



# DOCUMENTO DI SVILUPPO

<b>Gruppo</b>	<b>G34</b>	<b>Membri</b>	<b>Marco Pulze, Sara Boscardin, Luca Damaschetti</b>
<b>Titolo Documento</b>	<b>Documento di Sviluppo</b>		

# SOMMARIO

## DOCUMENTO DI SVILUPPO

<b>0) SCOPO DEL DOCUMENTO</b>	4
<b>1) USER FLOW DIAGRAM</b>	5
1.1) USER FLOW UTENTE NON AUTENTICATO	6
1.2) USER FLOW UTENTE NON IN APPARTAMENTO	7
1.3) USER FLOW UTENTE IN APPARTAMENTO	8
<b>2) STRUTTURA</b>	10
2.1) STRUTTURA DEL PROGETTO	11
2.2) LIBRERIE E DEPENDENCY	13
2.3) FILE DOTENV	15
<b>3) MODELLI DEL DATABASE</b>	15
3.1) MODELLO UTENTE	16
3.2) MODELLO APPARTAMENTO	17
3.3) MODELLO POST-IT	18
3.4) MODELLO PRODOTTO	18
<b>4) API</b>	19
4.1) DIAGRAMMA DI ESTRAZIONE DELLE RISORSE	19
4.2) RESOURCE MODELING DIAGRAM	20
4.2.1) AUTH RESOURCE MODELING DIAGRAM	20
4.2.2) APARTMENT RESOURCE MODELING DIAGRAM	21
4.2.3) APARTMENT CONTENT RESOURCE MODELING DIAGRAM	22
Lista della Spesa	22
Bacheca	22
4.3) DOCUMENTAZIONE E DESCRIZIONE API	23
4.3.1) AUTH	23
LOGIN	23
REGISTER	25
4.3.2) APPARTAMENTO	26
JOIN	26
DELETE	28
4.3.3) LISTA DELLA SPESA	29

---

AGGIUNGI .....	29
ELIMINA.....	30
GET ALL .....	30
SEGNA PRODOTTO.....	31
4.3.4) BACHECA .....	31
CREA .....	31
GET ALL .....	31
4.4) DOCUMENTAZIONE OPENAPI .....	32
<b>5) TESTING .....</b>	<b>34</b>
5.1) Documentazione dei Test - GET /api/lista/getAllProducts .....	35
<b>6) FRONTEND.....</b>	<b>36</b>
6.1) COMPONENTS .....	37
Header.tsx.....	37
Register.tsx .....	39
ApartmentCreateComp.tsx .....	42
ApartmentJoinComp.tsx .....	43
Login.tsx.....	45
HomeComp.tsx.....	47
BachecaComp.tsx .....	50
SpesaComp.tsx .....	52
6.2) PAGINE .....	56
Register.tsx .....	56
Login.tsx.....	57
Index.tsx.....	57
<b>7) DEPLOYMENT .....</b>	<b>63</b>
7.1) ESECUZIONE IN LOCALE .....	63
7.2) DEPLOYMENT CON VERCEL .....	64

## 0) SCOPO DEL DOCUMENTO

Il seguente documento descrive come è avvenuta la parte finale di sviluppo del progetto per l'applicazione RoomUnity.

Nella sezione 1 viene trattato lo User Flow Diagram, che mostra come l'utente si districa nella totalità delle funzionalità che dovranno essere implementate al fine della realizzazione del progetto/prototipo.

Nella sezione 2 vengono trattate le API, ossia il relativo lavoro di Back-End annesso alla struttura, con le configurazioni e le dipendenze del progetto che dovranno essere usate in fase di sviluppo.

Nella sezione 3 sono descritti i modelli che vengono utilizzati dal database dell'applicazione.

Nella sezione 4 vengono trattate la documentazione e le descrizioni delle API principali del progetto. Il capitolo analizza tramite il Resource Extraction Diagram e il Resource Model Diagram come dovranno essere sviluppate e implementate le API dell'applicazione e successivamente tratta la documentazione delle stesse mediante OpenApi.

La sezione 5 riporta le attività di testing delle API principali presentate nella sezione 4.

Nelle sezioni 6 e 7 si presenta il Front-End dell'applicazione e il Deployment della stessa.

**Nota:** A partire dal diagramma delle classi presentato nel documento D3, abbiamo identificato le funzionalità essenziali da mantenere per lo sviluppo dell'applicazione. Abbiamo scelto di implementare solo una parte delle funzionalità originariamente previste dal progetto RoomUnity, concentrandoci su quelle che garantissero una maggiore qualità del risultato finale e un uso più efficiente del tempo e delle risorse del team. Di conseguenza, il documento D4 non rifletterà completamente tutti i dettagli presenti nei documenti precedenti, ma resterà comunque coerente con essi e con il codice del progetto.

## 1) USER FLOW DIAGRAM

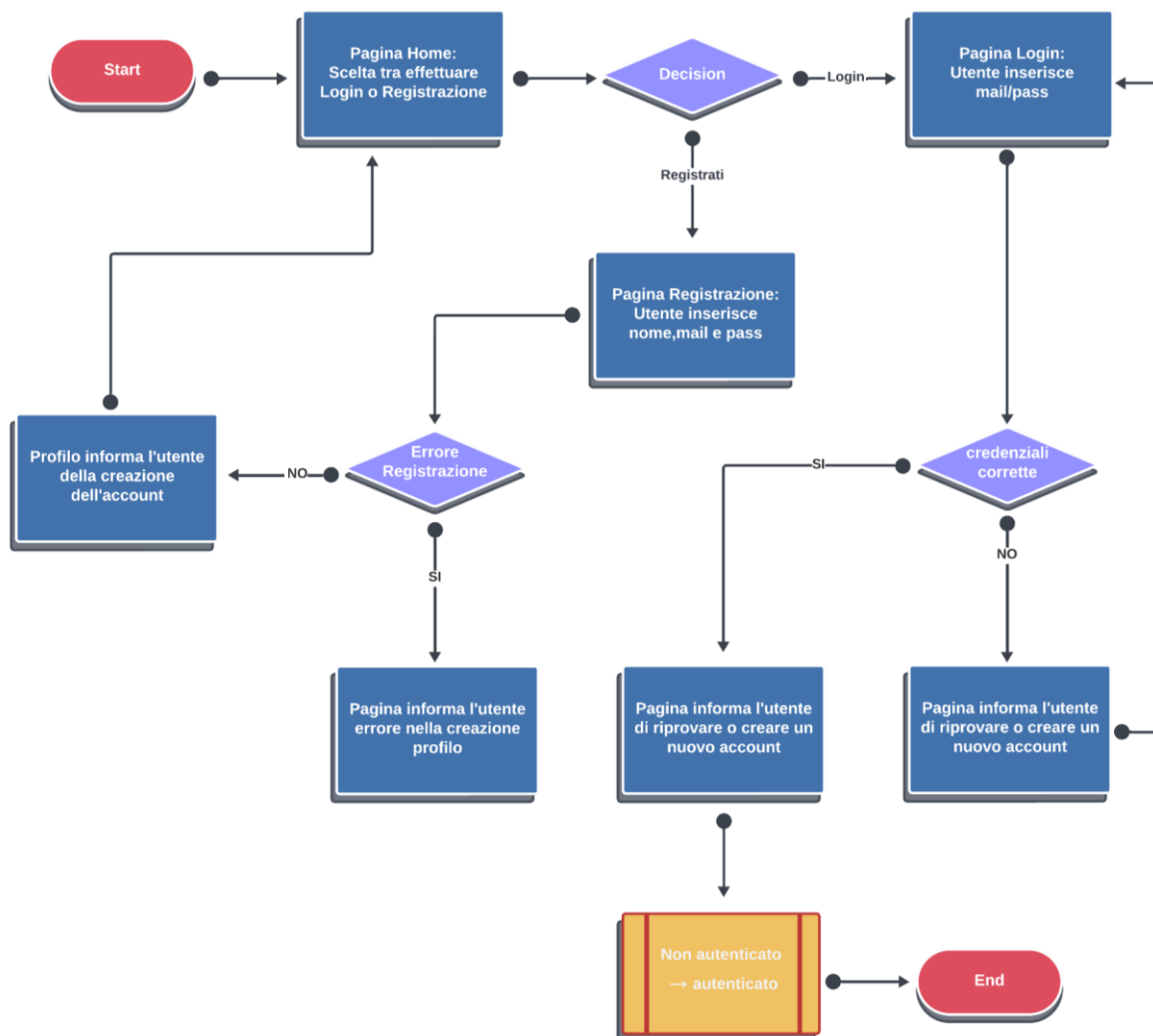
La sezione riporta lo User Flow Diagram per il progetto RoomUnity sviluppato. Il diagramma è stato diviso in tre parti per semplificarne la lettura e comprensione. Infatti, nonostante esista un'unica tipologia di utente all'interno dell'applicazione, durante l'utilizzo dell'applicazione gli sono rese disponibili funzionalità differenti (come descritto nei documenti precedenti).

### Legenda:



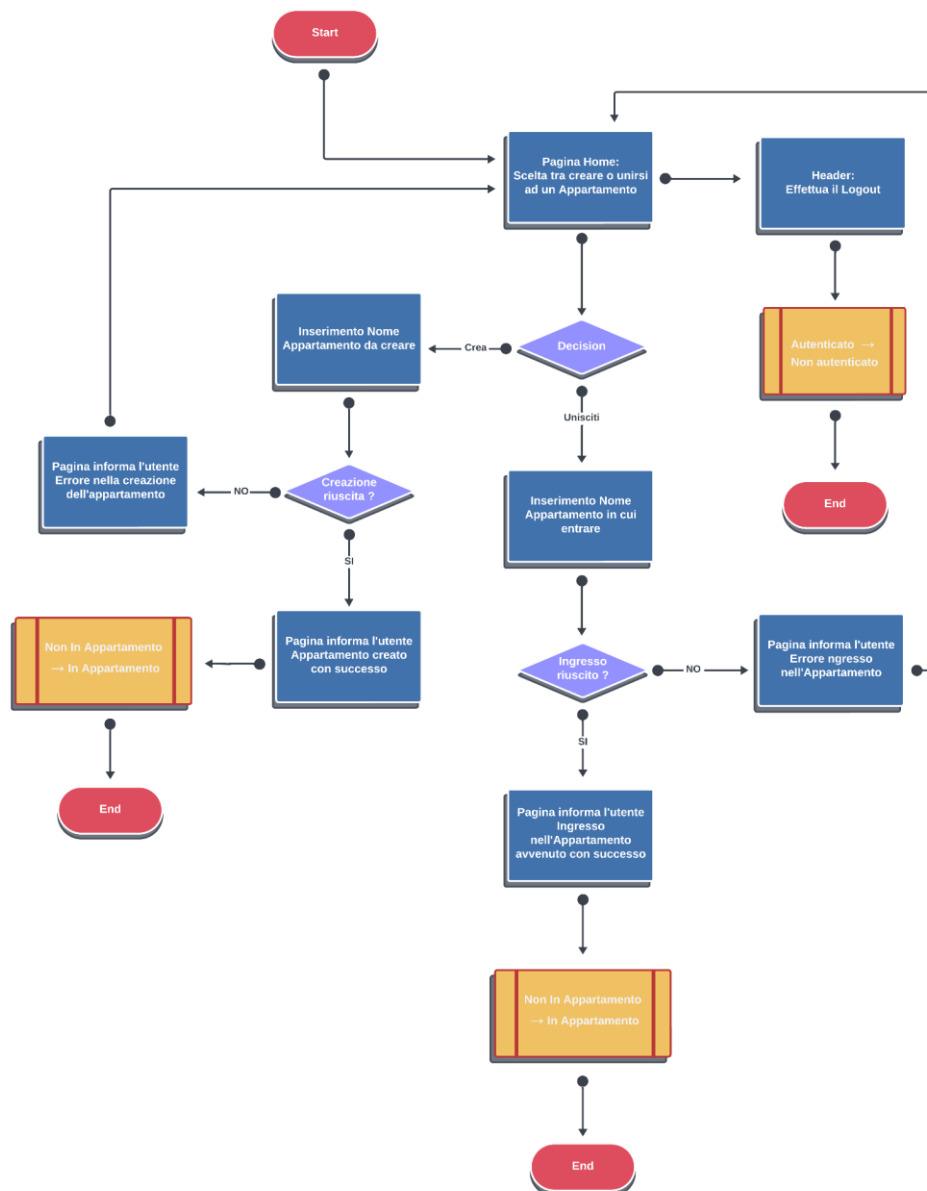
## 1.1) USER FLOW UTENTE NON AUTENTICATO

Il primo User Flow Diagram presentato è quello riguardante l'utente non ancora autenticato. Seguendo i requisiti definiti dal documento D1, l'utente potrà effettuare la registrazione (RF1) o alternatively, in caso l'utente abbia già creato un account in precedenza, il login (RF2).



## 1.2) USER FLOW UTENTE NON IN APPARTAMENTO

Il secondo User Flow Diagram è dedicato agli utenti che non sono ancora mai stati all'interno di un appartamento, oppure sono usciti e non ancora entrati in uno nuovo. Come per lo User Flow precedente, sono stati seguiti i requisiti funzionali del D1. L'utente potrà agire in due macrosezioni: una "sezione" di gestione dell'account e della sessione corrente, che gli permetta di effettuare il logout (RF3), e una sezione di gestione dell'appartamento dell'utente, dove potrà creare un appartamento nuovo (RF6) oppure entrare in un appartamento già esistente (RF7).

