire per permettere un caricamento progressivo.
 - **ID**

Permette di prelevare le informazioni di una singola prenotazione fornendo il suo ID.
 - **ricorrenze**

Questa risorsa restituisce tutte le ricorrenze legate a una specifica prenotazione.
 - **crea**

Tramite questa risorsa viene inserita nel sistema una nuova prenotazione insieme alle ricorrenze a lei collegate. Per questo motivo non è presente una risorsa per inserire una singola ricorrenza.
 - **modifica**

Questa risorsa permette di modificare i dati di una prenotazione già registrata nel sistema.
 - **elimina**

Tramite questa risorsa è possibile rimuovere i dati di una prenotazione dal sistema.
- Risorsa ricorrenza:
 - **lista e listaPerPeriodo**

Queste risorse permettono di ottenere un elenco di ricorrenze. È possibile specificare il numero di elementi da restituire e da quale elemento partire per permettere un caricamento progressivo.
 - **ID**

Con questa risorsa si possono ottenere i dati di una qualsiasi ricorrenza.
 - **modifica**

Questa risorsa permette di modificare i dati di una ricorrenza già registrata nel sistema.
 - **elimina**

Tramite questa risorsa è possibile rimuovere i dati di una ricorrenza dal sistema e scollegarla dalla sua prenotazione.

Nella seguente Figura 7 viene riportato il diagramma di estrazione delle risorse completo.

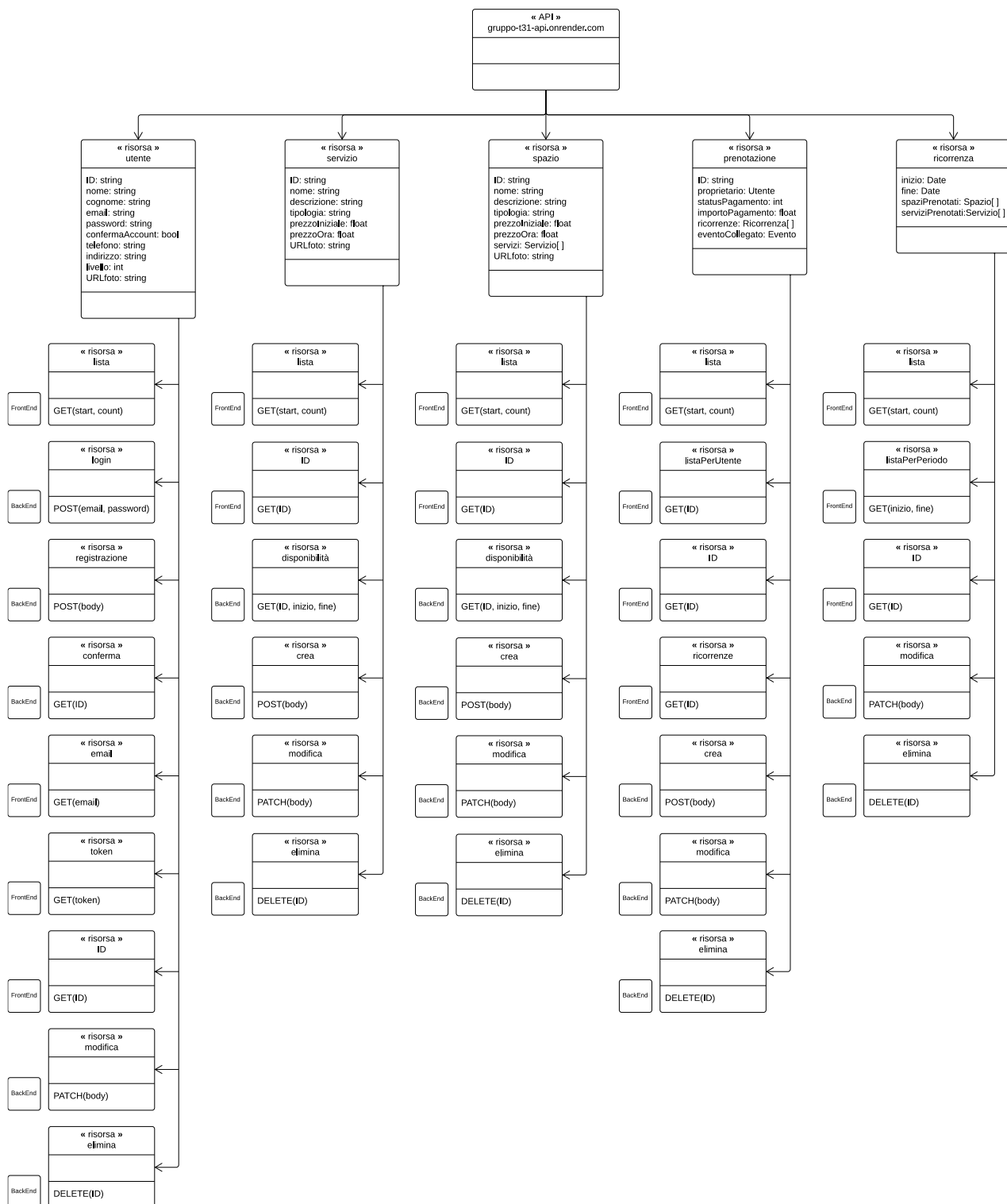


Figura 7 – Diagramma di estrazione delle risorse

Modelli delle risorse

In questo capitolo vengono riportati i diagrammi del modello delle risorse identificate grazie al diagramma precedente. Per ragioni di chiarezza e spazio, il diagramma è stato separato modello per modello. Insieme al suo diagramma, la risorsa viene descritta in modo più dettagliato, concentrandosi sui dati richiesti e generati.

Utente

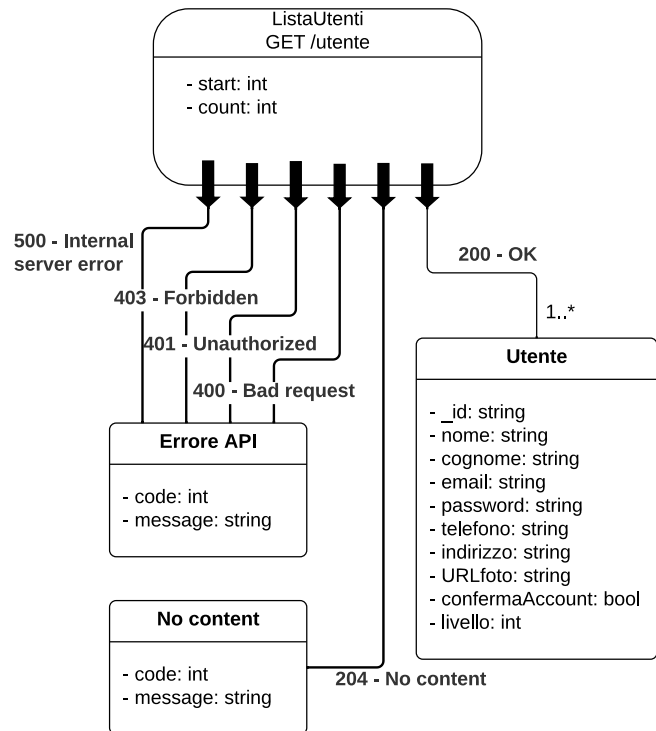
1. Elenco di utenti

- METODO: **GET**
- URI: **/utente**

Questa risorsa permette di ottenere velocemente più utenti senza effettuare multiple richieste.

È possibile includere nella query i parametri start e count. Start è l'indice del primo elemento da restituire, count è il numero di elementi da restituire.

Con dei valori adeguati per start e count è possibile caricare gli utenti in modo progressivo, ad esempio man mano che l'utente del sistema scorre verso in basso nella lista.



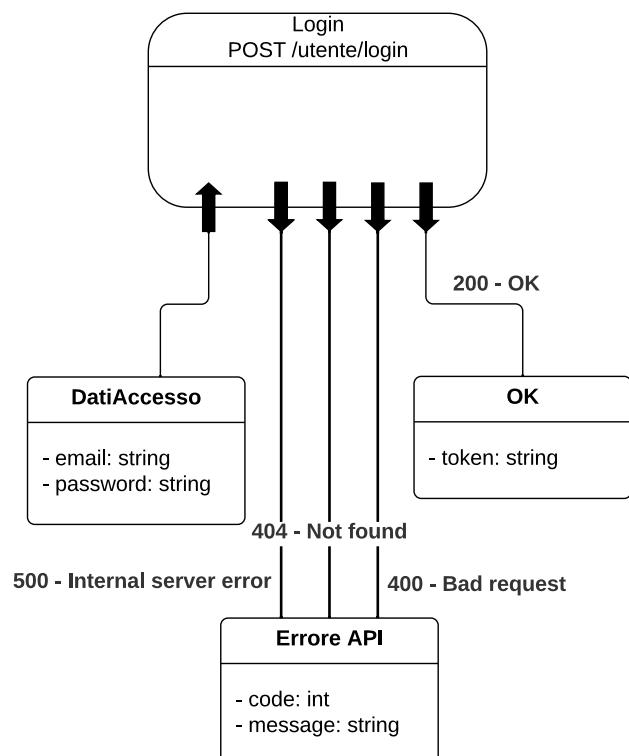
2. Login

- METODO: **POST**
- URI: **/utente/login**

Questa è la risorsa per l'autenticazione al sistema.

Quando viene invocata questa risorsa (specificando email e password), viene generato un token JWT per l'accesso a tutte le altre risorse che ne fanno richiesta.

Il token deve essere aggiunto agli headers nel parametro x-access-token.



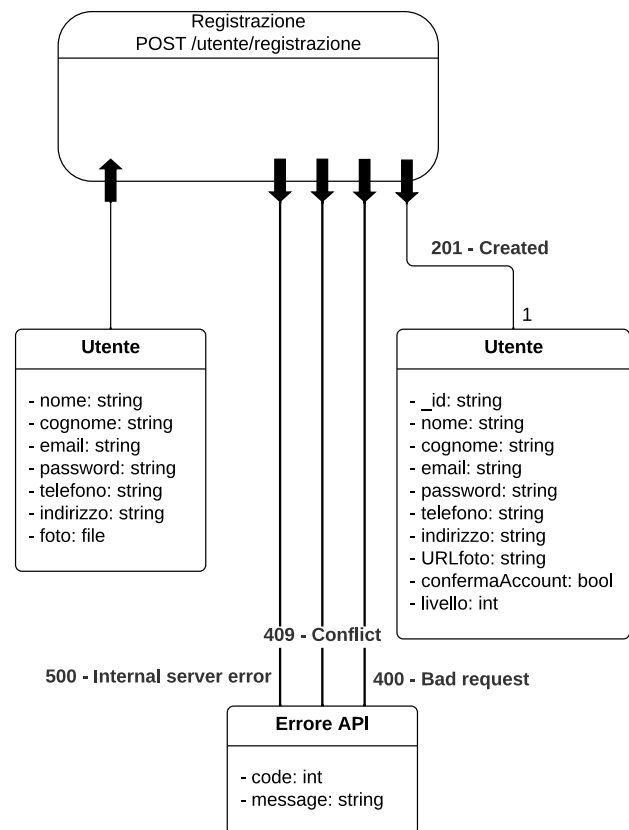
3. Registrazione

- METODO: **POST**
- URI: **/utente/registrazione**

Per registrare un nuovo utente nel sistema si utilizza questa risorsa.

Specificando i dati dell'utente, esso viene salvato nel database e viene restituito con l'aggiunta dei parametri aggiuntivi stabiliti dal sistema.

Nel caso in cui l'e-mail specificata in input fosse già stata utilizzata per registrare un altro utente, viene restituito l'errore 409 – Conflict.

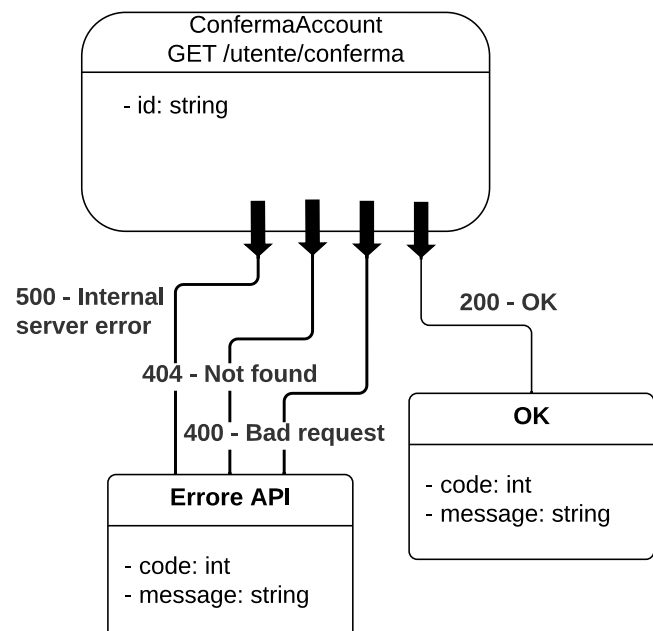


4. Conferma della registrazione

- METODO: **GET**
- URI: **/utente/conferma**

Questa risorsa è stata ideata a causa della necessità di verificare che un utente abbia utilizzato una e-mail di sua proprietà e alla quale abbia un regolare accesso.

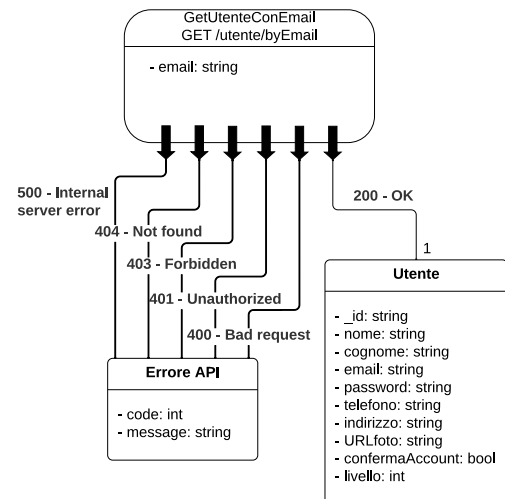
Fino a quando questa risorsa non viene chiamata specificando l'ID dell'utente, esso non sarà in grado di eseguire il login al sistema.



5. Cercare un utente tramite la sua e-mail

- METODO: **GET**
- URI: **/utente/byEmail**

Questa risorsa offre un modo per recuperare uno specifico utente, in questo caso fornendo l'e-mail dell'utente che si desidera ottenere.

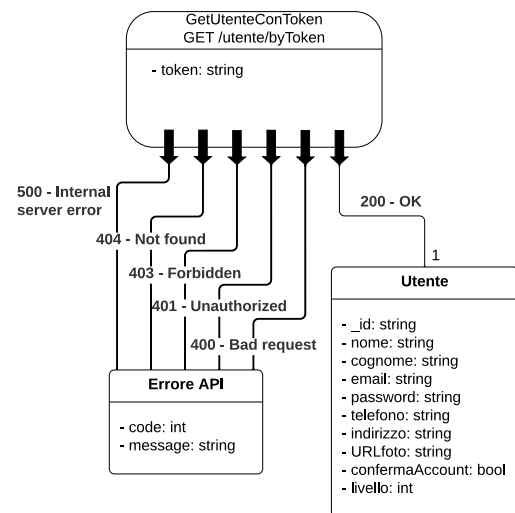


6. Cercare un utente tramite un token di accesso

- METODO: **GET**
- URI: **/utente/byToken**

Un altro modo di cercare un utente è quello di fornire un token JWT generato in precedenza per quell'utente.