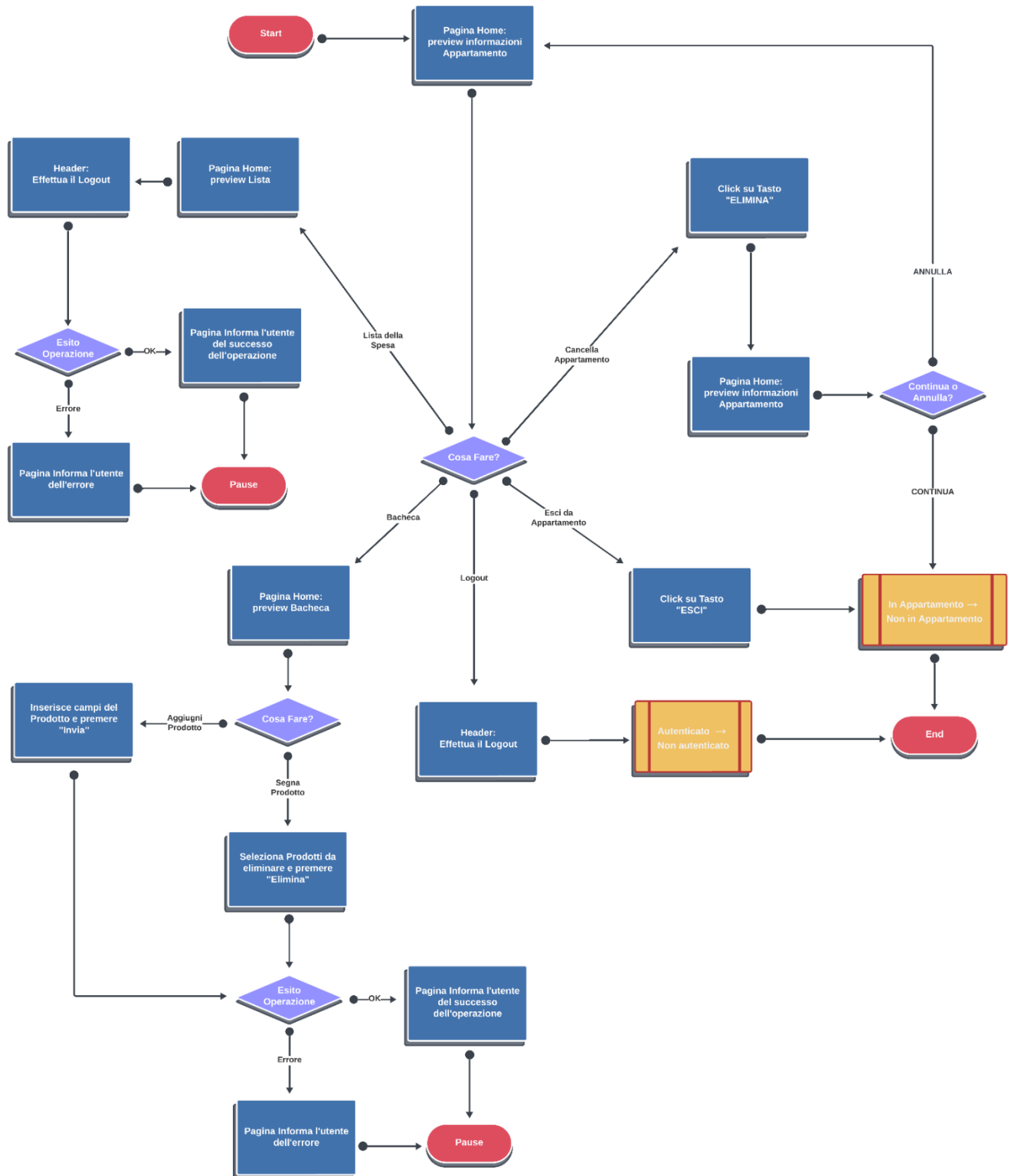


---

### **1.3) USER FLOW UTENTE IN APPARTAMENTO**

L'ultimo User Flow mostra il percorso seguito dall'utente che ha effettuato il login ed è attualmente all'interno di un appartamento. Analogamente al punto 1.2 anche qui l'utente può accedere alla "sezione" di gestione dell'account e della sessione corrente; oltre ad una "sezione" di gestione dell'appartamento in cui si trova attualmente (RF13). L'utente potrà inoltre accedere alla bacheca dell'appartamento (RF8), la lista della spesa (RF9), modificare il proprio stato di presenza in casa (RF12), oltre a poter cancellare l'appartamento (RF14) o uscire dallo stesso (RF15).





## 2) STRUTTURA

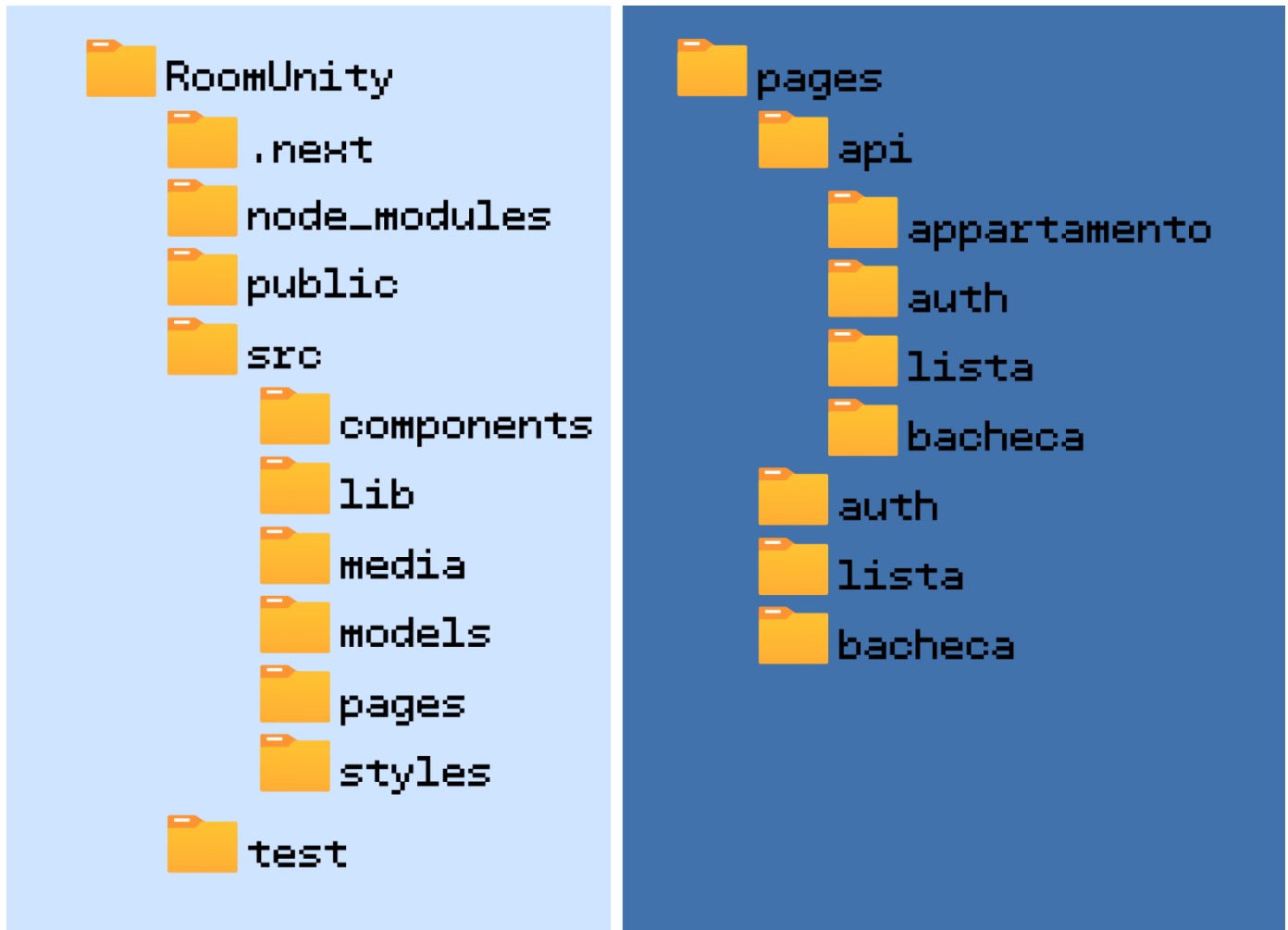
L'applicazione è stata costruita interamente su Next.js, un framework JavaScript di punta che permette di sviluppare un prodotto web performante e scalabile. In questa sezione, verrà trattata l'architettura interna del progetto, analizzando nel dettaglio la struttura dei file e delle directory che ne compongono la base.

Inizialmente ci sarà una panoramica generale sull'organizzazione del codice nelle rispettive cartelle, esplorando le diverse aree funzionali e le loro rispettive responsabilità e funzionalità.

Successivamente, si presenterà rapidamente l'elenco delle principali librerie e dei pacchetti che sono stati integrati nel progetto per estendere le funzionalità.

Infine, ci concentreremo sulla configurazione del file `.env`, un elemento cruciale per il corretto funzionamento dell'applicazione in diversi ambienti (sviluppo, produzione, ecc.). All'interno di questo file, sono definite le variabili d'ambiente che contengono informazioni sensibili, come le chiavi API, le stringhe di connessione al database e altri parametri di configurazione. Esploreremo le diverse variabili presenti e spiegheremo il loro impatto sul comportamento dell'applicazione.

## 2.1) STRUTTURA DEL PROGETTO



**.next:** Directory creata automaticamente da Next.js durante il processo di build. Contiene i file di output, come il codice JavaScript ottimizzato, le pagine pre renderizzate per migliorare le performance, e i file di configurazione per il runtime. Questa cartella non è presente nella repository del progetto GitHub in quanto autogenerata e di importanti dimensioni.

**node\_modules:** Questa directory contiene tutte le dipendenze e i pacchetti Node.js installati tramite npm. Questi pacchetti sono fondamentali per il funzionamento e lo sviluppo dell'applicazione. La cartella può crescere notevolmente in dimensione, contenendo framework, librerie e strumenti di sviluppo, man mano che vengono aggiunti nuovi pacchetti. Per questo motivo, questa cartella non è inserita all'interno della repository del progetto GitHub.

**public:** Cartella contenente gli asset liberamente accessibili in tutto il progetto, oltre che direttamente dalla pagina web.

**src:** La cartella src (abbreviazione di "source") contiene il codice sorgente principale dell'applicazione. All'interno di questa directory si trovano solitamente le componenti dell'applicazione, tra cui troviamo:

**components:** All'interno di questa directory sono raggruppati tutti i componenti utilizzati dal frontend. Una dettagliata descrizione di ciascun componente sarà fornita nella sezione apposita.

**Nota:** nonostante i componenti siano riutilizzabili, la gran maggioranza è stata usata massimo una o due volte, ma la loro scelta è scaturita dall'interesse nel poter sfruttare al meglio le feature del framework.

**lib:** All'interno di questa directory vi è il modulo che gestisce la connessione con il database, un modulo contenente le funzioni per la gestione sicura delle password ed un modulo che definisce l'interfaccia utilizzata per la creazione delle sessioni.

**media:** All'interno di questa directory sono ospitate le immagini utilizzate nell'interfaccia grafica del front end dell'applicazione.

**models:** All'interno di questa directory sono collocati i modelli definiti per l'utilizzo delle collection del database MongoDB.

**pages:** All'interno di questa directory sono collocate le pagine dell'applicazione, organizzate in sottocartelle. Oltre alle sottocartelle delle pagine è presente un'altra cartella API:

**api:** nella directory troviamo le interfacce di programmazione necessarie al progetto. Ogni file all'interno di questa cartella rappresenta un endpoint API che può essere richiamato dal frontend o da altre parti dell'applicazione. Le route API sono gestite come funzioni

serverless, che eseguono codice lato server per rispondere alle richieste HTTP.

**styles:** All'interno di questa directory sono collocati alcuni fogli di stile CSS che definiscono l'aspetto grafico dell'applicazione.

**test:** La cartella in questione contiene tutti i vari test effettuati sull'applicazione per valutare il funzionamento corretto delle API sviluppate, il tutto tramite la libreria jest.

## 2.2) LIBRERIE E DEPENDENCY

All'interno della directory del progetto è presente il file: package.json che descrive il progetto, includendo dettagli come il nome, la versione e gli script di esecuzione. Elenca anche le dipendenze e le devDependencies richieste, e contiene metadati come l'autore, la licenza e le configurazioni specifiche per diversi ambienti di esecuzione.

In questa stessa sezione si tratterà più in dettaglio il contenuto del file package.json riportando le varie dipendenze e librerie usate:

```
"dependencies": {  
  "bcrypt": "^5.1.1",  
  "bcryptjs": "^2.4.3",  
  "bootstrap": "^5.3.3",  
  "bootstrap-icons": "^1.11.3",  
  "cookie": "^0.6.0",  
  "jsonwebtoken": "^9.0.2",  
  "mongodb": "^6.8.0",  
  "next": "14.2.5",  
  "react": "^18",  
  "react-bootstrap": "^2.10.4",  
  "react-dom": "^18",  
  "react-spinners": "^0.14.1",  
  "react-toastify": "^10.0.5",  
  "swagger-jsdoc": "^6.2.8"  
},
```

Di queste librerie si passeranno in rassegna le principali e più determinanti per il progetto:

**bcrypt:** bcrypt è una libreria per la hash delle password. Utilizza un algoritmo di hashing sicuro che rende difficile decifrare le password salvate nel database, dunque aumenta la sicurezza.

**bootstrap:** bootstrap è un framework CSS che semplifica la creazione di interfacce utente responsive e stilisticamente coerenti. Fornisce stili e componenti pronti per una UI reattiva e moderna.

**cookie:** cookie facilita l'analisi e la gestione dei cookie nelle richieste e risposte HTTP.

**jsonwebtoken:** jsonwebtoken permette di creare e verificare JSON Web Tokens (JWT), che sono comunemente usati per autenticare e autorizzare gli utenti nelle applicazioni web.

**mongoose:** mongoose è il driver ufficiale per connettersi e interagire con un database MongoDB.

**next:** next è un framework per React che semplifica il rendering lato server (SSR), la generazione di pagine statiche (SSG) e la gestione delle route.

**react:** react è una libreria per costruire interfacce utente reattive. Consente di creare componenti UI riutilizzabili e gestire lo stato dell'applicazione.

**react-bootstrap & react-\***: **react-bootstrap** fornisce i componenti di Bootstrap come componenti React, facilitando l'uso di Bootstrap in un'applicazione React senza dover gestire direttamente le classi CSS di Bootstrap.

Le altre dipendenze servono a includere altre feature grafiche per migliorare l'esperienza dell'utente.

## 2.3) FILE DOTENV

**.env:** File di configurazione che contiene variabili d'ambiente, utilizzato per gestire in modo sicuro dati sensibili quali credenziali di accesso al DB, e altre configurazioni specifiche per ambienti di sviluppo.

- **MONGODB\_URI:** Questa variabile contiene la stringa di autenticazione per collegarsi al nostro cluster su mongodb.
- **DB\_NAME:** Nome del Database.
- **JWT\_SECRET:** La variabile contiene una stringa segreta utilizzata per firmare i token JWT (JSON Web Tokens).

**readme.md:** file che contiene alcune informazioni sul progetto in generale e le informazioni su come avviarlo.

## 3) MODELLI DEL DATABASE

Roomunity è basato su un robusto Back-End che sfrutta le potenzialità di MongoDB, ospitato sulla piattaforma cloud MongoDB Atlas. MongoDB, grazie alla sua flessibilità e scalabilità, permette di gestire in modo efficiente i dati. Per interagire con il database, si è scelto di utilizzare Mongoose, una popolare libreria Node.js che semplifica notevolmente le operazioni di creazione, lettura, aggiornamento e cancellazione dei dati. Utilizzando Mongoose, si sono definiti degli schemi rigorosi per strutturare i documenti del progetto, garantendo così la coerenza e l'integrità dei dati.