Il token ricevuto viene decodificato e viene restituito l'utente proprietario di quel token.

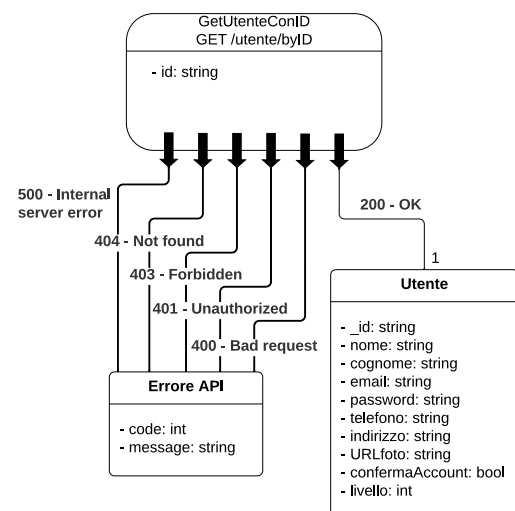


7. Cercare un utente tramite il suo ID

- METODO: **GET**
- URI: **/utente/byID**

L'ultimo modo per ottenere un utente è specificando il suo ID.

Siccome gli ID sono gli stessi utilizzati nel database, viene semplicemente fatta richiesta dell'utente con l'ID specificato.

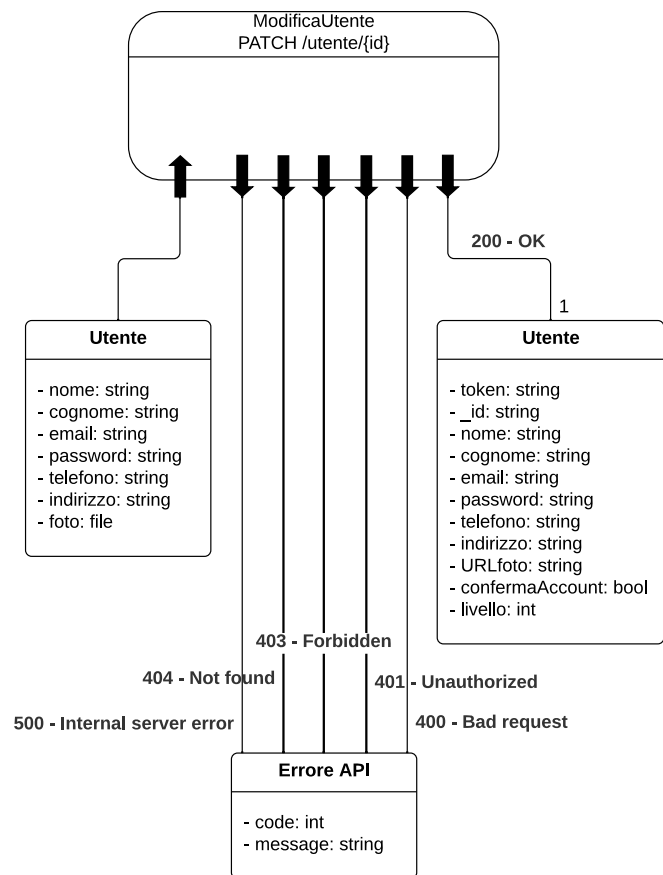


8. Modifica del profilo

- METODO: **PATCH**
- URI: **/utente/{id}**

Questa risorsa, usata per modificare un utente già registrato, fa uso di un parametro nell'URI per indicare quale utente deve essere modificato con i dati presenti nel body della richiesta.

Ovviamente, possono essere modificati solo alcuni dati dell'utente, in quanto la modifica di altri valori non è possibile o è riservata ad altre API (ad esempio la [conferma della registrazione](#)).

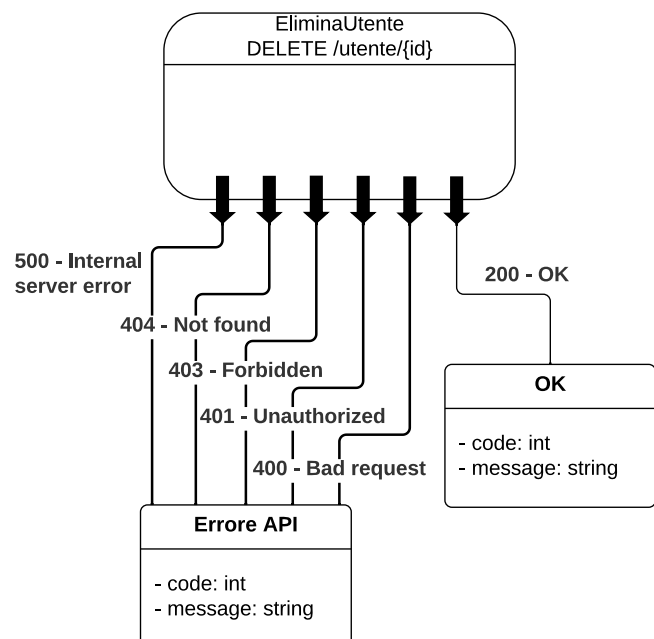


9. Cancellazione di un utente

- METODO: **DELETE**
- URI: **/utente/{id}**

Per rimuovere definitivamente un utente dal sistema basta chiamare questa risorsa.

Come specificato in precedenza, questa risorsa deve anche occuparsi dell'eliminazione di eventuali prenotazioni future effettuate dall'utente che deve essere eliminato.



Servizi

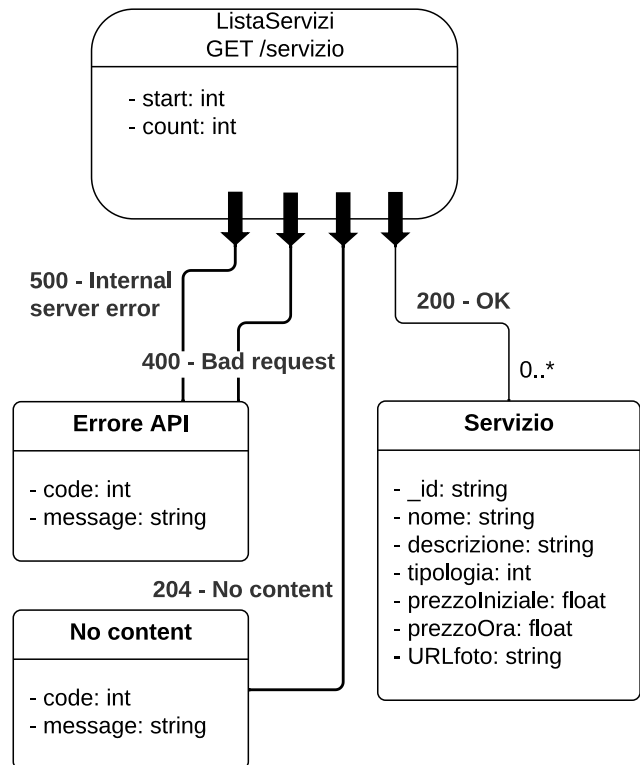
10. Elenco di servizi

- METODO: **GET**
- URI: **/servizio**

Questa risorsa permette di ottenere velocemente più servizi senza effettuare multiple richieste.

È possibile includere nella query i parametri start e count. Start è l'indice del primo elemento da restituire, count è il numero di elementi da restituire.

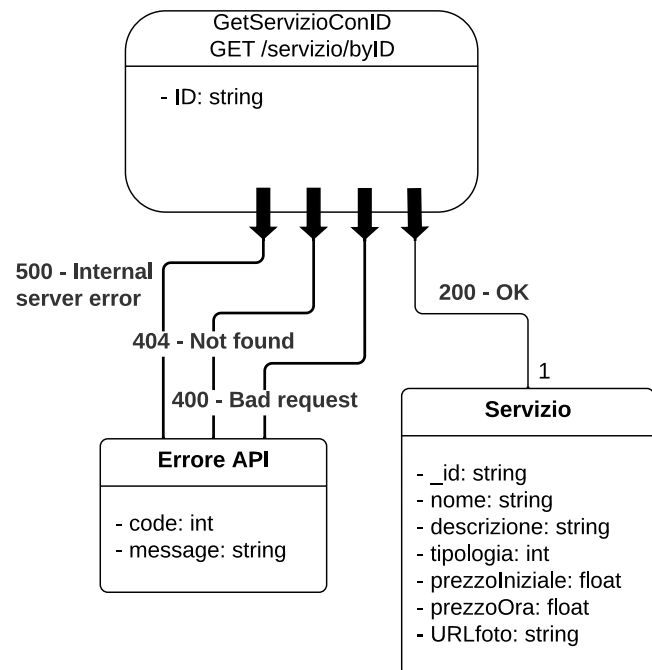
Con dei valori adeguati per start e count è possibile caricare i servizi in modo progressivo, ad esempio man mano che l'utente di segreteria sta visualizzando la lista di tutti i servizi inseriti.



11. Cercare un servizio tramite il suo ID

- METODO: **GET**
- URI: **/servizio/byID**

L'unico parametro univoco di un servizio è il suo ID; quindi, l'unica risorsa per ottenere uno specifico servizio è questa.

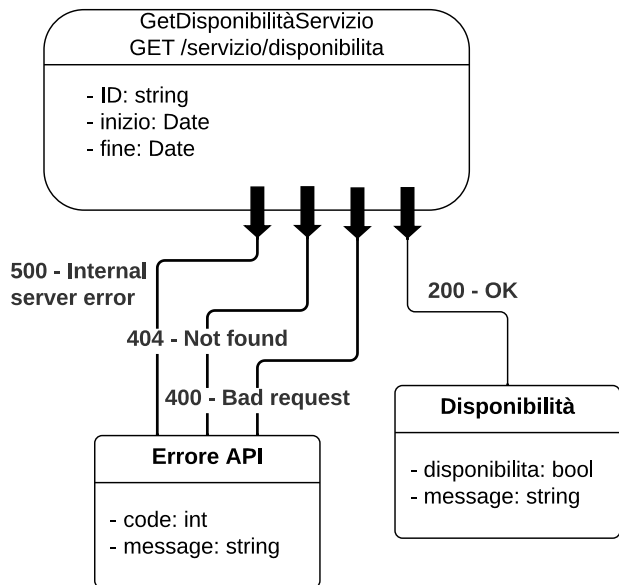


12. Verificare la disponibilità di un servizio

- METODO: **GET**
- URI: **/servizio/disponibilita**

Durante la creazione di una nuova prenotazione, è ovviamente possibile scegliere solo servizi che in quel periodo di tempo sono disponibili.

Questa risorsa verifica che nessuna ricorrenza nel periodo di tempo specificato abbia il servizio specificato dall'ID come servizio prenotato.

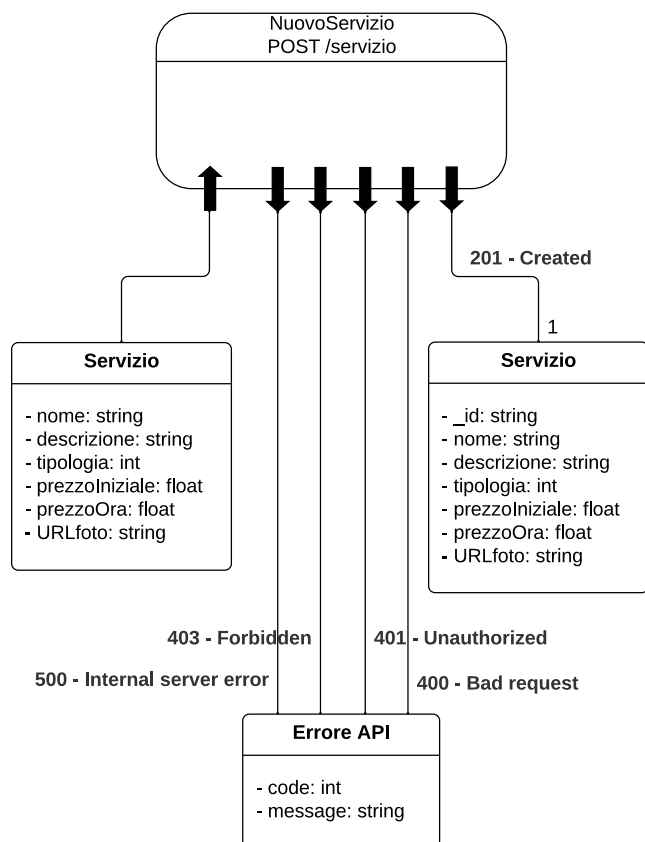


13. Inserire un nuovo servizio

- METODO: **POST**
- URI: **/servizio**

Questa risorsa, riservata agli utenti di segreteria, permette di inserire un nuovo servizio nel sistema.

Successivamente all'inserimento, il nuovo servizio può essere associato a degli spazi utilizzando la [risorsa per modificare questi ultimi](#).

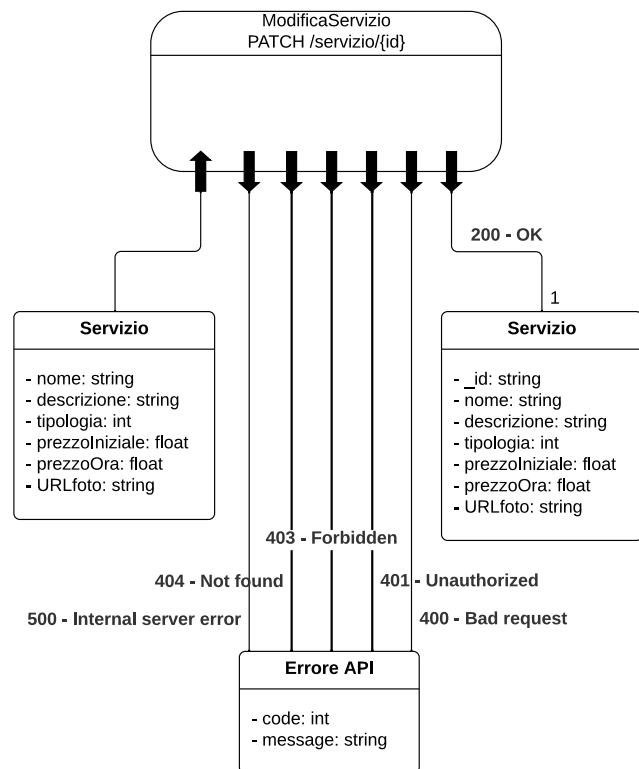


14. Modificare i dati di un servizio

- METODO: **PATCH**
- URI: **/servizio/{id}**

Questa risorsa modifica un servizio già inserito nel sistema.

Il funzionamento è molto simile a quella della risorsa per inserire nuovi servizi, l'unica differenza è che invece che inserirne uno nuovo, ne viene modificato uno già esistente (specificato dal parametro id presente nell'URI).

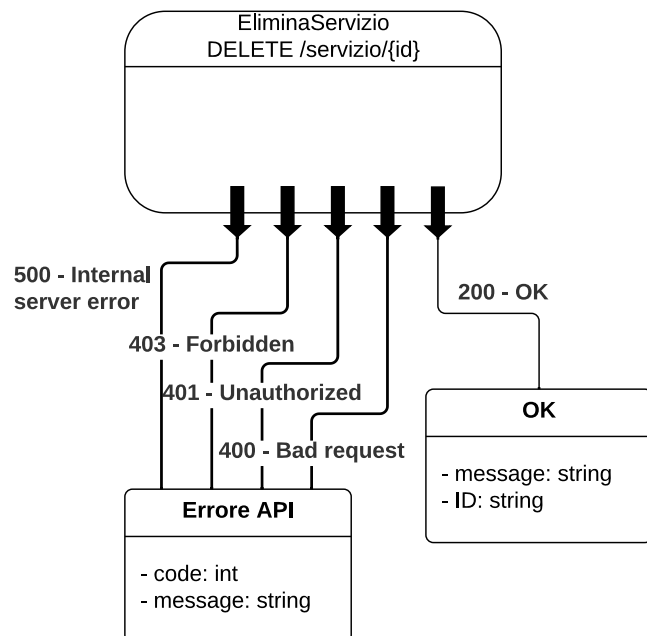


15. Eliminare un servizio

- METODO: **DELETE**
- URI: **/servizio/{id}**

Questa semplice risorsa elimina un servizio dal sistema.

L'unico aspetto da considerare è il fatto che il servizio eliminato deve essere rimosso anche da tutti gli spazi associati ad esso.



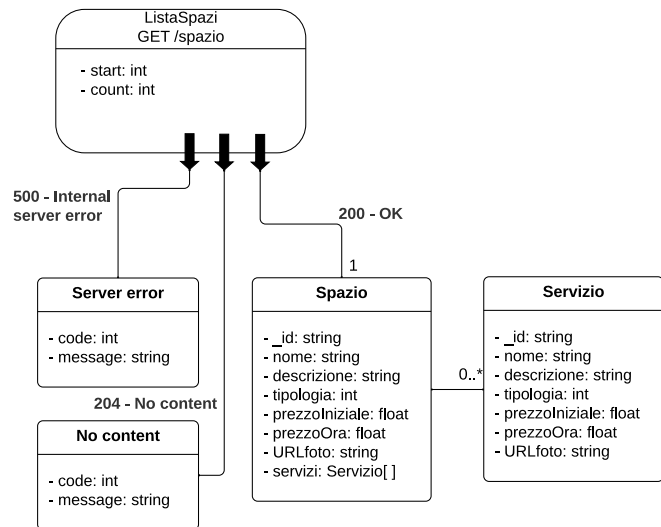
Spazi

16. Elenco di spazi

- METODO: **GET**
- URI: **/spazio**

Questa risorsa permette di ottenere velocemente più spazi senza effettuare multiple richieste.

È possibile includere nella query i parametri start e count. Start è l'indice del primo elemento da restituire, count è il numero di elementi da restituire.



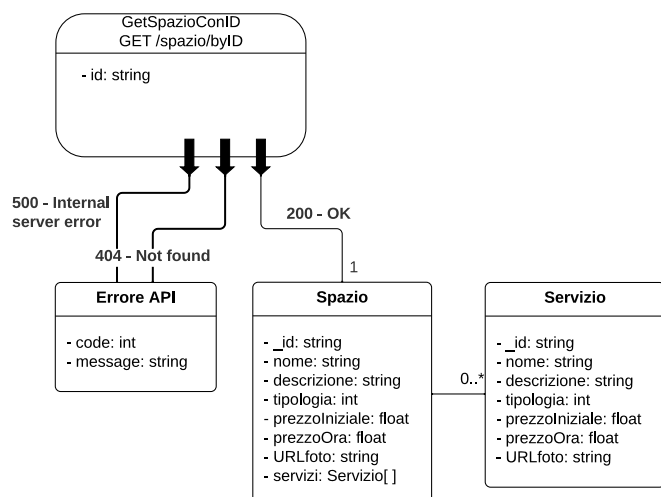
Grazie a una specifica funzionalità di MongoDB, il vettore di ID dei servizi disponibili per l'utilizzo in questo spazio viene sostituito direttamente con il vettore dei servizi corrispondenti, evitando così di doverli richiedere uno ad uno manualmente.

17. Cercare uno spazio tramite il suo ID

- METODO: **GET**
- URI: **/spazio/byID**

L'unico parametro univoco di uno spazio è il suo ID; quindi, l'unica risorsa per ottenere uno specifico spazio è questa.

Anche in questo caso viene restituito il vettore di servizi al posto del vettore di ID.

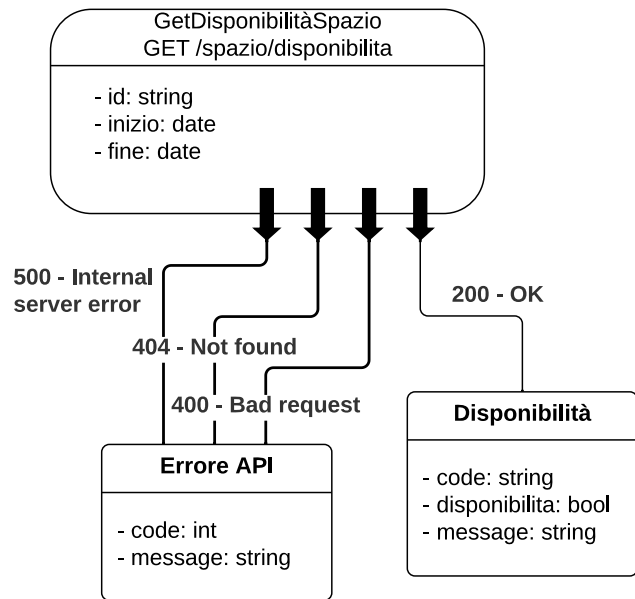


18. Verificare la disponibilità di uno spazio

- METODO: **GET**
- URI: **/spazio/disponibilita**

Questa risorsa funziona allo stesso modo della sua corrispondente per i servizi, verificando la disponibilità dello spazio in un certo periodo di tempo.

NB: in questo caso la disponibilità si riferisce solo allo spazio, i servizi ad esso collegati non vengono controllati, dato che non è detto che vengano aggiunti anch'essi alla prenotazione.



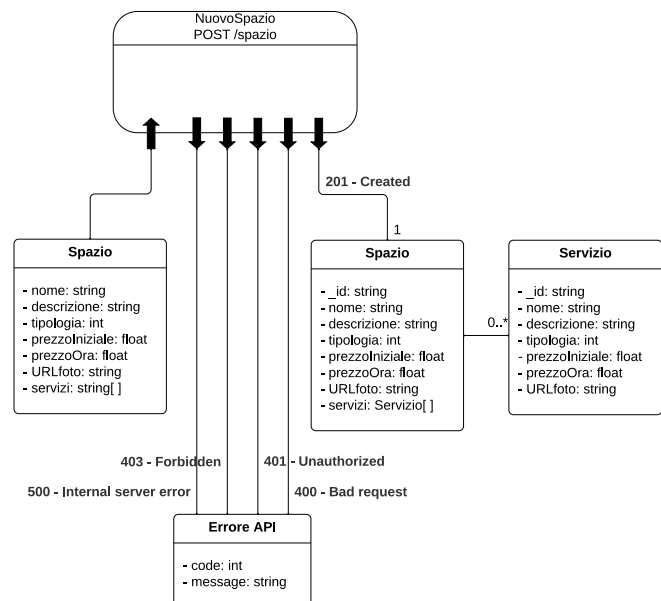
19. Inserire un nuovo spazio

- METODO: **POST**
- URI: **/spazio**

Questa risorsa, riservata agli utenti di segreteria, permette di inserire un nuovo spazio nel sistema.

Durante l'inserimento, è possibile specificare direttamente in un vettore gli ID dei servizi (già inseriti nel sistema) da associare allo spazio.

Se invece di devono inserire nuovi servizi, si deve utilizzare la [risorsa riservata per il loro inserimento](#) e poi [modificare lo spazio](#) aggiungendo il loro ID al vettore di servizi associati.

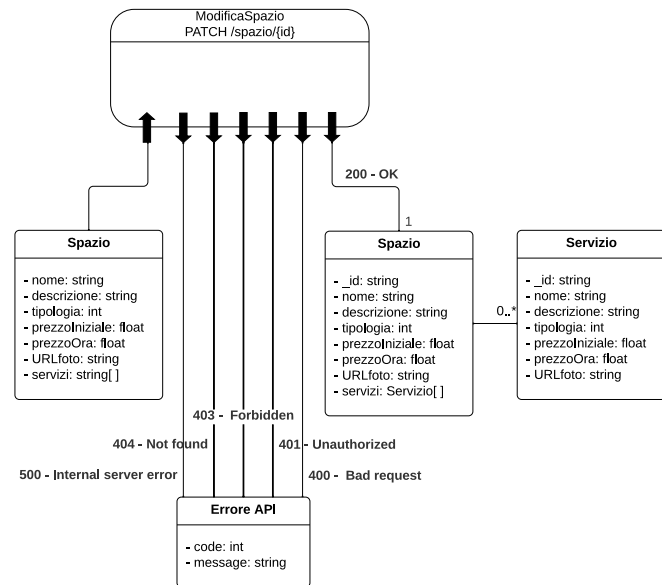


20. Modificare i dati di uno spazio

- METODO: **PATCH**
- URI: **/spazio/{id}**

Questa risorsa modifica uno spazio già inserito nel sistema.

Come già specificato nella descrizione delle risorse precedenti, tramite questa risorsa si possono anche associare o rimuovere servizi che possono essere aggiunti alla prenotazione di questo spazio.



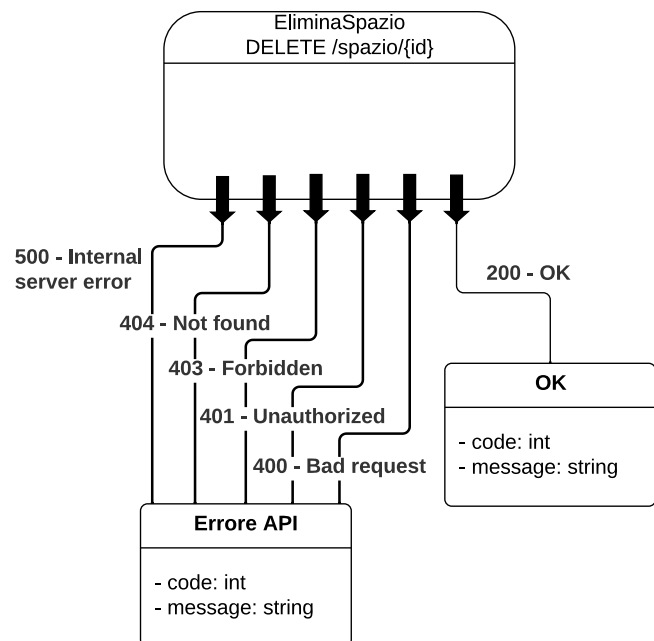
21. Eliminare uno spazio

- METODO: **DELETE**
- URI: **/spazio/{id}**

Questa risorsa rimuove uno spazio dal sistema.

La rimozione di uno spazio non comporta la rimozione dei servizi a lui associati, dato che potrebbero essere associati anche ad altri spazi ancora presenti.

Per eliminare anche i servizi, bisognerà chiamare la [risorsa per eliminare un servizio](#).



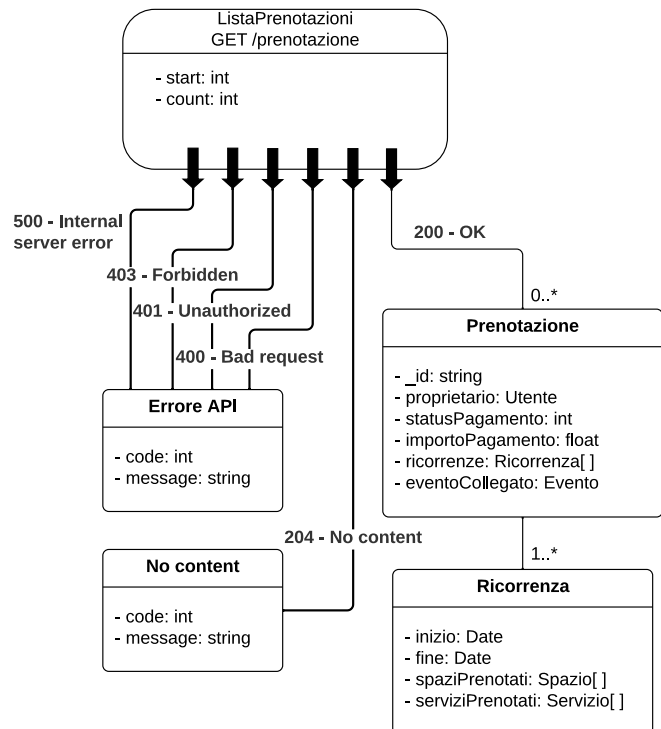
Prenotazioni

22. Elenco di prenotazioni

- METODO: **GET**
- URI: **/prenotazione**

Questa risorsa recupera una serie di prenotazioni dal sistema con il metodo start-count già descritto in precedenza.

Siccome non viene fatta una distinzione per utente, l'utilizzo di questa risorsa è riservato agli utenti di segreteria.

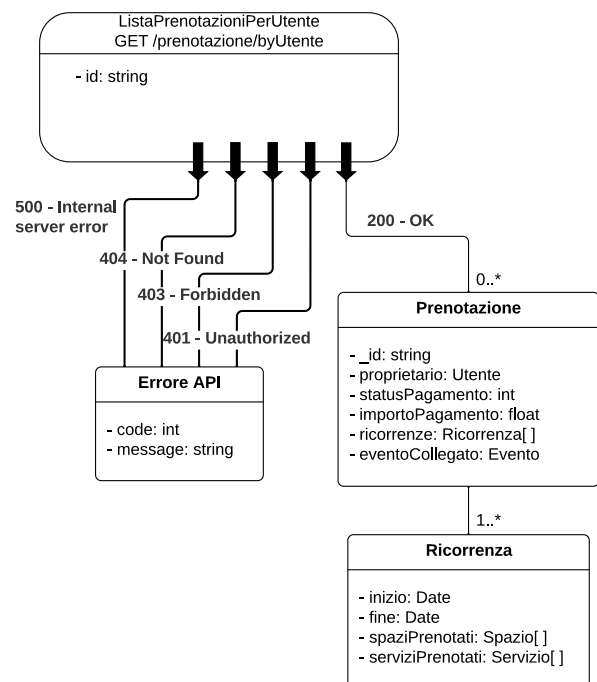


23. Elenco delle prenotazioni di un utente

- METODO: **GET**
- URI: **/prenotazione/byUtente**

Un modo disponibile anche agli utenti normali di recuperare prenotazioni è rappresentato da questa risorsa, che specificato l'ID di un utente restituisce le sue prenotazioni e le ricorrenze associate.

Si precisa che un utente ha il permesso di ottenere solo le sue prenotazioni. Nel caso l'ID specificato fosse diverso da quello dell'utente, viene restituito l'errore 403 - Forbidden.