In particolare, per l'applicazione Roomunity, abbiamo implementato i seguenti schemi MongoDB:

### 3.1) MODELLO UTENTE

```
1  import { ObjectId } from "mongodb"
2  //export serve perche sia visibile anche fuori, e default
3  export default class Utente {
4      constructor(
5
6          // Nome identificativo Utente, univoco
7          public nome: string,
8          // Email utente, univoca
9          public email: string,
10         // Password dell'Utente
11         public hashedPassword: string | undefined,
12         // Appartamento in cui è l'Utente
13         public appartamento?: string | undefined,
14
15         //id MongoDB
16         public _id?: ObjectId,
17     ) {}
18 }
19
```



### 3.2) MODELLO APPARTAMENTO

```
1  import { ObjectId } from "mongodb"
2  import PostIt from "../Post-it";
3  import Prodotto from "../Prodotto";
4
5  export default class Appartamento {
6      constructor(
7
8          // nome appartamento univoco
9          public nome: string,
10         // Collection di Post-It
11         public bacheca: PostIt[],
12         // Collection di Prodotti
13         public lista_spesa: Prodotto[],
14
15         //id MongoDB
16         public _id?: ObjectId,
17     ) {}
18 }
19
```

Inizialmente, si era previsto di includere nell'istanza dell'appartamento un campo per memorizzare una lista dei membri dell'appartamento, come indicato nei documenti precedenti (D3). Tuttavia, durante la fase di sviluppo, questo campo non è stato inserito, poiché mancavano alcune funzionalità legate all'appartamento e dunque la presenza di un campo "appartamento" all'interno della Classe Utente avrebbe reso ridondante e superfluo il campo "membri". Inoltre, ciò ha permesso di evitare il controllo aggiuntivo per verificare la presenza di un utente in più appartamenti contemporaneamente, riducendo così le ambiguità.

### 3.3) MODELLO POST-IT

```
1  import { ObjectId } from "mongodb"
2  //export serve perche sia visibile anche fuori, e default
3  export default class Postit {
4      constructor(
5
6          // Testo del Post-It
7          public contenuto: string,
8          // Data di pubblicazione post-it
9          public data: string,
10         // Autore Post-it, come nome dell'Utente
11         public autore: string,
12
13         //id MongoDB
14         public _id?: ObjectId,
15     ) {}
16 }
17
```

### 3.4) MODELLO PRODOTTO

```
1  import { Double, ObjectId } from "mongodb"
2  //export serve perche sia visibile anche fuori, e default
3  export default class Prodotto {
4      constructor(
5
6          // Nome del prodotto
7          public nome: string,
8          // Quantità del prodotto
9          public quantita: number,
10         // Prezzo Prodotto
11         public prezzo: number | undefined,
12         // Prodotto già acquistato
13         private already_signed: boolean,
14
15         //id MongoDB
16         public _id?: ObjectId,
17     ) {}
18 }
19
```

Nei documenti precedenti si era pensato di dividere i costi della spesa tra gli utenti. L'idea è stata sostituita con un la presenza di un campo *already\_signed*, che identifica

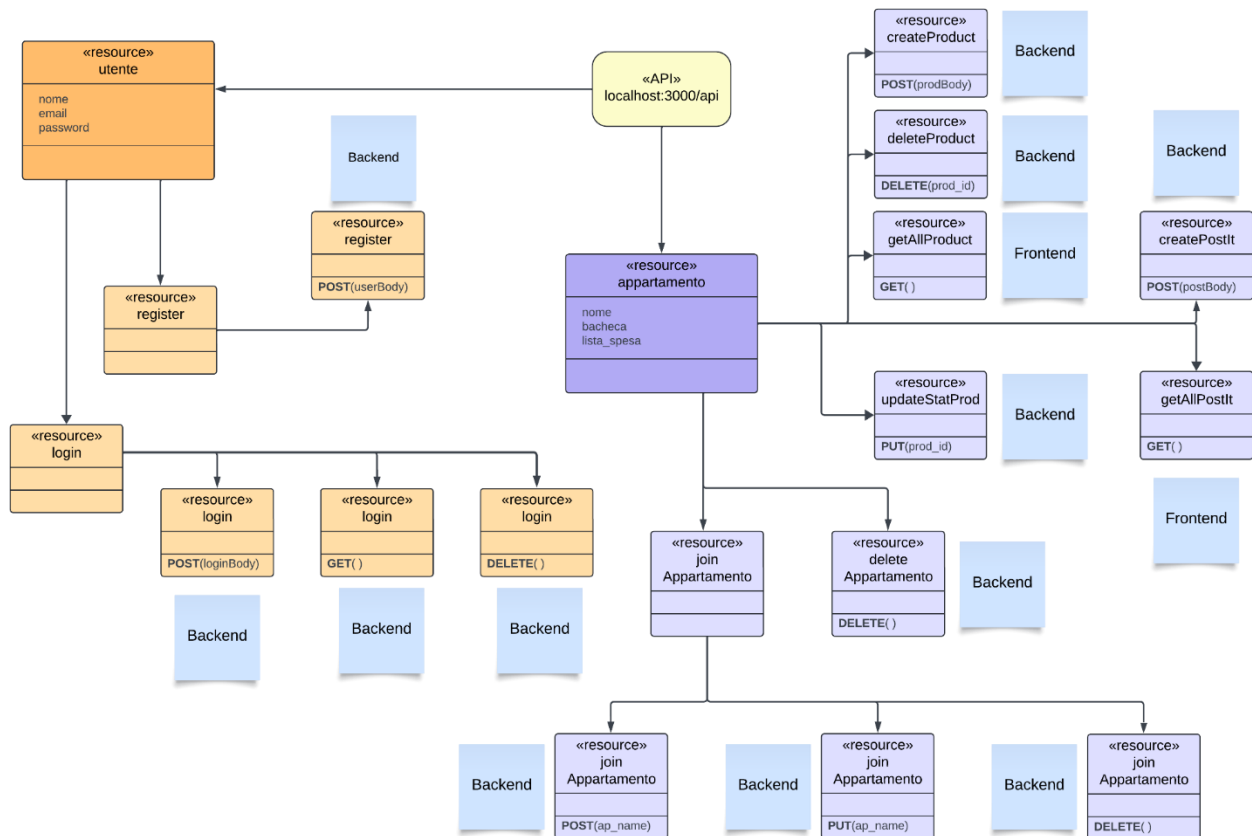
un prodotto comprato che però si vuole lasciare visibile temporaneamente agli altri utenti.

## 4) API

Questa sezione è dedicata alla descrizione e documentazione delle API principali del progetto. Attraverso l'uso del Resource Extraction Diagram e del Resource Model Diagram, verranno analizzate le modalità di sviluppo e implementazione delle API dell'applicazione. Successivamente, la documentazione verrà elaborata utilizzando OpenAPI, fornendo dettagli su come interagire con le API, comprese le richieste e le possibili risposte.

### 4.1) DIAGRAMMA DI ESTRAZIONE DELLE RISORSE

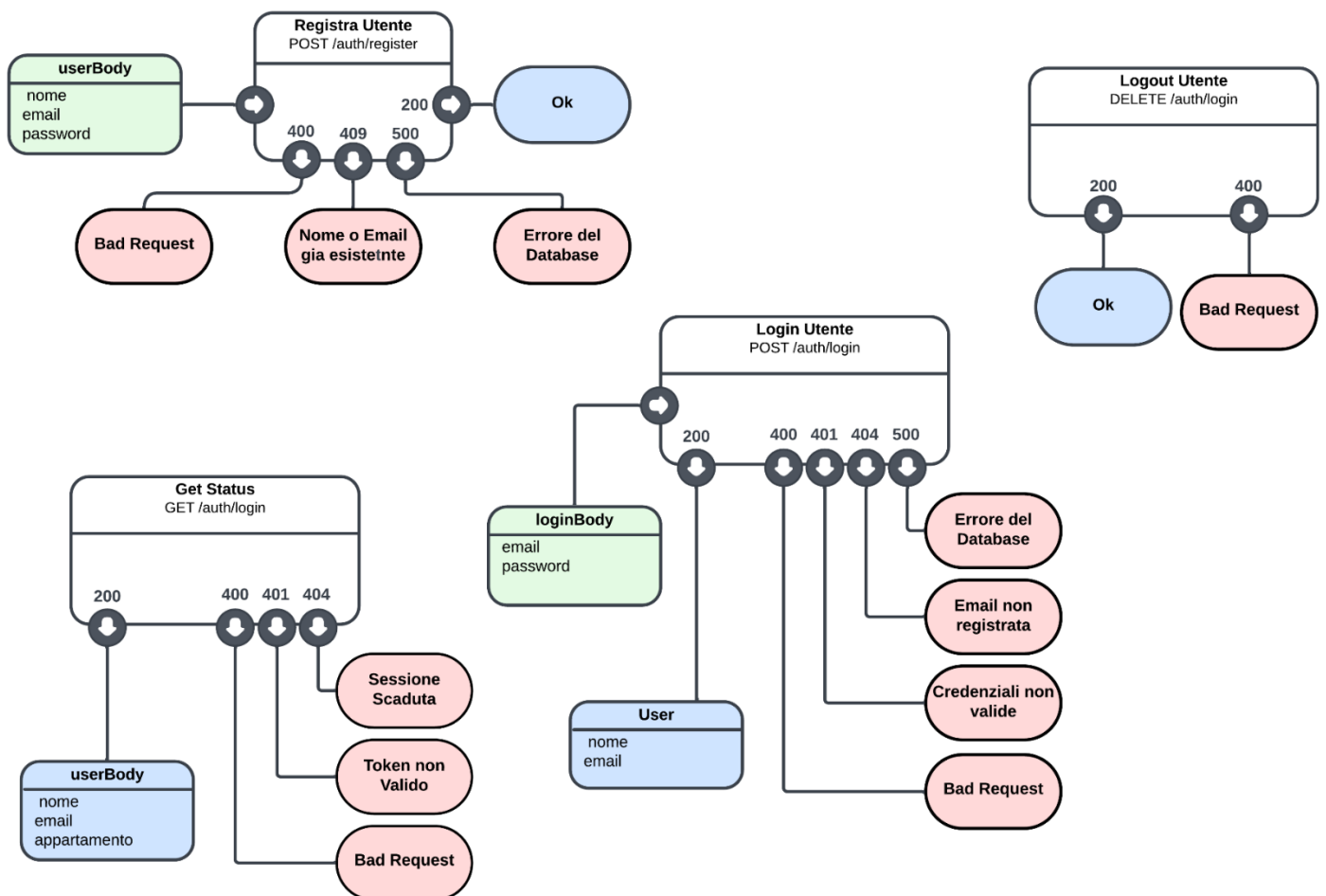
Per la progettazione delle API sono state identificate le principali e necessarie funzionalità da documentare attraverso il diagramma di estrazione delle risorse



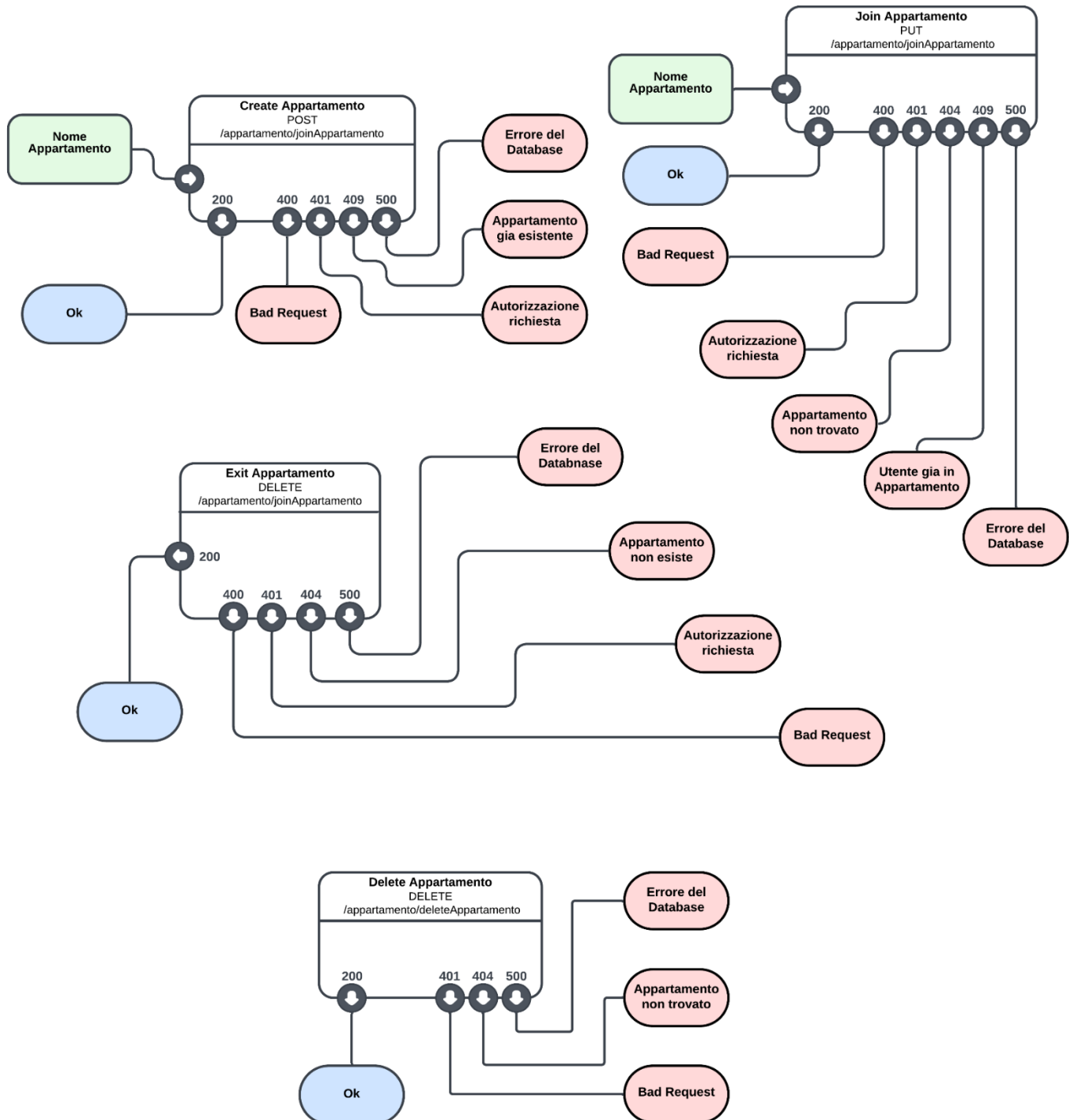
## 4.2) RESOURCE MODELING DIAGRAM

Successivamente è stato elaborato il diagramma di modellazione delle risorse, a partire dal diagramma precedente del punto 4.1. Questo diagramma va a sottolineare in particolar modo le possibili richieste alle varie API con annesse risposte. Per comodità di lettura abbiamo sempre deciso di dividerle in macrocategorie (user auth, apartment settings, apartment content).