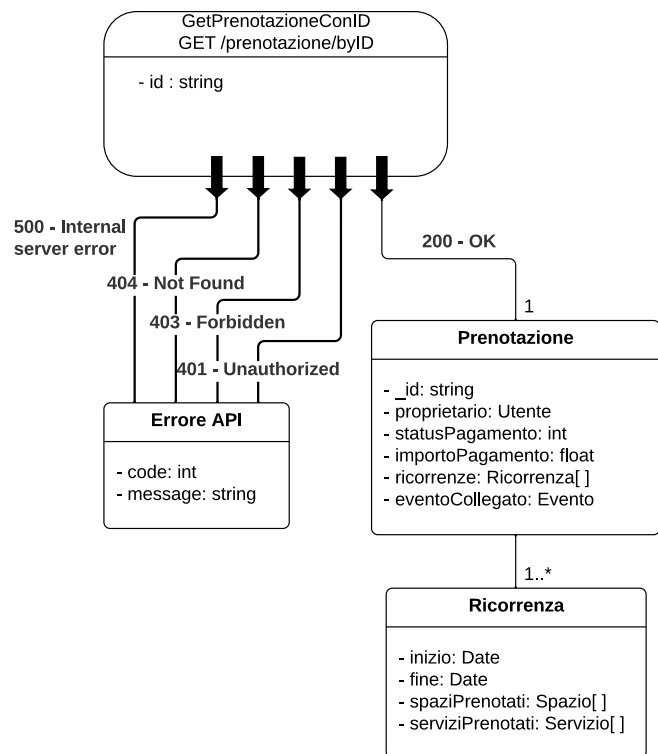


24. Cercare una prenotazione tramite il suo ID

- METODO: **GET**
- URI: **/prenotazione/byID**

Per ottenere una singola prenotazione (e le ricorrenze ad essa collegate) si può utilizzare questa risorsa.

Rimangono valide le restrizioni definite nella [risorsa precedente](#), ovvero il fatto che un utente normale può avere accesso solo alle sue prenotazioni.

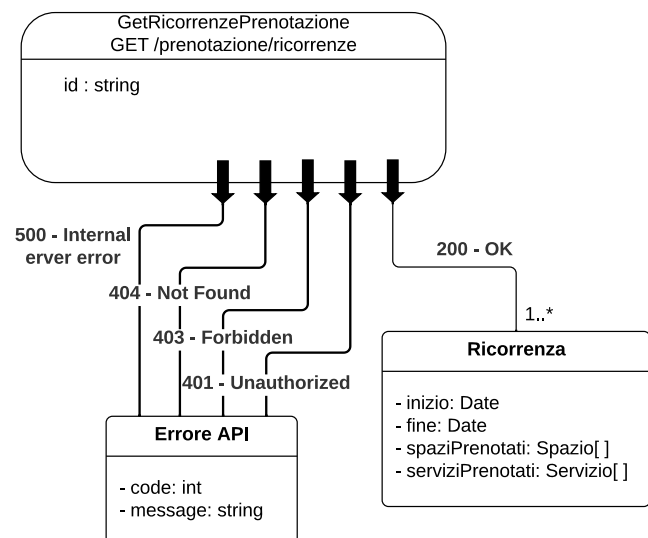


25. Ottenere le ricorrenze di una prenotazione

- METODO: **GET**
- URI: **/prenotazione/ricorrenze**

Se, avendo l'ID di una prenotazione, si desidera ottenere l'elenco delle sure ricorrenze si può invocare questa risorsa.

Durante lo sviluppo questa risorsa ha perso il suo motivo di utilizzo a causa della funzionalità di MongoDB di poter sostituire i vettori di ID con i vettori degli elementi corrispondenti.



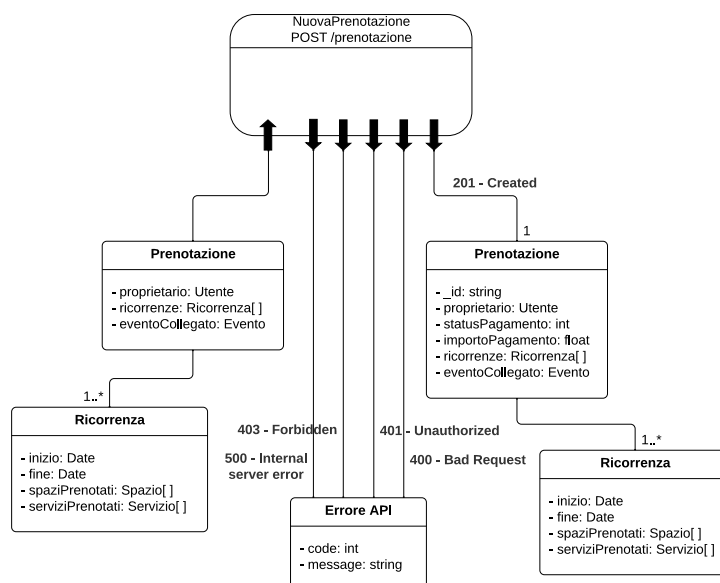
26. Inserire una nuova prenotazione

- METODO: **POST**
- URI: **/prenotazione**

Questa risorsa viene utilizzata da un utente per inserire una nuova prenotazione nel sistema.

Assieme all'oggetto prenotazione devono essere specificate le ricorrenze (una o più) associate alla prenotazione.

Per questo motivo, non è stata definita una risorsa per inserire una singola ricorrenza.

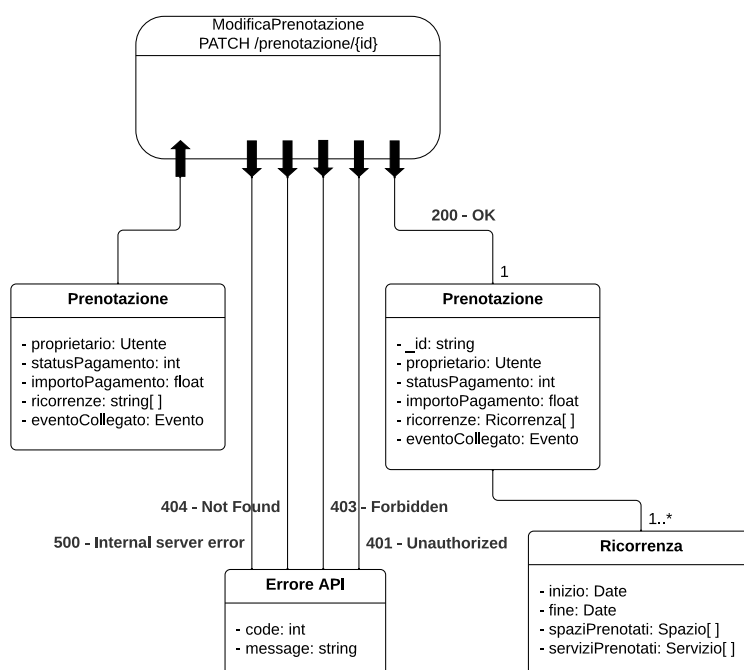


27. Modificare i dati di una prenotazione

- METODO: **PATCH**
- URI: **/prenotazione/{id}**

La modifica di una prenotazione già inserita nel sistema avviene tramite questa risorsa.

In questo caso la modifica delle risorse viene effettuata chiamando un'altra risorsa, riservata alla [modifica di una specifica ricorrenza](#).

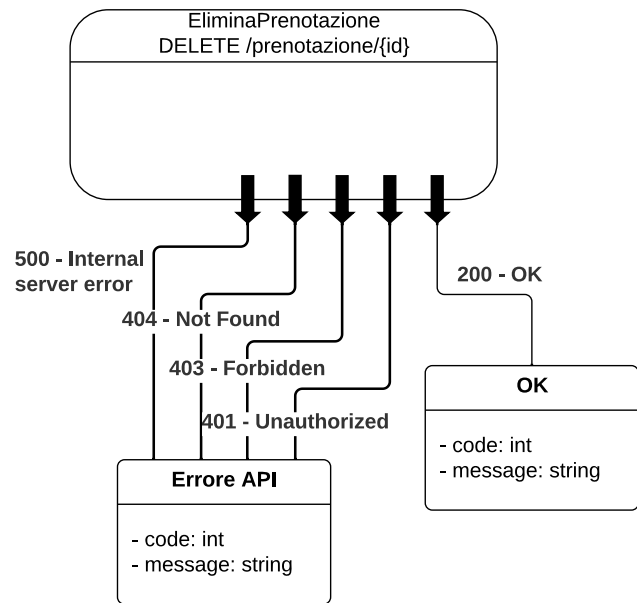


28. Eliminare una prenotazione

- METODO: **DELETE**
- URI: **/prenotazione/{id}**

La rimozione di una prenotazione, e di tutte le sue ricorrenze, viene effettuata con questa risorsa.

Siccome le ricorrenze sono presenti nell'oggetto prenotazione, è sufficiente specificare l'ID di quest'ultima.



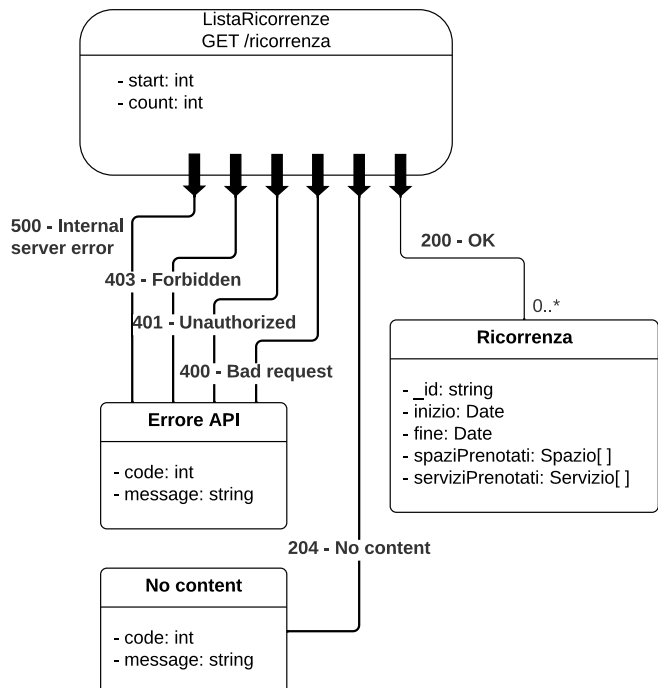
Ricorrenze

29. Elenco di ricorrenze

- METODO: **GET**
- URI: **/ricorrenza**

Per ottenere una serie di ricorrenze, non legate a una specifica prenotazione, si può utilizzare questa risorsa.

Il meccanismo di ottenimento è il solito start-count, già utilizzato per ottenere le liste degli elementi descritti in precedenza.

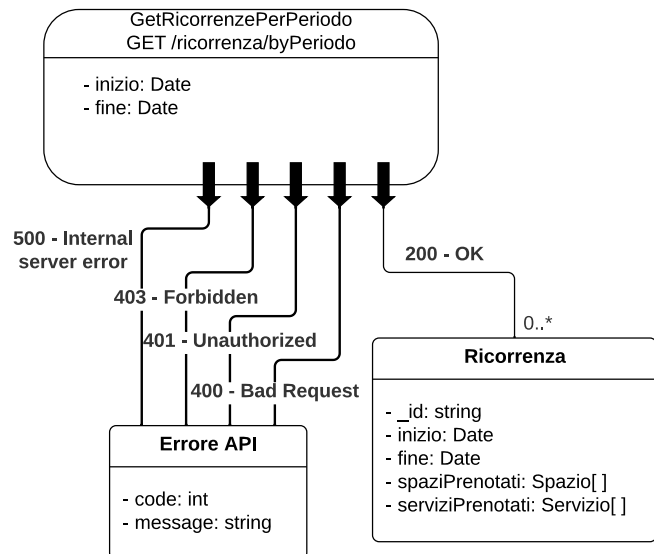


30. Elenco delle ricorrenze che si svolgono in un certo periodo

- METODO: **GET**
- URI: **/ricorrenza/byPeriodo**

Un modo di ottenere le ricorrenze molto più utile è quello di specificare il periodo nel quale avvengono.

Infatti, quando ad esempio si deve verificare se uno spazio è già stato prenotato da un'altra prenotazione, basta verificare che non sia presente nelle ricorrenze che accadono nel periodo considerato.

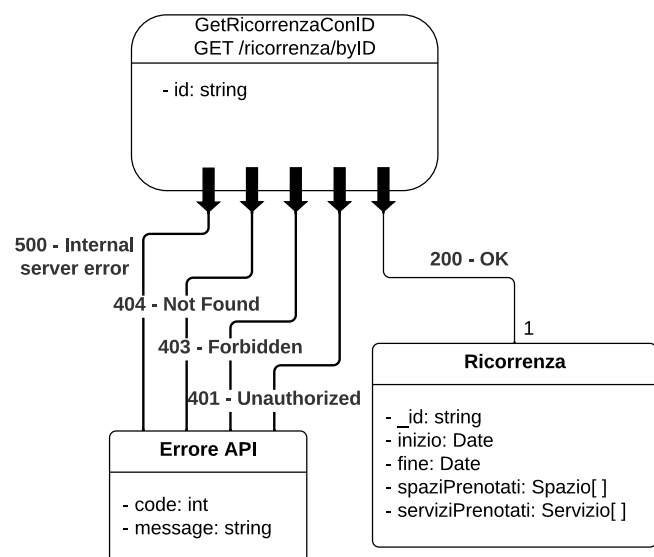


31. Cercare una ricorrenza tramite il suo ID

- METODO: **GET**
- URI: **/ricorrenza/byID**

Nel caso in cui si dovessero recuperare i dati di una ricorrenza specifica, può essere utilizzata questa risorsa.

Come già menzionato in precedenza, l'utilità di questa risorsa è ridotta dal momento in cui si utilizza la funzionalità di MongoDB di inserire gli elementi a posto del loro ID all'interno di un elemento contenitore (in questo caso, la prenotazione).

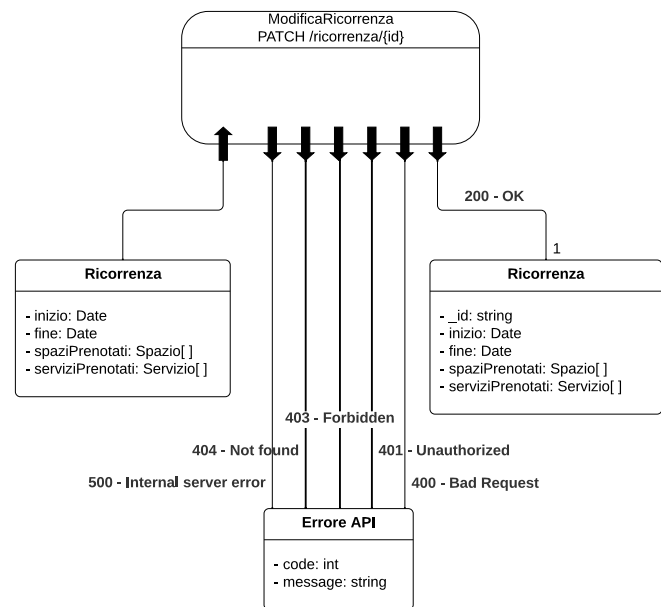


32. Modificare i dati di una ricorrenza

- METODO: **PATCH**
- URI: **/ricorrenza/{id}**

Questa risorsa permette di modificare i dati di una singola ricorrenza.

Si precisa che la prenotazione alla quale questa ricorrenza è collegata non può essere cambiata da questa risorsa.

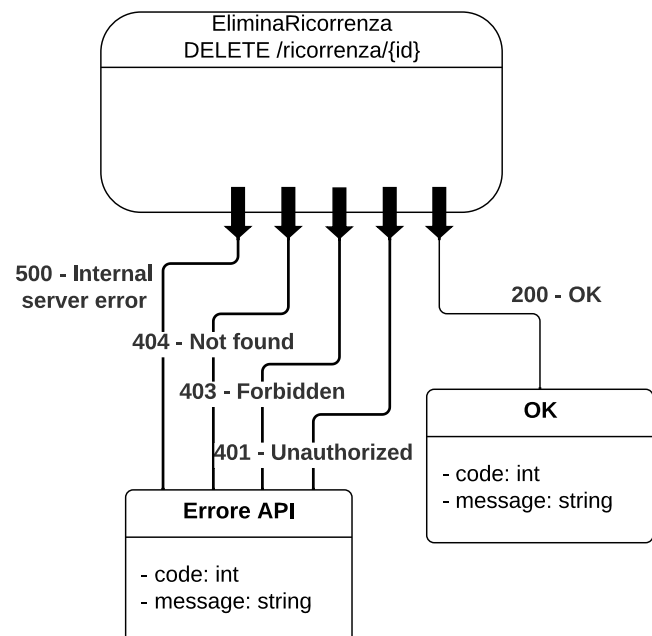


33. Eliminare una ricorrenza

- METODO: **DELETE**
- URI: **/ricorrenza/{id}**

Questa risorsa elimina una singola ricorrenza dal sistema (rappresentata dall'ID inserito nell'URI).

E' evidente che gli spazi e i servizi associati a questa ricorrenza non vengono in alcun modo eliminati, in quanto possono essere utilizzati da altre ricorrenze ancora presenti nel sistema.



Sviluppo delle API

In questo capitolo viene descritto lo sviluppo e il funzionamento delle API illustrate in precedenza. Nei casi in cui la lunghezza del codice lo consente, viene riportato anche uno screenshot del controller dell'API.

Utente

1. Elenco di utenti

L'API per ottenere la lista degli utenti è stata progettata con l'idea di permettere un caricamento dei dati progressivo, permettendo quindi di ottenere le varie parti della lista separatamente. Questo viene di grande aiuto quando gli elementi memorizzati sono in gran numero e restituirli tutti insieme diventerebbe un'operazione onerosa e probabilmente non necessaria.

Come si vede nello screenshot seguente, l'API accetta in input i parametri start e count, che vengono dati in input alle funzioni di mongoose skip() e limit().

```
//restituisce tutti gli utenti
const listaUtenti = (req, res) => {
  //prelevo il numero di utenti da restituire e da quale di essi iniziare
  let count = parseInt(req.query.count || 50);
  let start = parseInt(req.query.start || 0);

  if (isNaN(count) || isNaN(start)) {
    return res.status(400).json({ message: "start e count devono essere interi" });
  }
  if (count < 1 || start < 0) {
    return res.status(400).json({
      message: "start deve essere positivo e count maggiore di 0",
    });
  }

  //cerco gli spazi dando un filtro vuoto per ottenerli tutti
  Utente.find({}, (err, data) => {
    if (err) {
      return res.status(500).json({ code: 500, message: err }); //risposta in caso di errore
    }
    if (!data) return res.status(204).json({ code: 204, message: "Nessun utente trovato" });
    return res.status(200).json(data); //restituisco i dati di tutti gli spazi
  })
  .skip(start)
  .limit(count);
};
```

2. Login

L'API per eseguire il login preleva l'e-mail e la password dalla richiesta, cerca l'utente nel database fornendo l'e-mail come filtro e verifica che la password corrisponda a quella criptata nel database con l'aggiunta del salt.

Se i dati sono corretti viene generato un nuovo token con la funzione generaToken().

```
//login dell'utente
const loginUtente = (req, res) => {
  if (!req.body) return res.status(400).json({ code: 400, message: "Body mancante" });

  //prendo l'email dell'utente dai parametri della richiesta
  let email = req.body.email;
  //prendo la password dell'utente dai parametri della richiesta
  let password = req.body.password;

  //se non trovo email e password nella richiesta rispondo con bad request
  if (!email || !password) {
    return res.status(400).json({ message: "Specificare email e password" });
  }

  //cerco e restituisco l'utente con quella email e quella password
  Utente.findOne({ email: email }, (err, data) => {
    if (err)
      return res.status(500).json({ code: 500, message: err });
    if (!data)
      return res.status(404).json({ code: 404, message: "Email errata" });
    if (data.password !== hashPassword(password, data.salt))
      return res.status(404).json({ code: 404, message: "Password invalida" });
    if (!data.confermaAccount)
      return res
        .status(400)
        .json({ code: 400, message: "L'utente non ha confermato l'account" });

    token = generaToken(data);
    return res.status(200).json({ token: token });
  });
};
```

3. Registrazione

La registrazione di un utente preleva i dati dal body della richiesta POST, effettua alcuni controlli sui dati inseriti, cripta la password con l'aggiunta di un salt generato casualmente e salva le informazioni nel database.

4. Conferma della registrazione

La conferma della registrazione è un'API molto semplice. Dopo aver controllato che l'utente con il dato ID esista effettivamente, va a modificare il valore confermaAccount a true.

Questo permetterà all'utente di effettuare il login con successo.

```
const confermaUtente = (req, res) => {
  const id = req.query.id;
  if (!id) {
    return res.status(400).json({ code: 400, message: "Parametro mancante: id" });
  }
  if (!ObjectId.isValid(id)) {
    return res.status(400).json({ code: 400, message: "id invalido" });
  }

  Utente.findOne({ _id: ObjectId(id) })
    .then((data) => {
      if (!data) {
        res.status(404).json({
          code: 404,
          message: `L'id '${id}' non corrisponde a nessun utente`,
        });
      }
      data.confermaAccount = true;
      data.save()
        .then((data) => {
          return res.status(200).json({
            code: 200,
            message: `L'utente con l'id '${id}' è stato confermato`,
          });
        })
        .catch((err) => {
          return res.status(500).json({ code: 500, message: err });
        });
    })
    .catch((err) => {
      return res.status(500).json({ code: 500, message: err });
    });
};
```

5. Cercare un utente tramite la sua e-mail

Tramite l'email specificata nella richiesta, viene richiesto al database di trovare un solo utente (findOne()) che possiede l'e-mail stessa. Per farlo viene dato in input alla funzione il filtro sul campo email.

Le API basate sulla ricerca tramite parametro univoco sono tutte molto simili; quindi, riportiamo lo screenshot solo della ricerca tramite e-mail.

```
//restituisce le info di uno utente data l'email
const getUtenteConEmail = (req, res) => {
  //prendo l'email dell'utente dai parametri della richiesta
  if (!req.query || !req.query.email) {
    return res.status(400).json({ code: 400, message: "Email non specificata" });
  }

  let email = req.query.email;
  if (req.utente.livello >= 2 || req.utente.email == email) {
    //cerco e restituisco l'utente con quella email
    Utente.findOne({ email: email }, (err, data) => {
      if (err) return res.status(500).json({ code: 500, message: err });
      if (!data)
        return res.status(404).json({
          code: 400,
          message: "L'utente richiesto non esiste",
        });
      return res.status(200).json(data); //se trovo l'oggetto lo restituisco
    });
  } else {
    return res.status(403).json({ code: 403, message: "Utente non autorizzato" });
  }
};
```

6. Cercare un utente tramite un token di accesso

Questa API richiede la decodifica del token in ingresso e cerca nel database l'utente per il quale è stato generato il token. La struttura del codice è molto simile a quello riportato per l'[API di ricerca tramite e-mail](#).

7. Cercare un utente tramite il suo ID

Questo metodo di ricerca si rivela addirittura più semplice degli altri dato che gli ID vengono utilizzati anche come parametro univoco nel database. Sarà dunque sufficiente fornire il filtro con l'ID in input alla funzione di ricerca di mongoose.

Per aggiungere un minimo di variazione abbiamo deciso di riservare l'uso di questa API agli utenti di segreteria.

La struttura del codice è tuttavia molto simile a quello riportato per l'[API di ricerca tramite e-mail](#).

8. Modifica del profilo

La modifica del profilo segue un processo molto simile a quello della registrazione. L'unica parte aggiuntiva prevede che nella risposta, insieme ai dati dell'utente modificato, venga restituito anche un nuovo token in modo che l'utente possa continuare a utilizzare l'applicazione senza dover eseguire nuovamente il login ogni volta che apporta modifiche al proprio profilo.

9. Cancellazione di un utente

L'API per cancellare un utente, dopo aver effettuato controlli sui privilegi dell'utente richiedente, effettua una ricerca nel database per assicurarsi che esso esista e in

seguito, tramite la funzione `findOneAndDelete()` rimuove tutte le informazioni legate all'utente.

In aggiunta, se la foto profilo dell'utente era stata impostata (e quindi è diversa da quella di default), viene rimossa dal filesystem per evitare possibili sprechi di memoria.

```
//elimina un utente dato l'ID
const cancellaUtente = (req, res) => {
  if (!req.params || !req.params.id)
    return res.status(400).json({ code: 400, message: "id non specificato" });

  //prendo l'ID dell'utente dai parametri della richiesta
  const id = req.params.id;

  if (!ObjectId.isValid(id)) return res.status(400).json({ code: 400, message: "id invalido" });

  if (req.utente.livello < 2 && req.utente._id !== id)
    return res.status(403).json({ code: 403, message: "Utente non autorizzato" });

  //prima di eliminarlo verifico che l'utente esista
  Utente.findOne({ _id: ObjectId(id) }, (err, data) => {
    if (err) return res.status(500).json({ code: 500, message: err });
    if (!data)
      return res.status(404).json({ code: 404, message: "L'utente richiesto non esiste" });

    //cancello la foto profilo dell'utente
    if (data.URLfoto !== "/images/utenti/default.png") {
      const pathFoto = data.URLfoto.substring(1);
      fs.unlinkSync(pathFoto);
    }

    //cerco ed elimino l'utente con quell'ID
    Utente.findOneAndDelete({ _id: ObjectId(id) }, (err) => {
      if (err) return res.status(500).json({ code: 500, message: err });
      return res.status(200).json({
        code: 200,
        message: "Utente eliminato correttamente",
      });
    });
  });
};
```

Servizi

10. Elenco di servizi

Questa API ha lo stesso funzionamento di quella per [ottenere un elenco di utenti](#), descritta in modo più dettagliato precedentemente.

11. Cercare un servizio tramite il suo ID

Anche in questo caso, la ricerca tramite ID si rivela molto semplice e diretta, per un esempio sulla struttura del codice fare riferimento all'API di [ricerca di un utente tramite email](#).

12. Verificare la disponibilità di un servizio

La disponibilità di un servizio viene calcolata allo stesso modo di quanto fatto per uno spazio, la descrizione di [quell'API](#) riporta anche uno screenshot del codice.

13. Inserire un nuovo servizio

Come è possibile vedere dall'immagine in basso, l'inserimento di un nuovo servizio prevede il prelievo dei dati dal body della richiesta, viene controllato che il nome sia esistente (unico parametro obbligatorio), e viene creato un nuovo oggetto a partire dal modello del servizio, che viene infine salvato nel database.

Come già definito nella descrizione delle risorse, questa API non si occupa di collegare il nuovo servizio a degli spazi, per farlo si utilizza l'[API di modifica di uno spazio](#).

```
//inserimento di un nuovo servizio
const crea = (req, res) => {
  if (!req.body) {
    return res.status(400).json({ code: 400, message: "Dati mancanti" });
  }

  const nome = req.body.nome;
  const descrizione = req.body.descrizione;
  const tipologia = req.body.tipologia;
  const prezzoIniziale = req.body.prezzoIniziale;
  const prezzoOra = req.body.prezzoOra;
  const foto = req.body.foto;

  if (!nome) {
    return res.status(400).json({
      code: 400,
      message: "Parametro mancante: nome",
    });
  }

  const nuovoServizio = new Servizio({
    nome: nome,
    descrizione: descrizione,
    tipologia: tipologia,
    prezzoIniziale: prezzoIniziale,
    prezzoOra: prezzoOra,
    URLfoto: foto,
  });

  nuovoServizio
    .save()
    .then((data) => {
      return res.status(201).json(data);
    })
    .catch((err) => {
      return res.status(500).json({ code: 500, message: err });
    });
};
```


14. *Modificare i dati di un servizio*

Anche in questo caso, la modifica di un servizio riflette la struttura del codice usato per l'[inserimento](#), già riportata sopra.

15. *Eliminare un servizio*

La cancellazione di un servizio si occupa semplicemente di eliminare dal database il servizio identificato dall'id specificato nell'URI utilizzato per invocare questa API.

Spazi

16. *Elenco di spazi*

Questa API ha lo stesso funzionamento di quella per [ottenere un elenco di utenti](#), descritta in modo più dettagliato precedentemente.

17. *Cercare uno spazio tramite il suo ID*

Come per i servizi, la ricerca tramite ID si rivela molto semplice e diretta, per un esempio sulla struttura del codice fare riferimento all'API di [ricerca di un utente tramite email](#).

18. *Verificare la disponibilità di uno spazio*

Questa API ha un compito particolare, deve assicurarsi che lo spazio specificato non sia stato già prenotato nel periodo fornito in input.

Per farlo, dopo aver ottenuto lo spazio dal database, si ottengono tutte le ricorrenze che si svolgono, anche in parte, nel periodo compreso tra le date inizio e fine. Se il numero di ricorrenze ottenute è 0, è sicuro che questo spazio è disponibile per essere prenotato, altrimenti no.

```
//restituisce lo status dello spazio nel periodo indicato
const getDisponibilitaPeriodo = (req, res) => {
  //prendo l'ID dello spazio e le date dai parametri della richiesta
  let id = req.query.id;
  let inizio = req.query.inizio;
  let fine = req.query.fine;
  //cerco lo spazio con quell'ID
  Spazio.findOne({ _id: ObjectId(id) }, (err, data) => {
    if (err || !data) {
      return res.status(404).json({ code: 404, message: "Lo spazio richiesto non esiste" });
    } else {
      //prelevo tutte le ricorrenze nel periodo richiesto e vedo se lo spazio compare tra quelli prenotati
      const dataInizio = new Date(inizio);
      const dataFine = new Date(fine);

      Ricorrenza.count({
        inizio: { $lt: dataFine },
        fine: { $gt: dataInizio },
        spaziPrenotati: { $elemMatch: { $eq: ObjectId(id) } },
      })
        .then((numero) => {
          if (numero == 0) {
            return res.status(200).json({
              code: 200,
              disponibilita: true,
              message: `Lo spazio ${id} è disponibile nel periodo tra ${inizio} e ${fine}`,
            });
          } else {
            return res.status(200).json({
              code: 200,
              disponibilita: false,
              message: `Lo spazio ${id} NON è disponibile nel periodo tra ${inizio} e ${fine}`,
            });
          }
        })
        .catch((err) => {
          return res.status(500).json({ code: 500, message: err });
        });
    }
  });
};
```

19. Inserire un nuovo spazio

Il codice per inserire un nuovo spazio è molto simile a quello presentato per l'inserimento di un servizio.