### 4.2.1) AUTH RESOURCE MODELING DIAGRAM

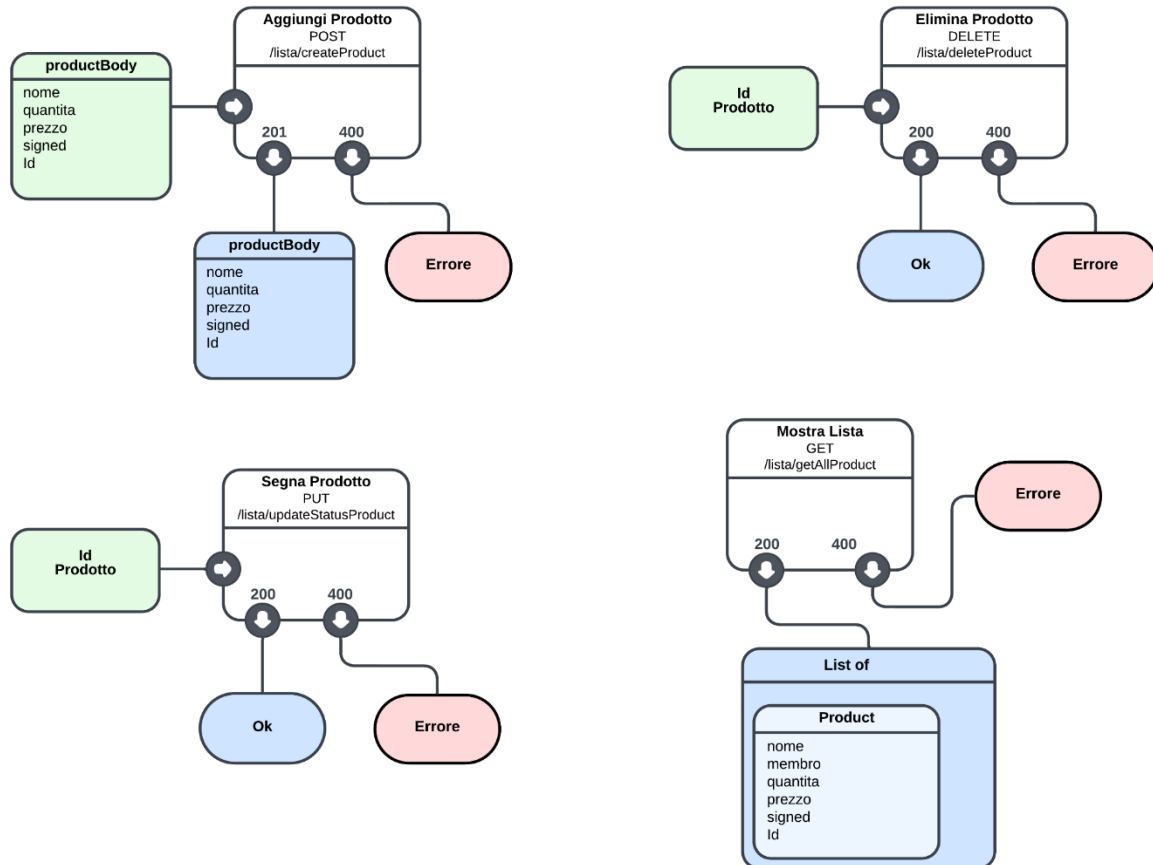


## 4.2.2) APARTMENT RESOURCE MODELING DIAGRAM

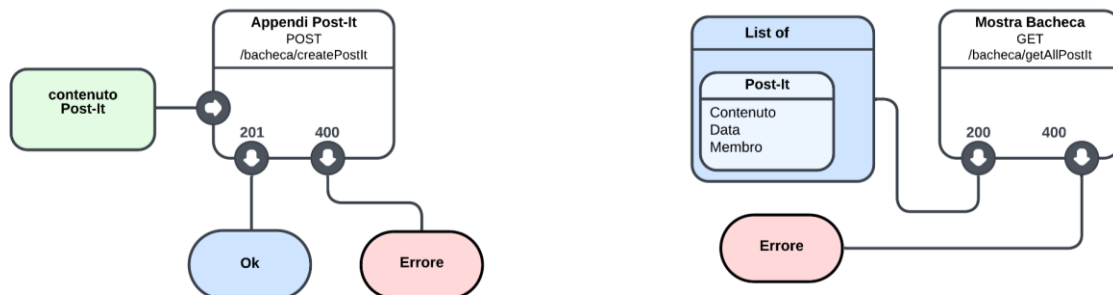


## 4.2.3) APARTMENT CONTENT RESOURCE MODELING DIAGRAM

### Lista della Spesa



### Bacheca



## 4.3) DOCUMENTAZIONE E DESCRIZIONE API

In questa sezione viene riportata la documentazione e la spiegazione testuale delle API usate nel progetto. Sono stati inseriti anche alcuni frammenti di codice, per una comprensione più esaustiva dello stesso.

Per dare un flusso di lettura e comprensione migliore, pur mantenendo una coerenza espositiva, la spiegazione delle API viene divisa in macrocategorie (rappresentata analogamente dalla divisione delle cartelle delle api nel progetto).

### 4.3.1) AUTH

#### LOGIN

L'API di login è progettata per gestire l'accesso degli utenti ai propri account (RF2) utilizzando il metodo POST sull'endpoint `/api/auth/login`. Il funzionamento segue il flusso tipico di un login: l'API riceve le credenziali dell'utente, cioè l'*Email* e la *Password*, e verifica la loro validità. In caso di credenziali corrette, restituisce un esito positivo, in caso contrario, restituisce un esito negativo, generando un errore 400 di BAD REQUEST.

```
} else if (req.method === "POST") {  
  
  // prendo Email e Password dal body della richiesta  
  const { email, password } = req.body as Data;  
  
  // controllo che il email e password siano presenti  
  if (!email || !password) {  
    res.status(400).json({success: false, message: "Bad Request"});  
    return;  
  }  
}
```

Questa API, come tutte le seguenti, si connette al server di mongoDB e accede alle collection per valutare che le credenziali inserite dall'utente siano corrette. Nelle prossime API non verrà spiegato il collegamento al server in quanto è analogo e dunque superfluo.

Collegamento al database di mongoDB in blocco try catch:

```
try {  
  // connessione al database  
  const client = await clientPromise;  
  const db = client.db("roomunity");  
  const user = await db.collection("utenti").findOne({ email: email });  
} catch (error) {  
  //ritorno errore per via del database generato dal blocco try  
  res.status(500).json({success: false, message: "Errore del Database"});  
}
```

Il controllo del login avviene all'interno di un blocco try, dove, utilizzando la libreria bcrypt, viene verificata la corrispondenza tra la *Password* inserita dall'utente e quella associata all'account.

```
// Uso bcryptjs per controllare match della pass  
const passwordMatch = await bcrypt.compare(password, user.hashPassword);  
  
if (!passwordMatch) {  
  res.status(401).json({success: false, message: "Credenziali non valide"});  
  return;  
}
```

L'API inoltre include un metodo **GET** che restituisce i dati dell'utente loggato nella sessione corrente (la sua utilità viene chiarita più avanti, con l'utilizzo dei componenti relativi agli annessi profile), e un metodo **DELETE** utilizzato come *logout*, il quale cancella i cookie (token jwt) cancellando implicitamente la sessione corrente (RF3)



La GET controlla la presenza e la validità del token all'interno dei cookies

```
if (req.method === "GET") {  
    // controllo presenza del token  
    if (token) {  
        // controllo integrita' del token  
        try {
```

La DELETE cancella i cookie e quindi il token:

```
} else if (req.method === "DELETE") {  
    res.setHeader('Set-Cookie', `token=; HttpOnly; Path=/; Expires=${new Date(0).toUTCString()}`)
```

## REGISTER

L'API di registrazione è progettata per gestire la registrazione degli utenti all'applicazione (RF1). Mediante un metodo di tipo **POST** all'endpoint `/api/auth/register`, è possibile inviare tramite la propria richiesta le informazioni necessarie alla creazione del nuovo account: *Nome, Email e Password*.

L'API restituisce un esito positivo se la procedura va a buon fine e le informazioni fornite dall'utente vengono registrate correttamente. Infatti, se l'email non è in un formato valido, la procedura genera un errore.

L'API esegue anche un controllo per verificare se esiste già un altro utente con lo stesso nome utente, che funge da identificativo nell'applicazione, o con la stessa email, anch'essa necessaria per il login e dunque anch'essa necessariamente univoca per garantire la corretta gestione degli account.

```
//Controllo se l'utente esiste già  
const userExists = await collection.findOne({nome: nome});  
const emailExists = await collection.findOne({email: email});  
if (userExists || emailExists) {  
    return res.status(409).json({success: false, message: "nome o email già esistenti"})  
}
```

Successivamente la password dell'utente viene criptata per questioni di sicurezza, usando la libreria di bcrypt. Infine, l'utente viene inserito nella collection del database.

```
//Creo un nuovo utente
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash(password, salt);
const newUser: Utente = { ...
}
// aggiungo l'utente alla collection
collection.insertOne(newUser);
```

## 4.3.2) APPARTAMENTO

### JOIN

L'API di ingresso in appartamento è progettata per gestire lo stato dell'utente nei rapporti dell'appartamento di interesse, ovvero l'ingresso, la creazione e l'uscita dagli stessi appartamenti. Mediante un metodo **PUT** all'endpoint `/api/appartamento/joinAppartamento` l'utente può inserire nella richiesta il nome dell'appartamento in cui vuole entrare. L'API restituirà un esito positivo se l'operazione viene completata con successo, un esito negativo altrimenti.

L'API effettua un controllo sull'esistenza dell'appartamento legato al nome, in caso negativo risulterà un errore.

```
// Controlla se l'appartamento esiste
const appartamentoExists = await collectionAppartamenti.findOne({ nome: apartmentName });
if (!appartamentoExists) {
  return res.status(404).json({ errore: "Appartamento non esiste" });
}

// Aggiorna l'utente per assegnarlo al nuovo appartamento
await collectionUtenti.updateOne(
  { nome: userName },
  { $set: { appartamento: apartmentName } }
);
```

Oltre a permettere l'ingresso in un appartamento esistente, nel caso in cui l'utente non desideri unirsi a uno già esistente, l'API consente, tramite il metodo **POST** all'endpoint `/api/appartamento/joinAppartamento`, di creare un nuovo appartamento (a cui sarà automaticamente aggiunto come membro). L'utente può specificare nella richiesta HTTP il nome desiderato per il nuovo appartamento, che

deve essere univoco. Se l'operazione di creazione avviene con successo, l'API restituirà un esito positivo, altrimenti restituirà un esito negativo.

L'API effettua un controllo sull'esistenza di un appartamento già esistente il cui nome coincide bit a bit con il nome deciso dall'utente, in caso affermativo restituirà un errore.

```
// Controlla se l'appartamento esiste già
const appartamentoExists = await collectionAppartamenti.findOne({ nome });
if (appartamentoExists) {
  return res.status(400).json({ errore: "Appartamento già esistente" });
}
```

Infine, avviene l'aggiunta della nuova istanza appartamento al database.

Infine, l'API gestisce anche l'uscita dell'utente dall'appartamento a cui appartiene. Mediante il metodo **DELETE** all'endpoint /api/appartamento/joinAppartamento, l'utente potrà uscire dall'appartamento. Analogamente alle altre casistiche in caso positivo o negativo si riceverà il relativo esito.

```
// Rimuove l'appartamento dall'utente
await collectionUtenti.updateOne(
  { nome: userName },
  { $set: { appartamento: undefined } }
);
```

Tutti i vari metodi dell'API controllano in primo luogo che l'utente abbia avviato una sessione, ovvero che abbia fatto un login e che quindi il suo token jwt contenuto nei cookie sia ancora valido. In caso alternativo l'esito delle varie operazioni risulta sempre negativo.

```
const token = req.cookies.token;
// controllo che l'utente abbia effettuato il login per
// effettuare la richiesta
if(!token){
  const e: Errore = { errore: "nessuna sessione utente" };
  res.status(404).json(e);
}
```

## DELETE

L'API di delete di un appartamento è progettata per eliminare correttamente un appartamento dall'applicazione (RF14). L'API utilizza un metodo **DELETE** all'endpoint `/api/appartamento/deleteApartamento`.

Successivamente, l'API si presterà a eliminare l'appartamento dal database, questo soltanto dopo che è stato fatto un controllo su tutti gli utenti per toglierli correttamente dall'appartamento. In caso di operazione positiva l'API ritornerà un esito positivo, in caso contrario questa ritornerà un errore con messaggio relativo all'errore che ha causato il fallimento dell'operazione.

```
//elimino dagli altri il proprio appartamento
await collectionUtenti.updateMany(
  {appartamento: utente.appartamento},
  {$set: { appartamento: undefined }}
);

await collectionAppartamenti.deleteOne(
  {nome: utente.appartamento}
);
```

Analogamente alla funzionalità di join, anche questa API richiede un controllo preliminare dell'esistenza e dei dati contenuti nel token JWT della sessione. Se il token JWT è assente o presenta problemi, l'API restituirà un errore, come avviene per l'API precedente. Poiché tutte le operazioni successive richiedono una forma di autenticazione basata su un token JWT valido e corretto, non verrà ripetuta la specifica di questo controllo per ogni singola API.

### 4.3.3) LISTA DELLA SPESA

#### AGGIUNGI

L'API è progettata per l'aggiunta di nuovi prodotti al database. Il processo avviene tramite una richiesta HTTP di tipo **POST** all'endpoint `/api/lista/createProduct`. La richiesta è strutturata in modo tale che sia necessario inserire i campi `nomeProdotto`, `quantità` e `prezzo`. Successivamente, la nuova istanza verrà aggiunta al database se la richiesta è andata a buon fine, altrimenti si avrà un errore.

```
// Estrai i dati del prodotto dal corpo della richiesta
const { nomeProdotto, quantità, prezzo } = req.body;

// Controllo sui parametri
if (!nomeProdotto) { ...
}
if (!nomeProdotto || !quantità || prezzo === undefined) {
  throw new Error("Nome del prodotto, quantità e prezzo sono obbligatori.");
}
```

Il controllo e il relativo messaggio di esito dell'operazione saranno simili anche per le API successive. Pertanto, per evitare di appesantire ulteriormente la documentazione, quella parte di codice non verrà più documentata.

```
// Risposta con tutti i dati inclusi
res.status(201).json({ ...
});
} catch (error: any) {
  console.error('Errore nell\'inserimento del prodotto:', error);
  return res.status(400).json({ success: false, message: error.message });
}
```

## ELIMINA

L'API è progettata per la rimozione di un prodotto dal database. Il processo avviene tramite una richiesta HTTP di tipo **DELETE** all'endpoint `/api/lista/deleteProduct`. La richiesta è strutturata in modo tale che l'ID del prodotto da eliminare debba essere fornito. Successivamente, il metodo eliminerà l'istanza corrispondente a tale ID dal database, se trovata.

```
// Elimina il prodotto dal database
const result = await collectionProdotti.deleteOne({ _id: new ObjectId(productId) });

if (result.deletedCount === 0) {
  throw new Error("Prodotto non trovato o già eliminato");
}
```

## GET ALL

L'API è ideata per la restituzione di tutti i prodotti inseriti nella lista della spesa dell'appartamento. Il processo avviene tramite una richiesta HTTP di tipo **GET** all'endpoint `/api/lista/getAllProduct`.