L'unica differenza risiede nel fatto che durante l'inserimento di un servizio si può specificare in un vettore gli ID dei servizi da associare allo spazio. Questo vettore viene estratto dal body come JSON e aggiunto ai campi da salvare nel database.

20. Modificare i dati di uno spazio

Anche in questo caso, la modifica di uno spazio riflette la struttura del codice usato per l'inserimento, già riportata sopra.

21. Eliminare uno spazio

L'eliminazione di uno spazio risulta essere ancora più semplice dell'eliminazione di un servizio grazie al fatto che, quando uno spazio viene eliminato, i servizi non vengono eliminati con esso.

Prenotazioni

22. Elenco di prenotazioni

Questa API ha lo stesso funzionamento di quella per [ottenere un elenco di utenti](#), descritta in modo più dettagliato precedentemente.

23. Elenco delle prenotazioni di un utente

Per le prenotazioni è disponibile un'altra API per ottenere una serie di prenotazioni, basata sull'utente che le ha create. In questo modo, quando un utente è loggato può andare a vedere le prenotazioni che ha fatto in precedenza.

Per filtrare le prenotazioni si utilizza quindi il campo proprietario, che contiene l'ID dell'utente che ha creato la prenotazione, come visibile nello screenshot seguente.

```
// get prenotazioni di un utente
const getPrenotazioniUtente = (req, res) => {
  //prendo l'id dell'utente dalla richiesta
  let id = req.query.id;

  //verifico che quell'utente abbia accesso alle prenotazioni richieste
  if (req.utente.livello < 2 && req.utente._id !== id)
    return res.status(403).json({ code: 403, message: "Utente non autorizzato" });

  Utente.findOne({ _id: ObjectId(id) }, (err, data) => {
    if (err) return res.status(500).json({ code: 500, message: err });
    if (!data)
      return res.status(404).json({
        code: 404,
        message: "L'utente non esiste",
      });
    // trova le prenotazioni dell'utente e le ritorna
    Prenotazione.find({ proprietario: id })
      .populate(populateSpazi)
      .populate(populateRicorrenze)
      .exec((err, data) => {
        if (err) return res.status(500).json({ code: 500, message: err });
        return res.status(200).json(data);
      });
  });
};
```

Insieme alla prenotazione, vengono per comodità restituite anche le sue ricorrenze e gli spazi prenotati. Per farlo sono stati definiti due oggetti (riportati nello screenshot seguente) che, passandoli in input alla funzione `populate()`, sostituiscono gli ID con l'oggetto corrispondente.

```
// oggetti json per popolare (fare left join) ai document spazi e servizi,  
// da passare come parametro a .populate  
const populateSpazi = {  
  path: "ricorrenze",  
  populate: {  
    path: "spaziPrenotati",  
    model: "Spazio",  
  },  
};  
  
const populateRicorrenze = {  
  path: "ricorrenze",  
  populate: {  
    path: "serviziPrenotati",  
    model: "Servizio",  
  },  
};
```

24. Cercare una prenotazione tramite il suo ID

Come per le altre risorse, la ricerca tramite ID si rivela molto semplice e diretta, per un esempio sulla struttura del codice fare riferimento all'API di [ricerca di un utente tramite email](#).

25. Ottenere le ricorrenze di una prenotazione

Questa API restituisce la lista di ricorrenze di una prenotazione. Da quando è stata utilizzata la funzione `populate()` per sostituire gli ID con gli oggetti corrispondenti (vedi API di [elenco delle prenotazioni](#)), questa API si è rivelata non più molto utile.

26. Inserire una nuova prenotazione

Questa API inserisce una prenotazione nel sistema. Il formato dei dati in input prevede che siano presenti anche tutte le ricorrenze da associare alla prenotazione.

Questa scelta è stata presa per permettere la massima flessibilità possibile sulle informazioni delle ricorrenze. Infatti, al posto di un singolo parametro sulla periodicità o il numero di ricorrenze, con questo metodo le ricorrenze possono accadere a intervalli irregolari e interessare spazi e servizi diversi.

Dal punto di vista del codice, vengono create prima tutte le ricorrenze e si tengono i loro ID, dopo si inserisce la prenotazione con i suoi dati generali e il vettore di ID delle ricorrenze. Durante questa operazione viene anche calcolato il prezzo della prenotazione, in base al prezzo iniziale e orario di spazi e servizi e al tempo di utilizzo degli stessi.

27. Modificare i dati di una prenotazione

La modifica di una prenotazione mantiene la stessa flessibilità dell'inserimento. Dunque, è possibile specificare una serie di ricorrenze associate alla prenotazione da modificare.

28. Eliminare una prenotazione

L'eliminazione di una prenotazione è diversa dall'eliminazione degli altri oggetti in quanto deve rimuovere anche le ricorrenze ad essa associate.

Quindi, dopo aver verificato che la prenotazione da eliminare effettivamente esista, vengono cancellate le sue ricorrenze e per ultima la prenotazione. Il codice è strutturato come visibile nello screenshot seguente.

```
// elimina prenotazione
const eliminaPrenotazione = (req, res) => {
  let id = req.params.id;

  Prenotazione.findOne({ _id: ObjectId(id) }, (err, data) => {
    if (err) return res.status(500).json({ code: 500, message: err });
    if (!data)
      return res.status(404).json({
        code: 404,
        message: "La prenotazione non esiste",
      });

    if (req.utente.livello < 2 && !req.utente._id.equals(data.proprietario)){
      console.log(req.utente._id);
      console.log(data.proprietario);
      console.log(typeof req.utente._id);
      return res.status(403).json({ code: 403, message: "Utente non autorizzato" });
    }

    const ricorrenze = data.ricorrenze;
    Ricorrenza.deleteMany({ _id: { $in: ricorrenze } })
      .then((data) => {
        Prenotazione.findOneAndDelete({ _id: ObjectId(id) }, (err, data) => {
          if (err) return res.status(500).json({ code: 500, message: err });
          return res.status(200).json({
            code: 200,
            message: "Prenotazione eliminata correttamente",
          });
        });
      })
      .catch((err) => {
        return res.status(500).json({ code: 500, message: err });
      });
  });
};
```

Ricorrenze

Come è già stato menzionato in precedenza, abbiamo deciso di non implementare un'API per l'inserimento di una ricorrenza singola in quanto riteniamo che le ricorrenze siano oggetti strettamente legati alle prenotazioni di cui fanno parte. Per questo, l'inserimento di ricorrenze nel sistema è effettuato dall'[API di creazione di una nuova prenotazione](#).

29. Elenco di ricorrenze

Questa API ha lo stesso funzionamento di quella per [ottenere un elenco di utenti](#), descritta in modo più dettagliato precedentemente.

30. *Elenco delle ricorrenze che si svolgono in un certo periodo*

Per le ricorrenze è disponibile un'altra API per ottenere una serie di ricorrenze, basata sul periodo nel quale si svolgono. Questo potrebbe risultare utile se il frontend avesse bisogno di effettuare un controllo preliminare su quali spazi sono occupati in un dato periodo.

Per filtrare le prenotazioni si utilizzano quindi due campi inizio e fine, inseriti in una query da fornire alla funzione `find()`, che restituisce le ricorrenze che si svolgono anche solo parzialmente nel periodo rappresentato.

```
// ottiene le ricorrenze in un intervallo di tempo
const getRicorrenzePerPeriodo = (req, res) => {
  if (req.query.inizio > req.query.fine) {
    return res.status(400).json({
      code: 400,
      errore: "La data di inizio è maggiore della data di fine",
    });
  }

  // cerco tutte le ricorrenze che sono "sovrapposte" anche parzialmente
  // all'intervallo di tempo specificato in req.query (ad esempio se viene passato l'intervallo
  // da 12:00 a 16:00, viene ritornata anche la ricorrenza che va da 11:00 a 13:00 perchè è nell'intervallo specificato)
  let query = {
    inizio: { $lt: req.query.fine },
    fine: { $gte: req.query.inizio },
  };

  Ricorrenza.find(query)
    .populate("spaziPrenotati")
    .populate("serviziPrenotati")
    .exec((err, data) => {
      if (err) {
        return res.status(500).json({ code: 500, errore: err });
      } else return res.status(200).json(data);
    });
};
```

31. *Cercare una ricorrenza tramite il suo ID*

Come per le altre risorse, la ricerca tramite ID si rivela molto semplice e diretta, per un esempio sulla struttura del codice fare riferimento all'API di [ricerca di un utente tramite email](#).

32. *Modificare i dati di una ricorrenza*

Nonostante non possa essere inserita una singola ricorrenza, è permesso modificarne una senza intaccare la prenotazione ad essa associata o le altre ricorrenze.

Dal punto di vista del codice, il funzionamento è comparabile a quello delle altre API di modifica degli oggetti memorizzati nel database.

33. *Eliminare una ricorrenza*

Anche l'eliminazione di una ricorrenza è simile a una classica eliminazione, l'unico aspetto da tenere presente è che eliminando una ricorrenza viene eliminato anche il suo ID presente nel vettore di ricorrenze della prenotazione associata.

Documentazione delle API

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

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

La pagina è stata definita accuratamente in modo da poter essere utilizzata per interagire con il sistema. Per farlo è sufficiente premere il pulsante "Try it out" relativo a una delle API, riempire i campi richiesti e premere il pulsante "Execute" per vedere la risposta del sistema.

In particolare, nella Figura 8 mostriamo la pagina web relativa alla documentazione che presenta tutte le 33 API sviluppate per l'applicazione, divise in base alla risorsa sulla quale operano. In fondo alla foto è possibile vedere anche la definizione dei modelli, che corrispondono alle strutture dati memorizzate nel database.



Progetto Gruppo T31 - IS2022 3.0.0

[Base URL: gruppo-t31-api.onrender.com/]

API del progetto del Gruppo T31 svolto durante il corso di Ingegneria del software - Anno Accademico 2022/23

Contact Luca Demattè - Referente gruppo T31
MIT

Schemas
HTTPS

utente

API per gli utenti del sistema

- POST** /utente/registrazione Creazione di un nuovo account utente
- POST** /utente/login Accesso al sistema
- GET** /utente/conferma Conferma registrazione dell'account
- GET** /utente Ottiene la lista degli utenti presenti nel sistema
- GET** /utente/byID Ottiene un utente tramite il suo ID
- GET** /utente/byEmail Ottiene un utente tramite la sua email
- GET** /utente/byToken Ottiene un utente tramite un token di accesso
- PATCH** /utente/{id} Modifica di un utente
- DELETE** /utente/{id} Eliminazione di un utente

spazio

API per gli spazi del sistema

- POST** /spazio Creazione di un nuovo spazio
- GET** /spazio Ottiene la lista degli spazi presenti nel sistema
- GET** /spazio/byID Ottiene uno spazio tramite il suo ID
- GET** /spazio/disponibilita Specificando una data di inizio e di fine, viene restituita la disponibilità (true o false) dello spazio richiesto nel periodo specificato
- PATCH** /spazio/{id} Modifica di uno spazio
- DELETE** /spazio/{id} Eliminazione di uno spazio

servizio

API per i servizi del sistema

- POST** /servizio Creazione di un nuovo servizio
- GET** /servizio Ottiene la lista dei servizi presenti nel sistema
- GET** /servizio/byID Ottiene un servizio tramite il suo ID
- GET** /servizio/disponibilita Specificando una data di inizio e di fine, viene restituita la disponibilità (true o false) del servizio richiesto nel periodo specificato
- PATCH** /servizio/{id} Modifica di un servizio
- DELETE** /servizio/{id} Eliminazione di un servizio

prenotazione

API per le prenotazioni del sistema

- POST** /prenotazione Creazione di una nuova prenotazione
- GET** /prenotazione Ottiene la lista delle prenotazioni presenti nel sistema
- GET** /prenotazione/byUtente Ottiene la lista delle prenotazioni di un dato utente
- GET** /prenotazione/byID Ottiene una prenotazione tramite il suo ID
- GET** /prenotazione/ricorrenze Ottiene la lista delle ricorrenze legate a una prenotazione
- PATCH** /prenotazione/{id} Modifica di una prenotazione
- DELETE** /prenotazione/{id} Eliminazione di una prenotazione e delle sue ricorrenze

ricorrenza

API per le ricorrenze del sistema

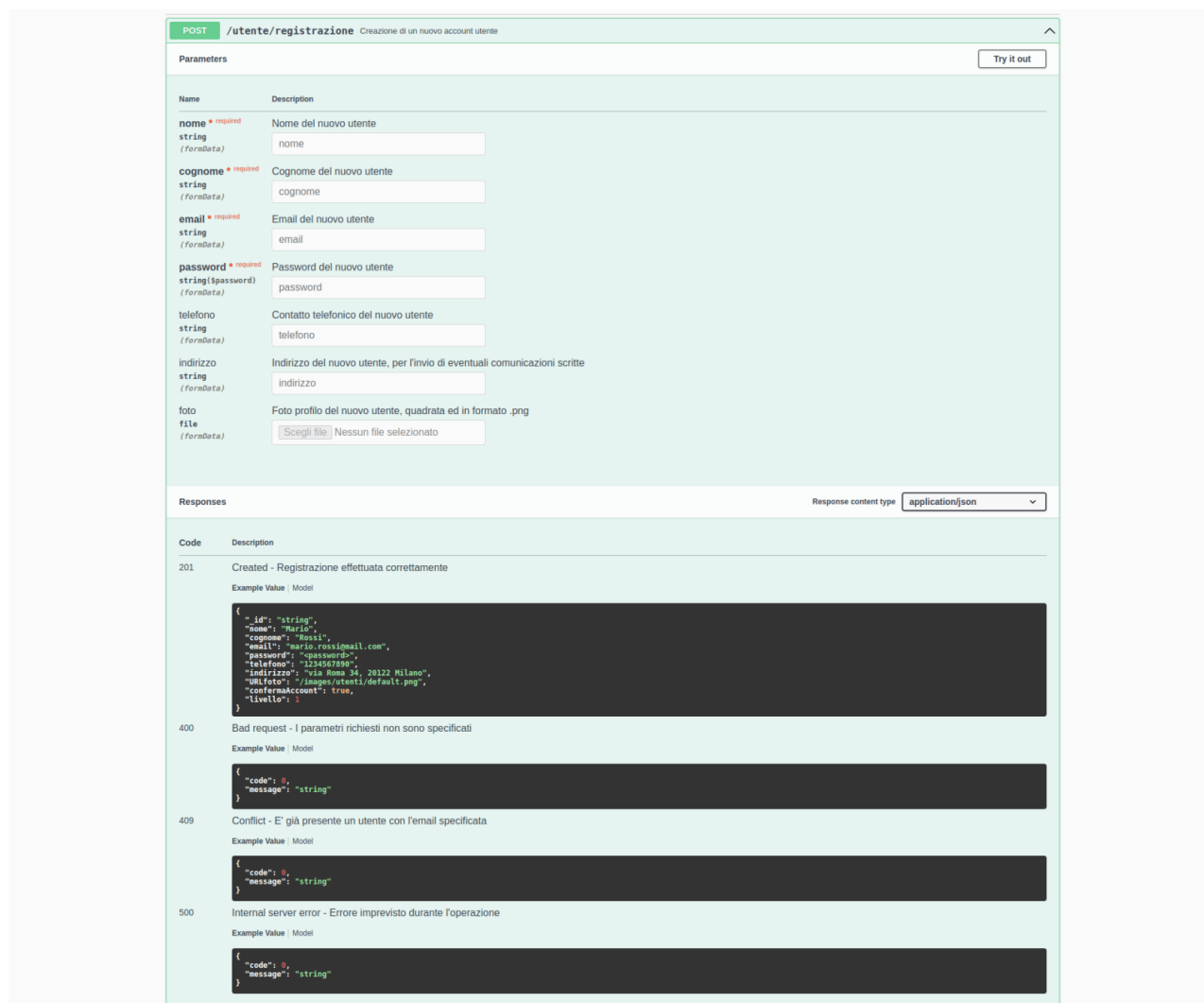
- GET** /ricorrenza Ottiene la lista delle ricorrenze presenti nel sistema
- GET** /ricorrenza/byID Ottiene una ricorrenza tramite il suo ID
- GET** /ricorrenza/byPeriodo Specificando una data di inizio e di fine, viene restituita la lista di ricorrenze comprese nel periodo specificato
- PATCH** /ricorrenza/{id} Modifica di una ricorrenza
- DELETE** /ricorrenza/{id} Eliminazione di una ricorrenza

Models

- Utente >
- Spazio >
- Servizio >
- Prenotazione >
- Ricorrenza >
- ErroreAPI >

Figura 8 – Screenshot completo della pagina generata da Swagger UI

Nella Figura 9 viene invece mostrata l'API **/utente/registrazione** espansa. È visibile il form compilabile e degli esempi di risposta dell'API con una veloce descrizione. Provando l'API verrà visualizzato anche il risultato dell'esecuzione in tempo reale.



POST /utente/registrazione Creazione di un nuovo account utente

Parameters

Name	Description
nome * required string (formData)	Nome del nuovo utente nome
cognome * required string (formData)	Cognome del nuovo utente cognome
email * required string (formData)	Email del nuovo utente email
password * required string(password) (formData)	Password del nuovo utente password
telefono string (formData)	Contatto telefonico del nuovo utente telefono
indirizzo string (formData)	Indirizzo del nuovo utente, per l'invio di eventuali comunicazioni scritte indirizzo
foto file (formData)	Foto profilo del nuovo utente, quadrata ed in formato .png Scegli file Nessun file selezionato

Responses

Response content type: application/json

Code	Description
201	Created - Registrazione effettuata correttamente Example Value Model <pre>{ "id": "string", "nome": "Mario", "cognome": "Rossi", "email": "mario.rossi@gmail.com", "password": "password", "telefono": "123456789", "indirizzo": "via Roma 34, 20122 Milano", "urlFoto": "/images/utenti/default.png", "confermaAccount": true, "livello": 1 }</pre>
400	Bad request - I parametri richiesti non sono specificati Example Value Model <pre>{ "code": 0, "message": "string" }</pre>
409	Conflict - E' già presente un utente con l'email specificata Example Value Model <pre>{ "code": 0, "message": "string" }</pre>
500	Internal server error - Errore imprevisto durante l'operazione Example Value Model <pre>{ "code": 0, "message": "string" }</pre>

Figura 9 – Screenshot della descrizione dell'API **/utente/registrazione**

L'endpoint da invocare per raggiungere la documentazione fornita da Swagger è:

<https://gruppo-t31-api.onrender.com/api-docs>

NB: a causa delle politiche del servizio di hosting, il primo caricamento della pagina potrebbe impiegare fino a un minuto circa. Per maggiori informazioni, consultare il [capitolo sul deployment](#).

Implementazione del FrontEnd

Il FrontEnd fornisce le funzionalità di visualizzazione, inserimento e cancellazione dei dati dell'applicazione. È stato realizzato con il framework per interfacce grafiche [Vue.js](#), che permette di realizzare applicazioni web dinamiche e single-page.

Design

Il design, realizzato in CSS, ha come caratteristica principale la divisione della pagina in contenitori semitrasparenti ad effetto "frosted glass". In questo modo essi possono essere sovrapposti liberamente lasciando intravedere il livello sottostante, ma senza risultare confusionari o illeggibili.

Un'altra caratteristica ricorrente sono gli angoli degli elementi, due dei quali sono smussati per dare un effetto "fumetto" e aumentare ancora di più il senso di tridimensionalità dato dalla semitrasparenza.

È stato fatto un forte uso di icone, fornite da [FontAwesome](#), per favorire un linguaggio più intuitivo e facile da comprendere, in linea con quanto stabilito dai requisiti non funzionali.

Esempi

Di seguito vengono riportati alcuni screenshot delle varie pagine dell'applicazione.

La landing page del sito è riportata nella Figura 10. Come definito inizialmente nei requisiti funzionali, abbiamo incluso una mappa indicante un'ipotetica posizione dell'oratorio.



Figura 10 – Homepage

La pagina mostrata nella Figura 11 si occupa della presentazione degli spazi e servizi dell'oratorio. Al caricamento della pagina viene fatta richiesta di spazi e servizi al BackEnd e vengono mostrati ordinatamente uno dopo l'altro.

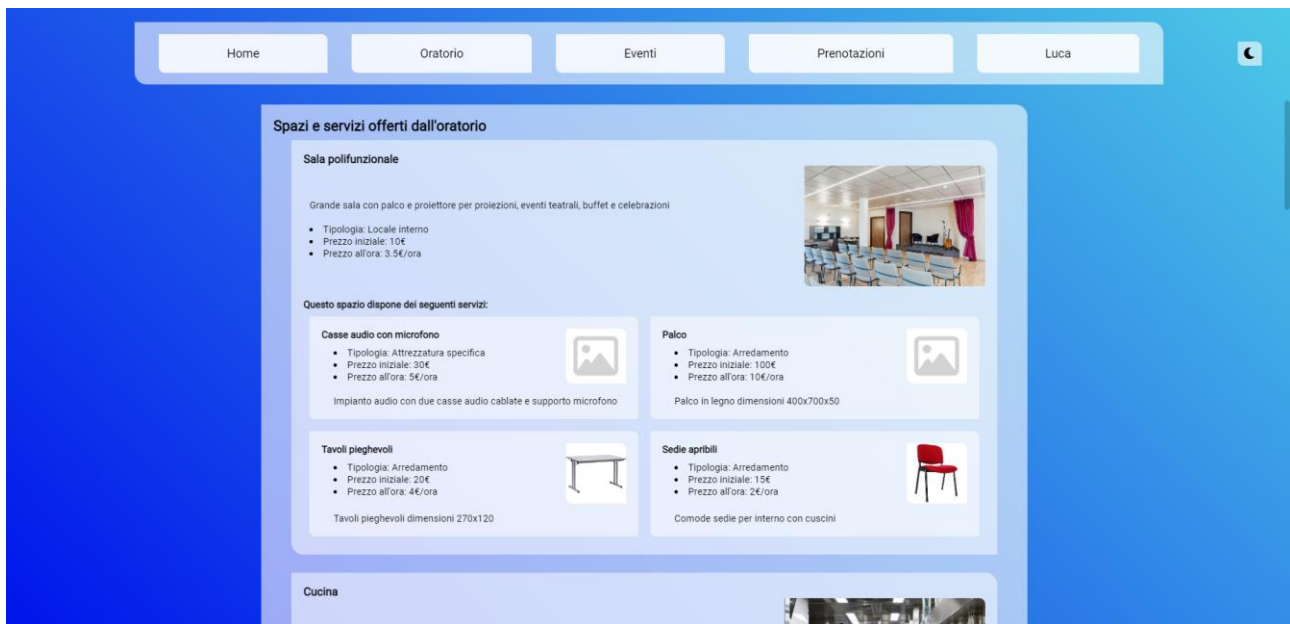


Figura 11 – Pagina per mostrare spazi e servizi

La Figura 12 riporta l'aspetto della pagina del profilo di un utente, che è formata da un form riempito con i dati dell'utente loggato al momento. I pulsanti sottostanti permettono di modificare i dati dell'utente (abilitando i campi del form) oppure di eliminare l'account.

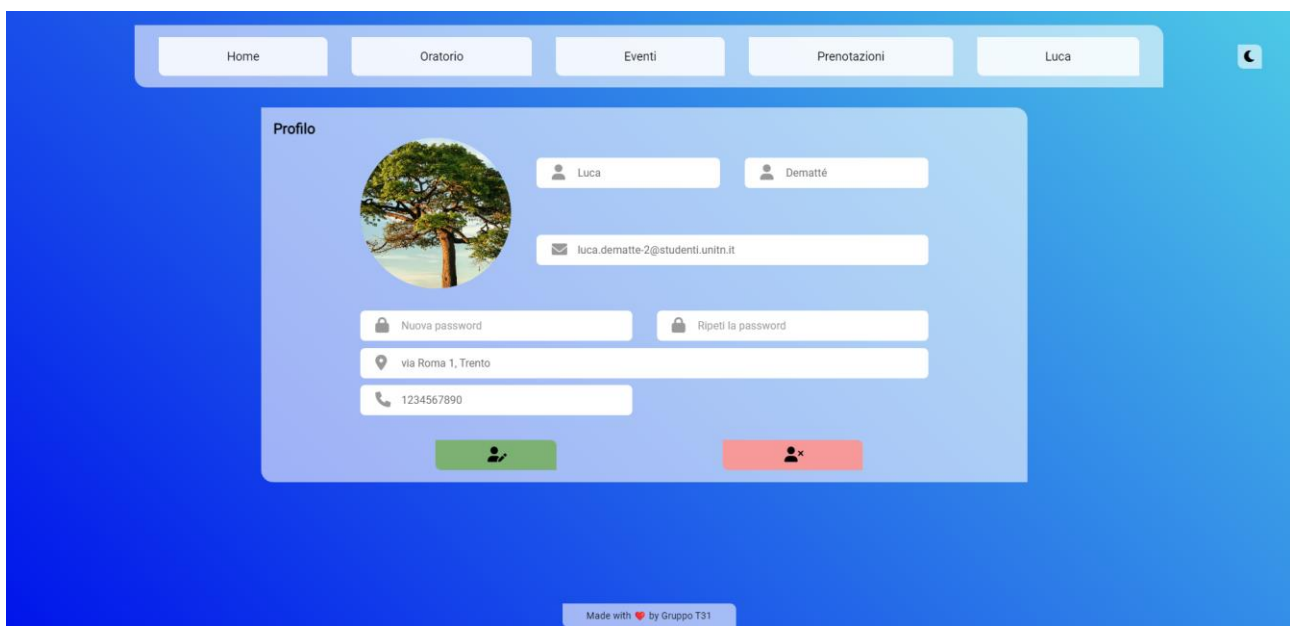


Figura 12 – Pagina del profilo

Selezionando il pulsante Prenotazioni nella navbar, viene visualizzata una pagina con l'elenco di tutte le prenotazioni dell'utente loggato (Figura 13). Da questa pagina premendo il bottone Nuova prenotazione, si può raggiungere il form per inserire una nuova prenotazione (Figura 14).

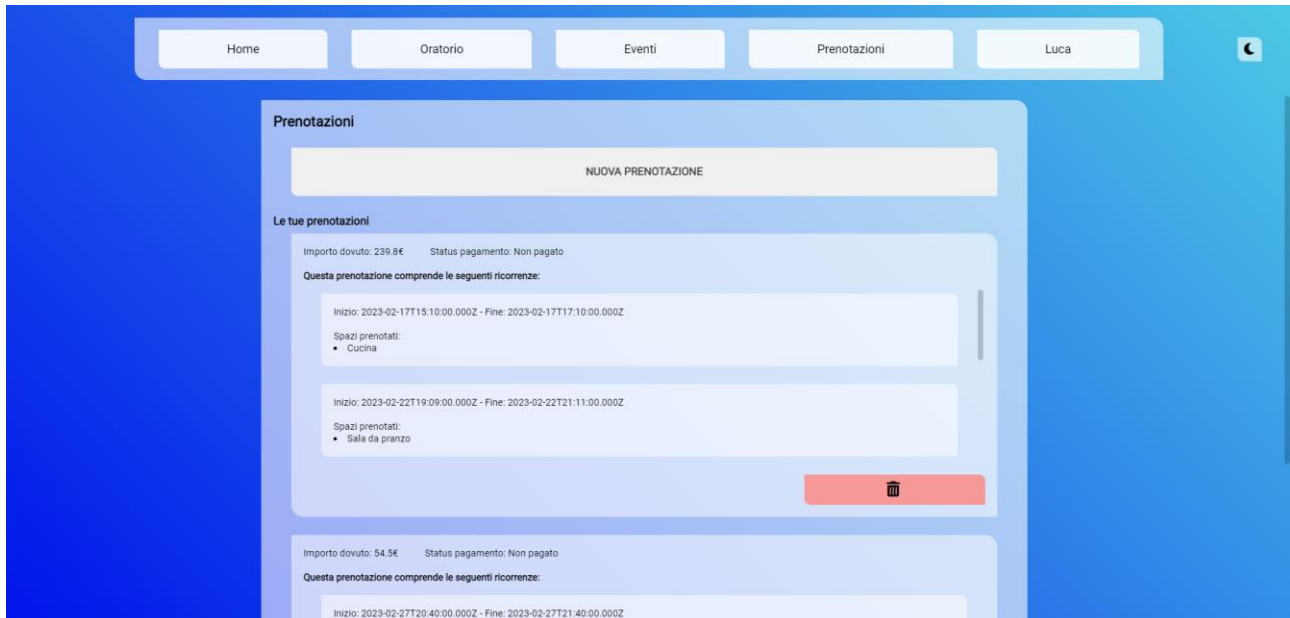


Figura 13 – Pagina lista prenotazioni

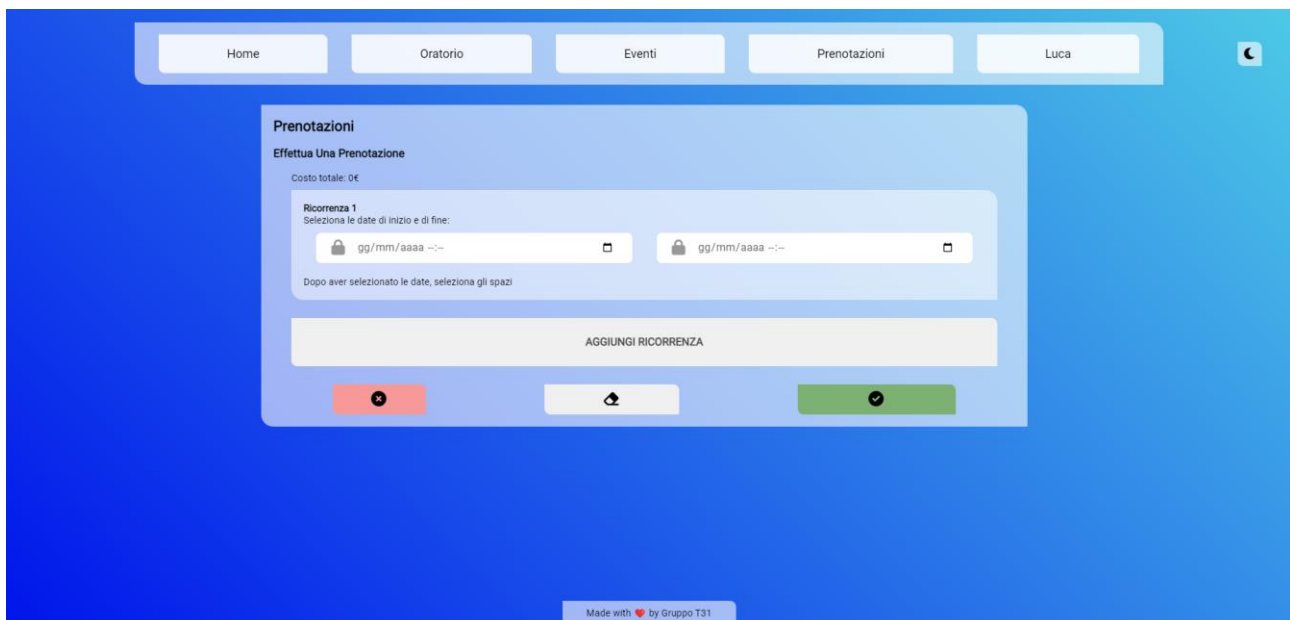


Figura 14 – Pagina di creazione di una nuova prenotazione

Repository GitHub e informazioni sul deployment

Repository GitHub

Come già accennato nei capitoli precedenti, abbiamo organizzato lo sviluppo del progetto in due repository: [una per il backend](#) e [una per il frontend](#).