Il funzionamento dell'API prevede innanzitutto di estrarre il nome dell'utente dal token JWT. Successivamente, l'API individua l'appartamento associato a tale utente e ne identifica tutti i membri. A questo punto, l'API verifica nella collezione dei prodotti tutti quelli aggiunti da uno qualsiasi di questi utenti e restituisce l'elenco risultante.

```
// Recupera l'utente e il suo appartamento
const utente = await collectionUtenti.findOne({ nome: nomeUser });

if (!utente) { ...
}

// mi salvo l'appartamento dell'utente
const appartamento = utente.appartamento;

// Recupera tutti gli utenti dell'appartamento
const utenti = await collectionUtenti.find({ appartamento }).toArray();

// Estrai i nomi degli utenti per la query dei prodotti
const userNames = utenti.map(u => u.nome);

// Recupera tutti i prodotti creati dagli utenti di quell'appartamento
const prodotti = await collectionProdotti.find({ nomeUtente: { $in: userNames } }).toArray();
```



## SEGNA PRODOTTO

L'API è progettata per cambiare lo stato di un prodotto (segnato o meno). Il metodo funziona iniziando una richiesta HTTP di tipo **PUT** all'endpoint `/api/lista/updateStatusProduct`. Questa richiesta richiede l'ID del prodotto da aggiornare, lo cerca nella collezione del database e, se trovato, modifica il valore del campo `already_signed` per riflettere il nuovo stato del prodotto.

## 4.3.4) BACHECA

### CREA

L'API è progettata per l'aggiunta di nuovi Post-It al database. Il processo avviene tramite una richiesta HTTP di tipo **POST** all'endpoint `/api/bacheca/createPostIt`. La richiesta è strutturata in modo tale che sia necessario inserire il contenuto visualizzabile dal Post-It. Successivamente verrà creata l'istanza Post-it, aggiungendo al contenuto dell'utente, la data di pubblicazione e il nome dell'autore. L'oggetto verrà poi inserito nel Database.

```
// Decodifica del token per ottenere il nome utente
const decoded: any = jwt.verify(token, process.env.JWT_SECRET!);
const nomeUser = decoded.nome;

// Estrai i dati del post-it dal corpo della richiesta
const { contenuto } = req.body;

if (!contenuto) { ...
}

// Ottieni la data odierna
const oggi = new Date().toISOString().split('T')[0];

// Creazione dell'oggetto Postit con la data odierna
const newPostit = new Postit(contenuto, oggi, nomeUser);
```

### GET ALL

L'API è ideata per la restituzione di tutti i prodotti inseriti nella lista della spesa dell'appartamento. Il processo avviene tramite una richiesta HTTP di tipo **GET** all'endpoint `/api/lista/getAllProduct`.

Il funzionamento dell'API (ripetendo la performance del precedente GET) funziona in maniera analoga a quella per i prodotti, dunque non verrà documentato ulteriormente.

## 4.4) DOCUMENTAZIONE OPENAPI

La documentazione delle API è stata realizzata tramite la libreria Swagger. All'interno del progetto è presente il file yaml consultabile per ottenere quanto riportato sotto.

### auth Gestione di accesso per gli utenti



**GET** /auth/login Recupera stato utente



**POST** /auth/login Effettua il login dell'utente



**DELETE** /auth/login Cancella lo stato dell'utente



**POST** /auth/signup Effettua la registrazione di un utente



### appartamento Gestione stato in Appartamento



**DELETE** /appartamento/deleteAppartamento Elimina l'appartamento



**POST** /appartamento/joinAppartamento Crea nuovo Appartamento



**PUT** /appartamento/joinAppartamento Entra in un Appartamento



**DELETE** /appartamento/joinAppartamento Esci dall'Appartamento



### lista della spesa Gestione dei Prodotti nella Lista della spesa



**POST** /lista/createProduct Aggiunge prodotto in lista



**DELETE** /lista/deleteProduct Cancella prodotto dalla lista



**GET** /lista/getAllProduct Restituisce tutti i prodotti in lista



**PUT** /lista/updateStatusProduct Cambia stato del prodotto in lista



### bacheca Gestione dei Postit in Bacheca



**POST** /bacheca/createPostIt Aggiunge Post-it in Bacheca



**GET** /bacheca/getAllPostIt Restituisce Post-it in Bacheca





---

Per ogni API si possono ottenere le seguenti informazioni:

- L'**endpoint** da utilizzare per accedervi;
- Il **metodo HTTP** della chiamata;
- I **parametri** che vengono richiesti e dovranno essere passati;
- Le varie tipologie di **risposte** che l'API può fornire con annessa motivazione.

Si è cercato di far funzionare la documentazione per il testing con l'effettivo funzionamento delle API ma abbiamo avuto dei problemi nel generare e salvare il token jwt che moriva periodicamente e quindi andrebbe generato costantemente. Pertanto, le API non sono testabili da qui.

Per consultare la documentazione completa, è presente, all'interno della repository dei deliverables nell'organizzazione GitHub, il file `openapiRoomUnity.yaml`.

## 5) TESTING

Il testing delle API è stato realizzato usando 4 librerie: jest, node-mocks-http, jsonwebtoken e mongodb.

Nella cartella `/src/test` sono presenti tre file di test: `CreateProduct.test.ts`, `getAllProducts.test.ts` e `joinAppartamento.test.ts`. Ognuno di questi corrisponde a una suite di test che copre gruppi di API simili. Ogni suite di test include tanti test case quanti sono i possibili risultati che le API possono restituire.

Jest, il framework principale, gestisce l'organizzazione dei test, simula il comportamento delle funzioni con `jest.mock`, e verifica i risultati attesi tramite `expect`.

`node-mocks-http` è utilizzato per creare mock di richieste e risposte HTTP, simulando il comportamento delle API.

`jsonwebtoken` viene mockato per simulare la verifica del token JWT durante i test relativi all'autenticazione.

Infine, `mongodb` viene utilizzato per interagire con il database MongoDB, permettendo l'esecuzione di operazioni sui dati necessari ai test.

File	% Stmts	% Branch	% Funcs	% Lines
All files	90	85	100	89.92
lib	57.14	25	100	57.14
mongodb.ts	57.14	25	100	57.14
models	100	100	100	100
Prodotto.ts	100	100	100	100
pages/api/appartamento	88.33	81.25	100	88.33
joinAppartamento.ts	88.33	81.25	100	88.33
pages/api/lista	100	100	100	100
createProduct.ts	100	100	100	100
getAllProducts.ts	100	100	100	100
Test Suites: 3 passed, 3 total				
Tests: 20 passed, 20 total				
Snapshots: 0 total				
Time: 7.739 s				
Ran all test suites.				

## 5.1) Documentazione dei Test - GET /api/lista/getAllProducts

Questa sezione descrive i test implementati per verificare il comportamento dell'endpoint GET /api/lista/getAllProducts

1. Restituzione di una lista di prodotti per un utente autenticato tramite JWT valid.
  - a. Obiettivo: verificare che l'API restituisca una lista di prodotti per un utente autenticato tramite JWT valido.
  - b. Scenario: l'utente Luca è autenticato e ha un prodotto associato.
  - c. Mock: viene simulato un token JWT valido con payload {nome: 'Luca'}.
  - d. Verifiche:
    - i. La risposta HTTP deve avere codice di stato 200.
    - ii. La lista dei prodotti deve contenere un elemento.
    - iii. Il prodotto restituito deve avere un \_id lungo 24 caratteri e il nome dell'utente deve essere Luca.
    - iv. Il nome del prodotto deve essere Spaghetti.
2. Errore in assenza di token
  - a. Obiettivo: verificare che l'API restituisca un errore quando non viene fornito alcun token JWT.
  - b. Scenario: nessun token viene incluso nella richiesta.
  - c. Verifiche:
    - i. La risposta HTTP deve avere codice di stato 400.
    - ii. Il messaggio di errore deve indicare "Autenticazione richiesta".
3. Errore per utente non trovato
  - a. Obiettivo: verificare che l'API restituisca un errore se l'utente autenticato non esiste nel database.
  - b. Scenario: Il token JWT contiene il nome di un utente inesistente.
  - c. Mock: il token JWT simula un utente con il nome NonExistingUser.
  - d. Verifiche:
    - i. La Risposta HTTP deve avere codice di stato 400.
    - ii. Il messaggio di errore deve indicare "Utente non trovato".
4. Nessun prodotto trovato per l'utente
  - a. Obiettivo: verificare che l'API restituisca una lista vuota se non vengono trovati prodotti per l'utente autenticato.

- b. Scenario: l'utente Luca è autenticato, ma non ci sono prodotti associati al suo account.
- c. Mock: viene simulato che non ci siano prodotti per l'utente Luca nel database.
- d. Verifiche:
  - i. La risposta HTTP deve avere codice di stato 200.
  - ii. La lista dei prodotti deve essere vuota.

## 6) FRONTEND

Ora verranno descritti in dettaglio i diversi componenti utilizzati per la creazione del FrontEnd dell'applicazione RoomUnity, sviluppati utilizzando il framework React. Il linguaggio utilizzato per sviluppare le varie funzionalità è TypeScript, integrato con JSX e CSS per la resa grafica delle pagine. Per il routing è stato scelto Next.js, un framework basato su React che facilita la gestione delle rotte e il rendering delle pagine.

Ogni componente si riferisce ad una caratteristica specifica del progetto, in modo da rendere il codice più modulare e riutilizzabile.

I componenti sviluppati sono:

- Header.tsx (roomunity\src\components)
- Register.tsx (roomunity\src\components)
- ApartmentCreateComp.tsx (roomunity\src\components)
- ApartmentJoinComp.tsx (roomunity\src\components)
- Login.tsx (roomunity\src\components)
- HomeComp.tsx (roomunity\src\components)
- BachecaComp.tsx (roomunity\src\components)
- SpesaComp.tsx (roomunity\src\components)

Visualizzati poi nelle pagine

- register.tsx (roomunity\src\auth)
- login.tsx (roomunity\src\auth)
- index.tsx(roomunity\src)

## 6.1) COMPONENTS

### Header.tsx

Il componente contenuto in Header.tsx fornisce l'header comune a tutte le pagine dell'applicazione RoomUnity. L'header è composto dal logo e, nel caso in cui l'utente sia autenticato, da un pulsante *logout*. L'header, infatti, gestisce anche il logout dell'utente e il successivo reindirizzamento.

È stato utilizzato un Hook di Next.js per rendere più semplice la gestione del codice. Nello specifico:

- *useRouter* (Next.js), che gestisce il routing e la navigazione all'interno dell'applicazione.

Inoltre, viene utilizzata un'interfaccia TypeScript per definire le proprietà che poi saranno passate come argomento al componente Header. In questo caso, *profile* è una proprietà che definisce se il profilo dell'utente è attualmente loggato nell'applicazione. Può dunque essere *undefined* nel caso in cui non sia loggato o di tipo *Utente* (in accordo con il modello definito in `@/models/Utente`) nel caso in cui abbia effettuato il login.

```
// Interfaccia che definisce le proprietà del componente Header
interface HeaderLVL {
  //profile serve a sapere se l'utente ha effettuato il login
  profile: undefined | Utente;
}
```

C'è una funzione:

- *handleLogout()*, che gestisce il processo di logout dell'utente. Viene chiamata nel momento in cui l'utente preme il pulsante *Logout*. La funzione avvia una chiamata API ad `"/api/auth/login/loginUtente"` con metodo DELETE includendo i cookie nella richiesta.  
Se l'esito della richiesta è positivo (*response.ok*), rimuove i token di autenticazione da `localStorage` e `sessionStorage` e mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuto logout.

```
//Funzione asincrona che gestisce il logout dell'utente
const handleLogout = async () => {
  try {
    //Effettua una richiesta DELETE all'API di login
    const response = await fetch("/api/auth/login/loginUtente", {
      method: "DELETE",
      headers: {
        "Content-Type": "application/json",
      },
      //Include le credenziali (cookie) nella richiesta
      credentials: "include",
    });
  }
};
```

Successivamente, avviene ricarica la pagina (verrà mostrata così la home con la possibilità di effettuare il login o la registrazione).

Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.

```
//Se la risposta dell'API è positiva
if (response.ok) {
  //Rimuove i token di autenticazione da localStorage e sessionStorage
  localStorage.removeItem("authToken");
  sessionStorage.removeItem("authToken");
  //Toast di successo della chiamata API
  toast.success("Logout avvenuto con successo, sarai reindirizzato a breve", {
    autoClose: 1500,
    onClose: () => {
      //Ricarica la pagina dopo la chiusura del toast
      router.reload();
    },
  });
} else {
  //Se la risposta dell'API è negativa mostra un messaggio di errore
  const errorData = await response.json();
  toast.error(errorData.message || "Qualcosa è andato storto", {
    autoClose: 1500
  });
  console.error("Errore durante il logout:", response.statusText);
}
```

Dunque, l'header viene visualizzato in ogni pagina dell'applicazione e contiene il logo dell'applicazione. Inoltre, a seconda che l'utente sia loggato o meno, presenta un tasto di *Logout*.

## Register.tsx

Il componente contenuto in Register.tsx fornisce un'interfaccia utente che si occupa della registrazione degli utenti nell'applicazione. In particolare, gestisce l'inserimento delle credenziali (*Username*, *Email* e *Password*) da parte dell'utente, l'invio delle informazioni di registrazione all'API di registrazione e la risposta della stessa.

Presenta un form con tre campi input per *Username*, *Email* e *Password* e un pulsante *Registrati* di tipo submit.

Sono stati utilizzati degli Hook di Next.js e React per rendere più semplice la gestione del codice. Nello specifico:

- *useRouter* (Next.js), che gestisce il routing e la navigazione all'interno dell'applicazione e, in questo caso, permette di reindirizzare l'utente in seguito ad un login effettuato correttamente.
- *useState* (React), che gestisce lo stato locale del componente, in questo caso le credenziali dell'utente, e il controllo dell'indicatore di caricamento *loading*.

```
//Stato locale per memorizzare le credenziali inserite dall'utente (Nome, Email e Password)
const [formData, setFormData] = useState({
  nome: "",
  email: "",
  password: ""
});

//Stato per gestire l'indicatore di caricamento durante il processo di registrazione
const [loading, setLoading] = useState(false);
```

Ci sono due funzioni:

- *handleChange*(e: *React.ChangeEvent<HTMLInputElement>*), che gestisce i cambiamenti nei campi di input del form aggiornando *formData* con le credenziali inserite dall'utente, utilizzando la funzione *setFormData* (fornita da *useState*).

```
//Funzione che gestisce i cambiamenti nei campi del form di registrazione
const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  const { name, value } = e.target;
  setFormData((prevData) => ({
    ...prevData,
    [name]: value,
  }));
  setError("");
};
```

- *handleRegister()*, che gestisce il processo di registrazione. Viene chiamata nel momento in cui l'utente preme il pulsante *Registrati*. Innanzitutto, verifica che i campi *Username*, *Email* e *Password* siano stati compilati e imposta lo stato *loading* a true per indicare che è in corso il processo di registrazione.

```
//Controllo preliminare per verificare che tutti i campi siano stati compilati
if (!formData.nome || !formData.password || !formData.email) {
  toast.error("Campi incompleti", {
    autoClose: 1500
  });
  return;
}
```

Successivamente, avvia una chiamata API ad `"/api/auth/register/registerUtente"` con metodo POST passando, come payload JSON, le credenziali inserite nel form.

```
//Effettua una richiesta POST all'API di registrazione
const response = await fetch("/api/auth/register/registerUtente", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  //Passa i dati del form come stringa JSON
  body: JSON.stringify(formData),
});
```



Se l'esito della richiesta è positivo (*response.ok*), mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuta registrazione. Successivamente, avviene il reindirizzamento alla pagina di login in */auth/login.tsx*.

Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.

Infine, imposta lo stato di *loading* a false, indipendentemente dal risultato.

```
//Se la risposta dell'API è positiva
if (response.ok) {
  //Toast di successo della chiamata API
  toast.success("Registrazione avvenuta con successo, sarai reindirizzato a breve",
    {
      autoClose: 1500,
      onClose: () => {
        //Reindirizza l'utente alla pagina di login dopo la chiusura del toast
        router.push("/auth/login");
      },
    },
  );
} else {
  //Se la risposta dell'API è negativa mostra un messaggio di errore
  const errorData = await response.json();
  toast.error(errorData.message || "Qualcosa è andato storto", {
    autoClose: 1500
  });
}
```

L'esecuzione del codice avviene nel modo seguente. L'utente inserisce le credenziali *Username*, *Email* e *Password* (se non le inserisce, vedrà un *toast.error*) nel form gestito dalla funzione *handleChange(e:React.ChangeEvent<HTMLInputElement>)* e preme il pulsante *Registrati*. In questo modo, viene chiamata la funzione *handleRegister()*, che imposta *loading* a true e invia le credenziali all'API di registrazione. Se l'API risponde positivamente, viene mostrato un messaggio di successo e l'utente viene reindirizzato alla pagina di login. Se l'API risponde con un errore, viene mostrato un messaggio di errore specifico. Infine, *loading* viene impostato su false.

**Nota:** Verranno omessi nei seguenti componenti gli screenshot del codice relativo agli Hook utilizzati e ai messaggi toast, in quanto analoghi a quelli mostrati in precedenza.

## ApartmentCreateComp.tsx

Il componente contenuto in `ApartmentCreateComp.tsx` fornisce un'interfaccia utente che si occupa della creazione di un appartamento virtuale da parte di un utente registrato all'applicazione. In particolare, gestisce l'inserimento del nome dell'appartamento (*Nome*) da parte dell'utente, l'invio delle informazioni atte alla creazione dell'appartamento all'API di creazione dell'appartamento e la risposta della stessa.

Presenta un form con un campo input per *Nome* e un pulsante *Crea un Appartamento* di tipo submit.

Sono stati utilizzati degli Hook di Next.js e React per rendere più semplice la gestione del codice. Nello specifico:

- *useRouter* (Next.js), che gestisce il routing e la navigazione all'interno dell'applicazione e, in questo caso, permette di reindirizzare l'utente in seguito alla creazione di un appartamento effettuata correttamente.
- *useState* (React), che gestisce lo stato locale del componente, in questo caso il nome dell'appartamento, e il controllo dell'indicatore di caricamento *loading*.

Ci sono due funzioni:

- *handleChange*(*e: React.ChangeEvent<HTMLInputElement>*), che gestisce i cambiamenti nel campo di input del form aggiornando *formData* con il nome dell'appartamento inserito dall'utente, utilizzando la funzione *setFormData* (fornita da *useState*).

```
//Funzione che gestisce i cambiamenti nel campo del form di creazione di un appartamento
const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  const { name, value } = e.target;
  setFormData((prevData) => ({
    ...prevData,
    [name]: value,
  }));
  setError("");
};
```

- *handleCreate*(*e: React.MouseEvent<HTMLButtonElement>*), che gestisce il processo di creazione dell'appartamento. Viene chiamata nel momento in cui l'utente preme il pulsante *Crea un Appartamento*. Innanzitutto, verifica che il campo *Nome* sia stato compilato e imposta lo stato *loading* a true per indicare che è in corso il processo di creazione dell'appartamento. Successivamente,

avvia una chiamata API ad “`/api/appartamento/joinAppartamento`” con metodo POST passando, come payload JSON, il nome dell’appartamento inserito nel form.

```
//Effettua una richiesta POST all'API di creazione di un appartamento
const response = await fetch("/api/appartamento/joinAppartamento", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  //Passa i dati del form come stringa JSON
  body: JSON.stringify(formData),
});
```

Se l’esito della richiesta è positivo (*response.ok*), mostra all’utente, attraverso un *toast.success*, un messaggio di avvenuta creazione dell’appartamento. Successivamente, avviene il reindirizzamento alla pagina home in *index.tsx*. Se l’esito della richiesta è negativo, mostra all’utente un *toast.error* contenente il messaggio di errore ricevuto dall’API.

Infine, imposta lo stato di *loading* a false, indipendentemente dal risultato.

L’esecuzione del codice avviene nel modo seguente. L’utente inserisce il *Nome* dell’appartamento (se non le inserisce, vedrà un *toast.error*) nel form gestito dalla funzione *handleChange(e:React.ChangeEvent<HTMLInputElement>)* e preme il pulsante *Crea un Appartamento*. In questo modo, viene chiamata la funzione *handleCreate()*, che imposta *loading* a true e invia il nome inserito all’API di creazione di un appartamento. Se l’API risponde positivamente, viene mostrato un messaggio di successo e l’utente viene reindirizzato alla home. Se l’API risponde con un errore, viene mostrato un messaggio di errore specifico. Infine, *loading* viene impostato su false.

## ApartmentJoinComp.tsx

Il componente contenuto in *ApartmentJoinComp.tsx* fornisce un’interfaccia utente che si occupa dell’ingresso in un appartamento virtuale già esistente da parte di un utente registrato all’applicazione. In particolare, gestisce l’inserimento del nome dell’appartamento (*Nome*) da parte dell’utente, l’invio delle informazioni atte

all'ingresso in un appartamento all'API di ingresso in un appartamento e la risposta della stessa.

Presenta un form con un campo input per *Nome* e un pulsante *Entra in Appartamento* di tipo submit.

Sono stati utilizzati degli Hook di Next.js e React per rendere più semplice la gestione del codice. Nello specifico:

- *useRouter* (Next.js), che gestisce il routing e la navigazione all'interno dell'applicazione e, in questo caso, permette di reindirizzare l'utente in seguito all'ingresso in un appartamento effettuato correttamente.
- *useState* (React), che gestisce lo stato locale del componente, in questo caso il nome dell'appartamento, e il controllo dell'indicatore di caricamento *loading*.

Ci sono due funzioni:

- *handleChange*(*e: React.ChangeEvent<HTMLInputElement>*), che gestisce i cambiamenti nel campo di input del form aggiornando *formData* con il nome dell'appartamento inserito dall'utente, utilizzando la funzione *setFormData* (fornita da *useState*).

```
//Funzione che gestisce i cambiamenti nel campo del form di associazione ad un appartamento
const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  const { name, value } = e.target;
  setFormData((prevData) => ({
    ...prevData,
    [name]: value,
  }));
  setError("");
};
```

- *handleJoin*(*e: React.MouseEvent<HTMLButtonElement>*), che gestisce il processo di ingresso in un appartamento. Viene chiamata nel momento in cui l'utente preme il pulsante *Entra in Appartamento*. Innanzitutto, verifica che il campo *Nome* sia stato compilato e imposta lo stato *loading* a true per indicare che è in corso il processo di ingresso in un appartamento. Successivamente, avvia una chiamata API ad `"/api/appartamento/joinAppartamento"` con metodo PUT passando, come payload JSON, il nome dell'appartamento inserito nel form.

```
//Effettua una richiesta PUT all'API di associazione ad un appartamento
const response = await fetch("/api/appartamento/joinAppartamento", {
  method: "PUT",
  headers: {
    "Content-Type": "application/json",
  },
  //Passa i dati del form come stringa JSON
  body: JSON.stringify(formData),
});
```

Se l'esito della richiesta è positivo (*response.ok*), mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuto ingresso nell'appartamento. Successivamente, avviene il reindirizzamento alla pagina home in *index.tsx*. Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API. Infine, imposta lo stato di *loading* a false, indipendentemente dal risultato.

L'esecuzione del codice avviene nel modo seguente. L'utente inserisce il *Nome* dell'appartamento (se non le inserisce, vedrà un *toast.error*) nel form gestito dalla funzione *handleChange(e:React.ChangeEvent<HTMLInputElement>)* e preme il pulsante *Entra in Appartamento*. In questo modo, viene chiamata la funzione *handleJoin()*, che imposta *loading* a true e invia il nome inserito all'API di ingresso in un appartamento. Se l'API risponde positivamente, viene mostrato un messaggio di successo e l'utente viene reindirizzato alla home. Se l'API risponde con un errore, viene mostrato un messaggio di errore specifico. Infine, *loading* viene impostato su false.

## Login.tsx

Il componente contenuto in *Login.tsx* fornisce un'interfaccia utente che si occupa dell'autenticazione degli utenti nell'applicazione. In particolare, gestisce l'inserimento delle credenziali (*Email* e *Password*) da parte dell'utente, l'invio delle informazioni di login all'API di login e la risposta della stessa.

Presenta un form con due campi input per *Email* e *Password* e un pulsante *Login* di tipo submit.

Sono stati utilizzati degli Hook di Next.js e React per rendere più semplice la gestione del codice. Nello specifico:

- *useRouter* (Next.js), che gestisce il routing e la navigazione all'interno dell'applicazione e, in questo caso, permette di reindirizzare l'utente in seguito ad un login effettuato correttamente.
- *useState* (React), che gestisce lo stato locale del componente, in questo caso le credenziali dell'utente, e il controllo dell'indicatore di caricamento *loading*.

Ci sono due funzioni:

- *handleChange(e: React.ChangeEvent<HTMLInputElement>)*, che gestisce i cambiamenti nei campi di input del form aggiornando *formData* con le credenziali inserite dall'utente, utilizzando la funzione *setFormData* (fornita da *useState*).

```
//Funzione che gestisce i cambiamenti nei campi del form di login
const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  const { name, value } = e.target;
  setFormData((prevData) => ({
    ...prevData,
    [name]: value,
  }));
};
```

- *handleLogin()*, che gestisce il processo di login. Viene chiamata nel momento in cui l'utente preme il pulsante *Login*. Innanzitutto, verifica che i campi *Email* e *Password* siano stati compilati e imposta lo stato *loading* a true per indicare che è in corso il processo di login. Successivamente, avvia una chiamata API ad *"/api/auth/login/loginUtente"* con metodo POST passando, come payload JSON, le credenziali inserite nel form.

```
//Effettua una richiesta POST all'API di login
const response = await fetch("/api/auth/login/loginUtente", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  //Passa i dati del form come stringa JSON
  body: JSON.stringify(formData),
});
```

Se l'esito della richiesta è positivo (*response.ok*), mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuto login. Successivamente, avviene il reindirizzamento alla pagina home in *index.tsx*.

Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.

Infine, imposta lo stato di *loading* a *false*, indipendentemente dal risultato.

L'esecuzione del codice avviene nel modo seguente. L'utente inserisce le credenziali Email e Password (se non le inserisce, vedrà un *toast.error*) nel form gestito dalla funzione *handleChange(e:React.ChangeEvent<HTMLInputElement>)* e preme il pulsante *Login*. In questo modo, viene chiamata la funzione *handleLogin()*, che imposta *loading* a *true* e invia le credenziali all'API di login. Se l'API risponde positivamente, viene mostrato un messaggio di successo e l'utente viene reindirizzato alla home. Se l'API risponde con un errore, viene mostrato un messaggio di errore specifico. Infine, *loading* viene impostato su *false*.

## HomeComp.tsx

Il componente contenuto in *HomeComp.tsx* fornisce un'interfaccia utente che si occupa della pagina home visualizzata dall'utente a seconda del suo stato di autenticazione e della sua associazione con un appartamento. Inoltre, gestisce l'eliminazione di un appartamento e l'uscita di un utente da un appartamento.

Sono stati utilizzati degli Hook di Next.js e React per rendere più semplice la gestione del codice. Nello specifico:



- *useRouter* (Next.js), che gestisce il routing e la navigazione all'interno dell'applicazione.
- *useState* (React), per gestire lo stato locale del componente, come la visualizzazione del popup di conferma.

Inoltre, viene utilizzata un'interfaccia TypeScript per definire le proprietà che poi saranno passate come argomento al componente *HomeComp*. In questo caso, *profile* è una proprietà che definisce se il profilo dell'utente è attualmente loggato nell'applicazione. Può dunque essere *undefined* nel caso in cui non sia loggato o di tipo *Utente* (in accordo con il modello definito in `@/models/Utente`) nel caso in cui abbia effettuato il login.

```
// Interfaccia che definisce le proprietà del componente HomeComp
interface HomeProps{
  //profile serve a sapere se l'utente ha effettuato il login
  profile: undefined | Utente;
}
```

Dunque, il componente assume comportamenti diversi a seconda del valore di *profile*:

```
<div className="login-container d-flex align-items-center">
  {/* Controlla se l'utente è autenticato (profile è definito) */}
  {profile ? (
    //Se l'utente è autenticato, controlla se è già associato a un appartamento
    profile.appartamento ? (
      //Se l'utente è membro di un appartamento
```

- se è di tipo *Utente*, l'utente è autenticato. In questo caso il componente effettua un'altra verifica, ossia se l'utente è associato ad un appartamento o meno a seconda del valore di *profile.appartamento*:
  - se *profile.appartamento* è definito, l'utente è già membro di un appartamento e dunque viene mostrata un'interfaccia con le funzionalità dell'applicazione attraverso l'utilizzo di *BachecaComp.tsx* e *SpesaComp.tsx*. Inoltre, vengono visualizzati due pulsanti: *ESCI* e *ELIMINA*. Essi permettono di uscire o eliminare l'appartamento in cui si trova l'utente (le funzioni saranno spiegate successivamente).