Abbiamo fatto ampio utilizzo del branching per permettere uno sviluppo parallelo su diverse parti dell'applicazione. Inoltre, abbiamo sperimentato l'uso delle issues e milestones per coordinarci nello sviluppo, anche se alla fine abbiamo deciso di coordinarci con altri mezzi.

Deployment

A causa della chiusura dei server su Heroku per gli account gratuiti, abbiamo scelto di utilizzare due servizi di hosting differenti per il deployment: GitHub Pages e Render.com.

La separazione di frontend e backend su servizi diversi significa che in ognuno dei due bisogna specificare l'indirizzo dell'altro e che il backend deve impostare gli headers CORS in maniera corretta, listando il frontend come 'allowed-origin'.

Backend

Il Back-end è hostato su [render.com](#). Questo è un servizio che permette di hostare online progetti in diversi linguaggi (node.js incluso) e fornisce un dominio pubblico per accedervi.

Hostare il backend grazie a questo servizio ha dei pro e dei contro:

PRO:

- gratuito, con la possibilità di pagare per avere feature aggiuntive,
- deployment automatico se collegato al repository e branch del progetto.

CONTRO:

- lentezza,
- il servizio va in ibernazione dopo un certo periodo di inattività.

NB: a causa di questo problema la prima chiamata alle api potrebbe richiedere più tempo del normale (fino a 1 minuto circa).

- storage:

Il servizio cancella tutti i file salvati quando va in ibernazione.

NB: Questo significa che le foto profilo caricate dagli utenti vengono perse dopo un certo periodo di inattività

Entrambi questi contro potrebbero essere risolti passando al piano a pagamento di render.com, ma possono essere tollerati nello scope del nostro progetto.

Il backend può essere raggiunto al seguente indirizzo:

<https://gruppo-t31-api.onrender.com/>

Frontend

Il frontend è hostato su [Github Pages](#), un servizio di hosting per pagine statiche fornito da GitHub che preleva i contenuti direttamente dal repository. Questo viene fatto grazie ad un subtree del repository Frontend che contiene solo la directory **/dist** del progetto Vue. Questa directory viene costruita dal comando 'build' e contiene il sito in forma statica pronto per essere hostato.

Hostare il sito tramite Github Pages ha dei pro e dei contro.

PRO:

- gratuito,
- reattivo,
- deployment automatico dopo aver modificato il branch di deployment.

CONTRO:

- la natura statica di github pages crea dei conflitti con i router di Vue.

Per questo le pagine non sono direttamente accessibili, ma bisogna accedere alla home e navigare il sito tramite la navbar.

NB: questo vieta anche di refreshare le pagine diverse dalla home.

Questi problemi potrebbero essere risolti passando a un servizio di hosting diverso, ma renderebbe il processo di deployment significativamente più complesso ed il tempo di caricamento delle pagine più alto.

Github Pages

Github Pages is designed to host your personal, organization, or project pages from a GitHub repository.

Your site is live at <https://t31-is2022.github.io/Frontend/>
Last deployed by  DiegoOniarti 21 minutes ago

[Visit site](#)

Build and deployment

Source

Deploy from a branch ▾

Branch

Your GitHub Pages site is currently being built from the deployment branch. [Learn more.](#)

 deployment ▾

 / (root) ▾

Save

Il frontend può essere raggiunto al seguente indirizzo:

<https://t31-is2022.github.io/Frontend/>

Testing

Il testing delle API è avvenuto tramite il framework Jest.

È stata creata una testing suite completa per il controller utente.js. In particolare, nella testing suite vengono effettuati unit test per tutti gli endpoint presenti nel controller utente.

Per effettuare i test è necessario che il server di backend sia attivo; infatti, la verifica del corretto funzionamento del codice avviene effettuando delle richieste fetch verso il server. Per effettuare le richieste è stato utilizzato il modulo node-fetch, che implementa le fetch API in node.js e permette al programmatore di creare ed effettuare fetch request.

La risposta del server viene analizzata e confrontata con quella attesa dallo sviluppatore, ad esempio se viene effettuata una richiesta con un token non valido ci aspettiamo di ricevere dal server una risposta con status code 403 Forbidden. Le richieste che vengono effettuate durante il testing simulano quelle che verranno fatte dal front-end durante un normale caso d'uso di utilizzo del sito.

A seguire uno screenshot dell'output del comando jest -coverage:

```

PASS test/utente.test.js (13.804 s)
  listaUtenti: GET /
    ✓ start e count non sono di tipo number (54 ms)
    ✓ token mancante (1 ms)
    ✓ token non valido (2 ms)
    ✓ start e count numeri negativi (325 ms)
    ✓ non vengono passati parametri allora utilizza quelli di default (732 ms)
    ✓ start=2 e count=5 sono parametri validi (713 ms)
  loginUtente: POST /login
    ✓ cerca utente con email: test@test e password: test (187 ms)
    ✓ cerca utente con email: non@esiste e password: non esiste (121 ms)
    ✓ password e email non specificate (4 ms)
  registrazione: POST /registrazione
    ✓ parametri nuovo utente non specificati (3 ms)
    ✓ creazione nuovo utente con email già presente nel database (42 ms)
    ✓ creazione nuovo utente con dati validi (710 ms)
  getUtenteConEmail: GET /byEmail
    ✓ email non specificata nella query (267 ms)
    ✓ cerca utente che esiste (715 ms)
    ✓ token mancante (2 ms)
    ✓ token non valido (3 ms)
    ✓ cerca utente che non esiste (849 ms)
  getUtenteConID: GET /byID
    ✓ id non specificata nella query (271 ms)
    ✓ cerca utente con id presente nel database (717 ms)
    ✓ token mancante (3 ms)
    ✓ token non valido (2 ms)
    ✓ cerca con id non valido (143 ms)
    ✓ cerca id valido, ma non presente nel database (362 ms)
  modificaUtente: PATCH /:id
    ✓ id non valido (408 ms)
    ✓ id valido, ma parametri del body non specificati (308 ms)
    ✓ id non corrisponde a nessun utente (716 ms)
    ✓ modifiche effettuate correttamente (127 ms)
  cancellaUtente: DELETE /:id
    ✓ id non valido (217 ms)
    ✓ id valido ma non corrisponde a nessun utente (371 ms)
    ✓ utente eliminato correttamente (1124 ms)
  confermaUtente: GET /conferma
    ✓ id non specificato (2 ms)
    ✓ id non valido (2 ms)

-----|-----|-----|-----|-----|-----|
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----|
All files |    100 |    100 |    100 |    100 |
  utente.js |    100 |    100 |    100 |    100 |
-----|-----|-----|-----|-----|
Test Suites: 1 passed, 1 total
Tests:       32 passed, 32 total
Snapshots:   0 total

```

Inoltre, Jest crea automaticamente una pagina html nella quale è possibile visualizzare il risultato dei test ed altre informazioni utili.

Di seguito vengono riportate alcuni test delle API di registrazione e login:

```
// email test@test e' presente nel database
test('creazione nuovo utente con email gia presente nel database', async () => {
  expect.assertions(1)
  var response = await fetch(url+api, {
    method: 'POST',
    body: new URLSearchParams({
      nome: 'test',
      cognome: 'test',
      email: 'test@test',
      password: 'test',
      telefono: '123456789',
      indirizzo: 'test',
      URLfoto: 'url/test/foto'
    })
  });
  expect(response.status).toEqual(409)
});

test('creazione nuovo utente con dati validi', async () => {
  expect.assertions(1)
  var response = await fetch(url+api, {
    method: 'POST',
    body: new URLSearchParams({
      nome: 'test',
      cognome: 'test',
      email: 'nuovo@utente',
      password: 'test',
      telefono: '123456789',
      indirizzo: 'test',
      URLfoto: 'url/test/foto'
    })
  });
  // rimuove l'utente appena inserito nel db
  await Utente.findOneAndDelete({ email: "nuovo@utente"})
  expect(response.status).toEqual(201)
});
```

```
describe('loginUtente: POST /login', () => {
  const api = '/login'

  // utente con queste credenziali esiste nel database di testing
  test('cerca utente con email: test@test e password: test', async () => {
    expect.assertions(1)
    var response = await fetch(url+api, {
      method: 'POST',
      body: new URLSearchParams({
        email: "test@test",
        password: "test"
      })
    });
    expect(response.status).toEqual(200)
  })

  // utente con queste credenziali NON esiste nel database di testing
  test('cerca utente con email: non@esiste e password: non esiste', async () => {
    expect.assertions(1)
    var response = await fetch(url+api, {
      method: 'POST',
      body: new URLSearchParams({
        email: "non@esiste",
        password: "non esiste"
      })
    });
    expect(response.status).toEqual(404)
  })
});
```


Di seguito la tabella dei casi di test effettuati. Dove non diversamente specificato si assume che la richiesta è stata effettuata con un token jwt valido.

Numero Test Case	Descrizione Test Case	Test Data	Precondizioni	Risultato Atteso	Risultato Incontrato
1.1	Lista utenti con parametri non validi	<start> not a number <count> not a number		Errore 400	Errore 400
1.2	Lista utenti con token mancante	<start> numero <count> numero <x-access-token> vuoto		Errore 401	Errore 401
1.3	Lista utenti con token non valido	<start> numero <count> numero <x-access-token> token non valido		Errore 403	Errore 403
1.4	Lista utenti con parametri negativi	<start> numero negativo <count> numero negativo		Errore 400	Errore 400
1.5	Lista utenti con parametri di default	<start> vuoto <count> vuoto		OK 200 e json contenente gli utenti secondo i parametri di default	OK 200
1.6	Lista utenti con parametri validi	<start> numero positivo <count> numero positivo <x-access-token> vuoto		OK 200 e json contenente gli utenti secondo i parametri di default	OK 200
2.1	Login utente valido	<email> test@test <password> test		OK 200 viene ritornato il token	OK 200
2.2	Login utente che non esiste	<email> non@esiste <password> nonEsiste	Email non@esiste non è presente nel database	Errore 404	Errore 404
2.3	Login con password e email non specificate	<email> vuoto <password> vuoto		Errore 400	Errore 400

3.1	Registrazione con parametri non specificati	<nome> vuoto <cognome> vuoto <email> vuoto <password> vuoto <telefono> vuoto <indirizzo> vuoto <URLfoto> vuoto		Errore 400	Errore 400
3.2	Registrazione utente già presente nel database	<nome> qualsiasi string <cognome> qualsiasi string <email> test@test <password> string valida <telefono> numero <indirizzo> string <URLfoto> percorso foto	Utente con email test@test è presente nel database	Errore 409	Errore 409
3.3	Creazione nuovo utente con dati validi	<nome> qualsiasi string <cognome> qualsiasi string <email> nuovo@utente <password> string valida <telefono> numero <indirizzo> string <URLfoto> percorso foto	Utente con email nuovo@utente non è presente nel database	OK 201 e viene creato nuovo utente nel database (utente che viene subito dopo rimosso)	OK 201 e viene creato nuovo utente nel database (utente che viene subito dopo rimosso)
4.1	Get utente con email non specificata	<email> vuoto		Errore 400	Errore 400
4.2	Get utente con email presente nel database	<email> test@test	Utente con email test@test è presente nel database	OK 200 e viene ritornato un json con l'utente con mail specificata	OK 200

4.3	Get utente token mancante	<email> qualsiasi email <x-access-token> vuoto		Errore 401	Errore 401
4.4	Get utente token non valido	<email> qualsiasi email <x-access-token> token non valido		Errore 403	Errore 403
4.5	Get utente con email che non esiste nel database	<email> non@esiste	Utente con email non@esiste non è presente nel database	Errore 404	Errore 404
5.1	Get utente id non specificato	<id> vuoto		Errore 400	Errore 400
5.2	Get utente con id presente nel database	<id> id del test user	<id> id esiste nel database associato a test user con email test@test	OK 200 e viene ritornato un json contenente i dati dell'utente con id specificato	OK 200
5.3	Get utente con token livello 2 mancante	<id> qualsiasi id <x-access-token> vuoto		Error 401	Error 401
5.4	Get utente con token livello 2 non valido	<id> qualsiasi <x-access-token> non valido		Error 403	Error 403
5.5	Get utente id non valido	<id> id non valido		Error 400	Error 400
5.6	Get utente id valido ma non presente nel database	<id> id non esiste nel database	<id> nel database non esistono utenti con l'id specificato	Error 404	Error 404
6.1	Modifica utente id non valido	<id> id non valido		Error 400	Error 400
6.2	Modifica utente id valido ma parametri da modificare non specificati	<id> id test user <nome> vuoto <cognome> vuoto <email> vuoto <password> vuoto <telefono> vuoto <indirizzo> vuoto		Error 400	Error 400

		<URLfoto> vuoto			
6.3	Modifica utente id non presente nel database	<id> id non esiste	<id> nel db non sono presenti utenti con l'id indicato	Error 404	Error 404
6.4	Modifica utente con successo	<id> id test user <nome> qualsiasi string <cognome> qualsiasi string <email> test@test <password> string valida <telefono> numero <indirizzo> string <URLfoto> percorso foto	Utente con email test@test è presente nel database	OK 200 e vengono modificati i dati dell'utente indicato	OK 200 e vengono modificati i dati dell'utente indicato
7.1	Cancella utente id non valido	<id> id non valido		Errore 400	Errore 400
7.2	Cancella utente id valido ma non presente nel database	<id> id non esiste nel database	<id> id non esiste nel database	Errore 404	Errore 404
7.3	Utente eliminato con successo	<id> id utente da eliminare	Viene creato un utente temporaneo da eliminare	OK 200 e viene eliminato l'utente con l'id specificato	OK 200 e viene eliminato l'utente con l'id specificato
8.1	Conferma utente id non specificato	<id> vuoto		Errore 400	Errore 400
8.2	Conferma utente id non valido	<id> id non valido		Errore 400	Errore 400
8.3	Conferma utente effettuata con successo	<id> test user id	<id> id valido appartiene all'utente test@test	OK 200 e viene settato a 'true' il campo utenteConfermato dell'utente con l'id specificato	
8.4	Conferma utente id valido ma non presente nel database	<id> id non esiste nel database	<id> id non esiste nel database	Error 404	