- se *profile.appartamento* non è definito, viene mostrata un'interfaccia con la doppia possibilità per l'utente di creare un nuovo appartamento o effettuare l'ingresso in un appartamento già esistente attraverso l'utilizzo di *ApartmentCreateComp.tsx* e *ApartmentJoinComp.tsx*.
- se è *undefined*, l'utente non è autenticato e dunque viene mostrata una pagina di benvenuto con la possibilità per l'utente di andare alla pagina di login in */auth/login.tsx* premendo sul pulsante *Login* o alla pagina di registrazione in */auth/register* premendo sul pulsante *Registrati*.

Sono presenti tre funzioni:

- *handleDelete()*, che gestisce l'eliminazione dell'appartamento in cui si trova l'utente. Viene chiamata nel momento in cui l'utente preme il pulsante *ELIMINA*. Invia una chiamata API a *"/api/appartamento/deleteAppartamento"* con metodo DELETE passando, come payload JSON, le credenziali.  
Se l'esito della richiesta è positivo (*response.ok*), mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuta eliminazione dell'appartamento. Successivamente, la pagina viene ricaricata per riflettere l'aggiornamento.  
Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.
- *handleExit()*, che gestisce l'uscita dell'utente dall'appartamento. Viene chiamata nel momento in cui l'utente preme il pulsante *ESCI*. Invia una chiamata API a *"/api/appartamento/joinAppartamento"* con metodo DELETE passando, come payload JSON, le credenziali.  
Se l'esito della richiesta è positivo (*response.ok*), mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuta uscita dell'appartamento. Successivamente, la pagina viene ricaricata per riflettere l'aggiornamento.  
Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.
- *openPopup()* e *closePopup()*. che gestiscono la visualizzazione di un popup di conferma per l'eliminazione dell'appartamento. *openPopup()* mostra il popup e *closePopup()* lo chiude.

L'esecuzione del codice avviene nel modo seguente. Se l'utente è autenticato e associato ad un appartamento, viene visualizzata la home con le funzionalità

dell'appartamento: la Lista della Spesa e i Post-It. Se l'utente è autenticato ma non è associato ad un appartamento, viene visualizzata la home con la possibilità di creare un appartamento o di unirsi ad un appartamento già esistente. Se l'utente non è autenticato, viene visualizzata la home con la possibilità di andare alla pagina di login o a quella di registrazione. In quest'ultimo caso, l'utente ha anche la possibilità, premendo i pulsanti *ESCI* o *ELIMINA*, di abbandonare o eliminare l'appartamento. Quando l'utente preme il pulsante *ELIMINA*, si apre un popup di conferma. Se conferma l'eliminazione, la funzione *handleDelete()* viene chiamata, cancellando l'appartamento. Quando l'utente preme il pulsante *ESCI*, viene chiamata *handleExit()* per rimuovere l'utente dall'appartamento.

## BachecaComp.tsx

Il componente contenuto in *BachecaComp.tsx* fornisce un'interfaccia utente che si occupa della gestione della bacheca contenente i Post-It. In particolare, gestisce l'inserimento di nuovi Post-It da parte dell'utente e la visualizzazione di quelli postati (da lui o da altri utenti nell'appartamento), l'invio delle informazioni relative ai Post-It rispettivamente all'API di creazione dei Post-It e di recupero dei Post-It e le risposte delle stesse.

Presenta un form con un campo input *Scrivi un post-it...* e un pulsante *Invia*, oltre a un contenitore che mostra i post-it precedenti.

Sono stati utilizzati degli Hook di Next.js e React per rendere più semplice la gestione del codice. Nello specifico:

- *useRouter* (Next.js), che gestisce il routing e la navigazione all'interno dell'applicazione.
- *useState* (React), che gestisce lo stato locale del componente, in questo caso i post-it e il loro contenuto.
- *useEffect* (React), per eseguire una funzione al caricamento del componente che si occupa di recuperare i post-it esistenti.

```
//Effetto per chiamare fetchPostits al caricamento del componente
useEffect(() => {
  fetchPostits();
}, []);
```

Ci sono due funzioni:

- *addPost()*, che gestisce la creazione di nuovi post-it. Viene chiamata nel momento in cui l'utente preme il pulsante *Invia*. Innanzitutto, verifica che il campo di testo sia stato compilato. Successivamente, invia una chiamata API a *"/api/post-it/createPostIt"* con metodo POST passando, come payload JSON, il contenuto del post-it.

```
try {  
  //Effettua una richiesta POST all'API di creazione dei post-it  
  const response = await fetch("/api/post-it/createPostIt", {  
    method: "POST",  
    headers: {  
      "Content-Type": "application/json",  
    },  
    //Passa i dati del form come stringa JSON  
    body: JSON.stringify({ contenuto: postContent }),  
  });
```

Se l'esito della richiesta è positivo (*response.ok*), mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuta creazione del post-it.

Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.

- *fetchPostits()*, che viene chiamata al caricamento del componente (grazie a *useEffect*) e recupera i post-it esistenti tramite una chiamata API a *"/api/post-it/getAllPostIt"* con metodo GET.

```
//Funzione asincrona per recuperare i post-it  
const fetchPostits = async () => {  
  try {  
    //Effettua una richiesta GET all'API per ottenere tutti i post-it  
    const response = await fetch("/api/post-it/getAllPostIt");
```

Se l'esito della richiesta è positivo (*response.ok*), aggiorna lo stato con i post-it recuperati.

Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.

L'esecuzione del codice avviene nel modo seguente. Al caricamento del componente, la funzione *fetchPostits()* viene eseguita e recupera tutti i post-it della bacheca tramite una chiamata API. Quando l'utente inserisce un testo nell'area di input e preme il pulsante *Invia*, viene chiamata la funzione *addPost()* che aggiunge il nuovo post-it e aggiorna la bacheca visualizzata tramite una chiamata API. Se le API rispondono positivamente, viene mostrato un messaggio di successo. Se le API rispondono con un errore, viene mostrato un messaggio di errore specifico.

### SpesaComp.tsx

Il componente contenuto in *SpesaComp.tsx* fornisce un'interfaccia utente che si occupa della gestione della lista della spesa. In particolare, gestisce l'inserimento, l'eliminazione e la visualizzazione dei prodotti nella lista della spesa da parte dell'utente, l'invio delle informazioni relative ai prodotti rispettivamente all'API di creazione dei prodotti, di recupero dei prodotti e di cancellazione dei prodotti e le risposte delle stesse.

Presenta un form con i campi *Nome prodotto*, *quantità* e *prezzo* e un pulsante *Aggiungi prodotto*, oltre a mostrare l'intera lista della spesa con un pulsante *Elimina* per ogni prodotto presente.

Sono stati utilizzati degli Hook di Next.js e React per rendere più semplice la gestione del codice. Nello specifico:

- *useRouter* (Next.js), che gestisce il routing e la navigazione all'interno dell'applicazione.
- *useState* (React), che gestisce lo stato locale del componente, in questo caso i prodotti della lista della spesa, il nome del prodotto, la quantità e il prezzo.
- *useEffect* (React), per eseguire una funzione al caricamento del componente che recupera i prodotti esistenti nella lista della spesa.

```
//Effetto per chiamare fetchPostits al caricamento del componente
useEffect(() => {
  | fetchProducts();
}, []);
```

Ci sono cinque funzioni:

- *addProduct()*, che gestisce l'inserimento di nuovi prodotti nella lista. Viene chiamata nel momento in cui l'utente preme il pulsante *Aggiungi Prodotto*. Innanzitutto, verifica che i campi *Nome prodotto*, *quantità* e *prezzo* siano stati compilati. Successivamente, invia una chiamata API a *"/api/lista/createProduct"* con metodo POST passando, come payload JSON, il nome prodotto, la quantità e il prezzo inseriti nel form.

```
try {  
  //Effettua una richiesta POST all'API di creazione di un prodotto  
  const response = await fetch("/api/lista/createProduct", {  
    method: "POST",  
    headers: {  
      "Content-Type": "application/json",  
    },  
    //Passa i dati del form come stringa JSON  
    body: JSON.stringify({  
      nomeProdotto,  
      quantita,  
      prezzo,  
    }),  
  });  
}
```

Se l'esito della richiesta è positivo (*response.ok*), mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuta aggiunta del prodotto.

Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.

- *deleteProduct()*, che gestisce l'eliminazione dei prodotti presenti nella lista. Viene chiamata nel momento in cui l'utente preme il pulsante *Elimina*. Invia una chiamata API a *"/api/lista/deleteProduct"* con metodo DELETE passando, come payload JSON, l'id del prodotto da eliminare.

```
//Funzione asincrona per gestire l'eliminazione di un prodotto
const deleteProduct = async (productId: string) => {
  try {
    //Effettua una richiesta DELETE all'API di eliminazione di un prodotto
    const response = await fetch("/api/lista/deleteProduct", {
      method: "DELETE",
      headers: {
        "Content-Type": "application/json",
      },
      //Passa il productId come stringa JSON
      body: JSON.stringify({ productId }),
    });
  }
};
```

Se l'esito della richiesta è positivo (*response.ok*), mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuta eliminazione del prodotto.

Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.

- *fetchProducts()*, che viene chiamata al caricamento del componente (grazie a *useEffect*) e recupera i prodotti presenti nella lista della spesa tramite una chiamata API a *"/api/lista/getAllProducts"* con metodo GET.

Se l'esito della richiesta è positivo (*response.ok*), aggiorna lo stato con i prodotti recuperati.

```
//Funzione asincrona per recuperare i post-it
const fetchProducts = async () => {
  try {
    //Effettua una richiesta GET all'API per ottenere tutti i prodotti
    const response = await fetch("/api/lista/getAllProducts");
    const data = await response.json();
  }
};
```

Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.

- *toggleProductStatus()*, che viene chiamata nel momento in cui l'utente interagisce con la spunta accanto a un prodotto e si occupa di segnalarlo come acquistato. Invia una chiamata API a */api/lista/updateStatusProduct* con metodo PUT, passando, come payload JSON, l'id del prodotto da aggiornare.

```
//Funzione asincrona per gestire lo stato di acquisto di un prodotto (già acquistato/non acquistato)
const toggleProductStatus = async (productId: string) => {
  try {
    //Effettua una richiesta PUT all'API per aggiornare lo stato di un prodotto
    const response = await fetch("/api/lista/updateStatusProduct", {
      method: "PUT",
      headers: {
        "Content-Type": "application/json",
      },
      //Passa il productId come stringa JSON
      body: JSON.stringify({ productId }),
    });
  }
};
```

Se l'esito della richiesta è positivo (*response.ok*), mostra all'utente, attraverso un *toast.success*, un messaggio di avvenuto aggiornamento dello stato del prodotto e aggiorna lo stato locale della lista dei prodotti invertendo il valore del campo *already\_signed* (acquistato/non acquistato) per il prodotto specifico.

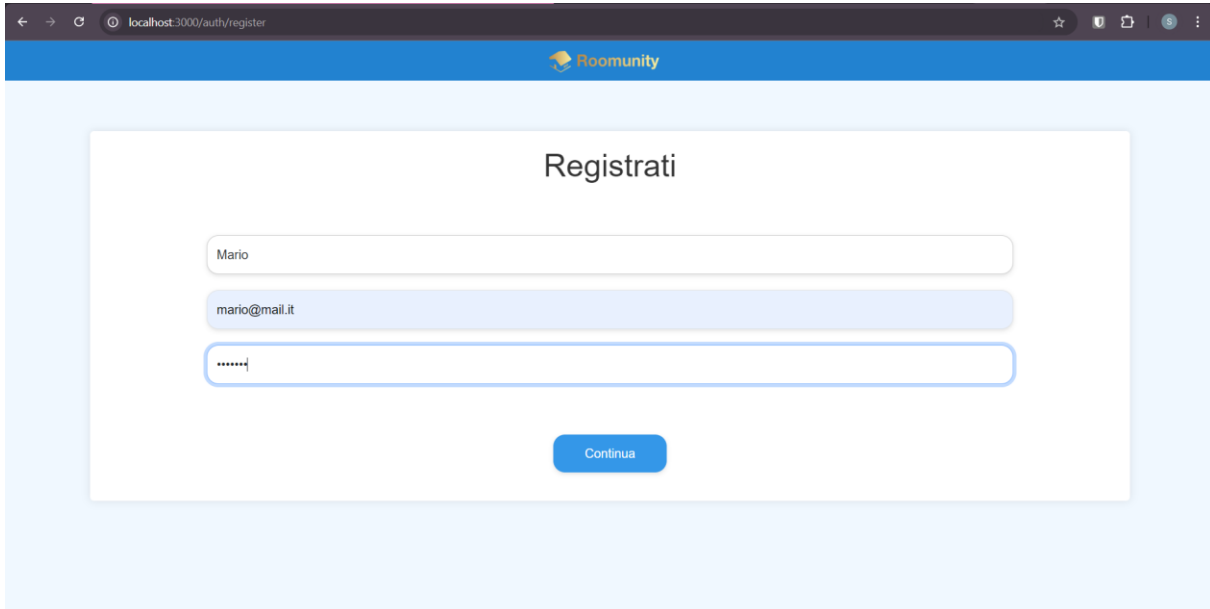
```
// Aggiorna lo stato di acquisto del prodotto nello stato locale
setProdotti((prev) =>
  prev.map((p) =>
    p._id?.toString() === productId ? { ...p, already_signed: !p.already_signed } : p
  )
);
```

Se l'esito della richiesta è negativo, mostra all'utente un *toast.error* contenente il messaggio di errore ricevuto dall'API.

L'esecuzione del codice avviene nel modo seguente. Al caricamento del componente, la funzione *fetchPostits()* viene eseguita e recupera tutti i prodotti presenti nella lista della spesa tramite una chiamata API. Quando l'utente inserisce *Nome prodotto*, *quantità* e *prezzo* nel form e preme il pulsante *Aggiungi Prodotto*, viene chiamata la funzione *addProduct()*, che aggiunge il nuovo prodotto e aggiorna la lista della spesa visualizzata tramite una chiamata API. Quando l'utente preme il pulsante *Elimina* a fianco a un prodotto presente nella lista della spesa viene chiamata la funzione *toggleProductStatus()*, che segna il prodotto come acquistato e aggiorna la lista della spesa visualizzata tramite una chiamata API. Quando l'utente preme la spunta a fianco a un prodotto presente nella lista della spesa viene chiamata la funzione *deleteProduct()*, che rimuove il prodotto e aggiorna la lista della spesa visualizzata tramite una chiamata API. Se le API rispondono positivamente, viene mostrato un messaggio di successo. Se le API rispondono con un errore, viene mostrato un messaggio di errore specifico.

## 6.2) PAGINE

### Register.tsx



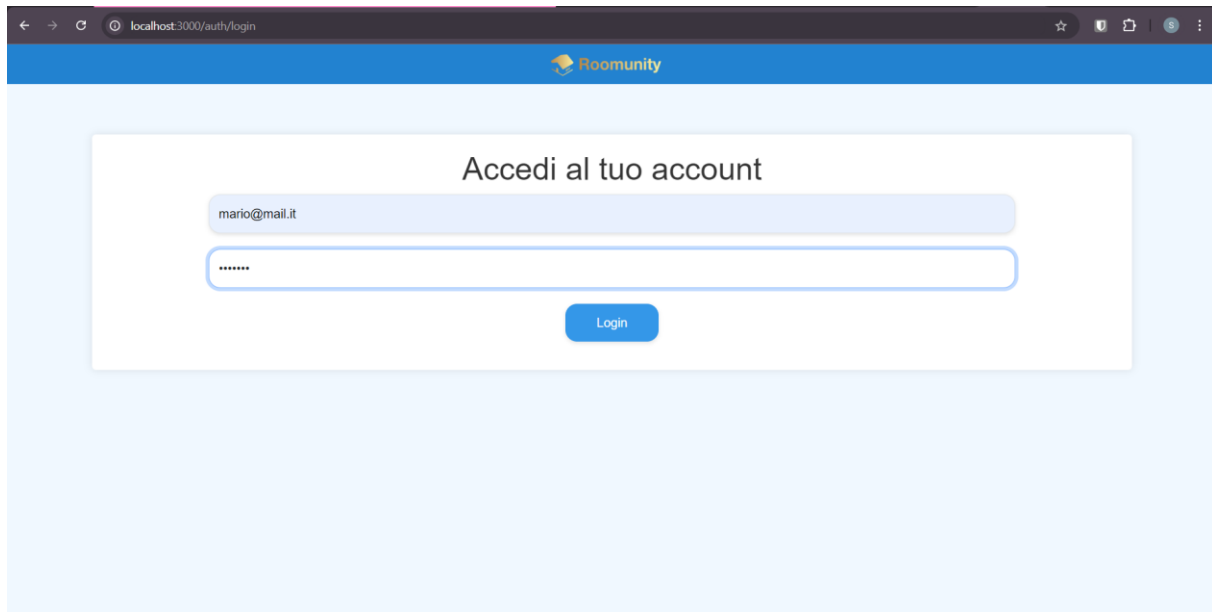
La pagina Register.tsx rappresenta la pagina di registrazione dell'applicazione. Essa include l'Header e il RegisterComp descritti in precedenza. In questo caso, l'Header mostra solamente il logo, in quanto l'utente non è ancora autenticato (dunque *profile* è *undefined*), mentre il RegisterComp fornisce un form in cui l'utente può inserire le proprie credenziali (*Nome*, *Email* e *Password*) e un pulsante *Login* per effettuare il login.

Se la registrazione avviene con successo l'utente visualizza un *toast.success*. Viceversa, se ci sono problemi durante la registrazione, l'utente visualizza un *toast.error* contenente un messaggio di errore specifico.

Si tratta dunque di un'interfaccia utente semplice, ma efficace, che consente agli utenti di inserire le proprie credenziali e effettuare la registrazione.



## Login.tsx



La pagina Login.tsx rappresenta la pagina di login dell'applicazione. Essa include l'Header e il LoginComp descritti in precedenza. In questo caso, l'Header mostra solamente il logo, in quanto l'utente non è ancora autenticato (dunque *profile* è *undefined*), mentre il LoginComp fornisce un form in cui l'utente può inserire le proprie credenziali (*Email* e *Password*) e un pulsante *Login* per effettuare il login.

Se il login avviene con successo l'utente visualizza un *toast.success*. Viceversa, se ci sono problemi durante il login, l'utente visualizza un *toast.error* contenente un messaggio di errore specifico.

Si tratta dunque di un'interfaccia utente semplice, ma efficace, che consente agli utenti di inserire le proprie credenziali e effettuare il login.

## Index.tsx

La pagina Index.tsx rappresenta la pagina home dell'applicazione, ossia il cuore di RoomUnity. Essa include l'Header e l'HomeComp descritti in precedenza. In questo caso, è necessario verificare se l'utente è autenticato o meno. È dunque stato utilizzato un Hook di React per rendere più semplice la gestione del codice. Nello specifico:

- *useState* (React), che gestisce lo stato locale del componente, in questo caso il profilo dell'utente.

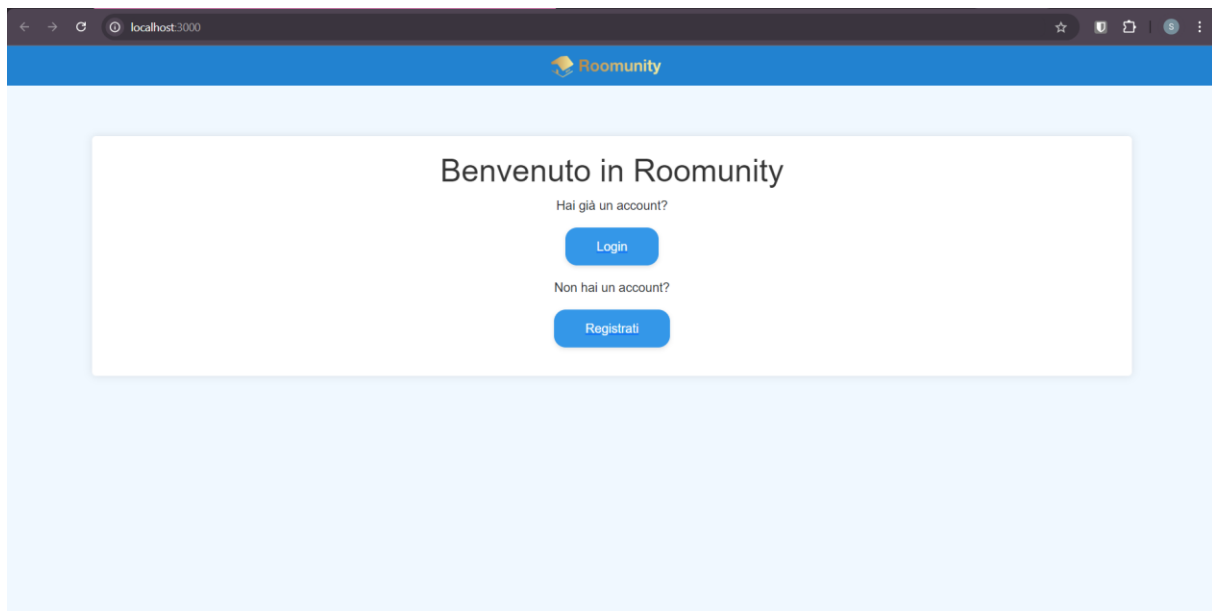
- *useEffect* (React), che gestisce l'operazione di verifica dell'autenticazione dell'utente al caricamento della pagina attraverso una chiamata API a `"/api/auth/login/loginUtente"` con metodo GET.

Se la risposta dell'API è positiva, significa che l'utente è autenticato e *profile* viene impostato con i dati ottenuti come payload JSON.

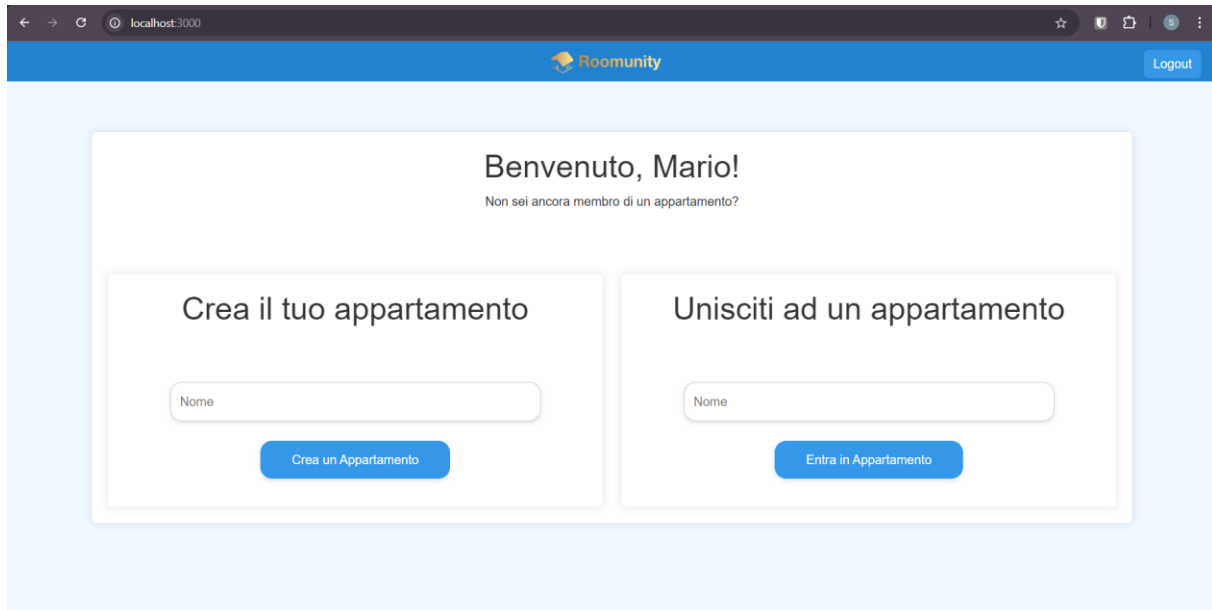
Se la risposta dell'API è negativa, significa che l'utente non si è autenticato e dunque *profile* viene impostato su `undefined`.

Successivamente, l'Header prende come argomento *profile* e dunque, a seconda che l'utente sia autenticato o meno, mostra il pulsante *Logout*. Anche l'HomeComp restituisce una pagina diversa a seconda che l'utente sia autenticato o meno, come descritto in precedenza.

### Utente non autenticato



## Utente autenticato non ancora in un Appartamento



The screenshot shows a web browser window with the URL `localhost:3000`. The page has a blue header with the "RoomUnity" logo and a "Logout" button. The main content area is white and contains a greeting "Benvenuto, Mario!" followed by the text "Non sei ancora membro di un appartamento?". Below this, there are two side-by-side white boxes. The left box is titled "Crea il tuo appartamento" and contains a text input field labeled "Nome" and a blue button labeled "Crea un Appartamento". The right box is titled "Unisciti ad un appartamento" and contains a text input field labeled "Nome" and a blue button labeled "Entra in Appartamento".

## Utente autenticato in un Appartamento



The screenshot shows a web browser window with the URL `localhost:3000`. The page has a blue header with the "RoomUnity" logo and a "Logout" button. The main content area is white and contains a greeting "Benvenuto in Appartamento Mario, Bruno!" followed by the text "Sei membro di un appartamento!". Below this, there is a "Post-it" section. It features three yellow sticky notes with the following content: "Ciao!" by Mario on 2024-09-05, "Ciao, sono Bruno, un nuovo inquilino!" by Bruno on 2024-09-05, and "Ciao a tutti!" by Bruno on 2024-09-05. Below the sticky notes is a text input field labeled "Scrivi un post-it..." and a blue button labeled "Invia".

localhost:3000

Lista Spesa

**Prodotto: Melanzane**  
Quantità: 2  
Prezzo: 5€  
Creato da: Bruno

Segna come comprato ☒ [Elimina](#)

**Prodotto: Kiwi**  
Quantità: 2  
Prezzo: 3€  
Creato da: Bruno

Segna come comprato ☐ [Elimina](#)

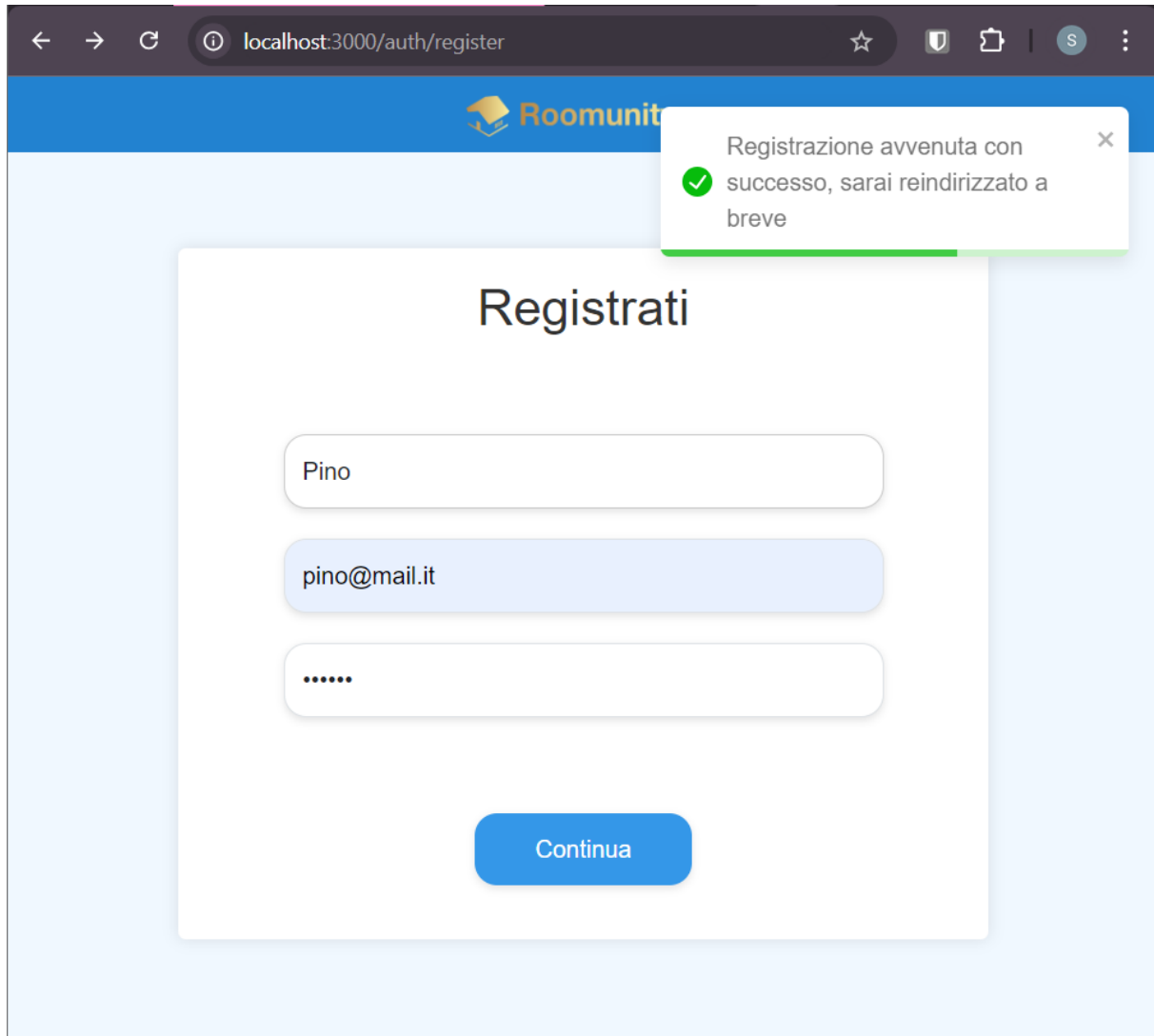
Nome prodotto

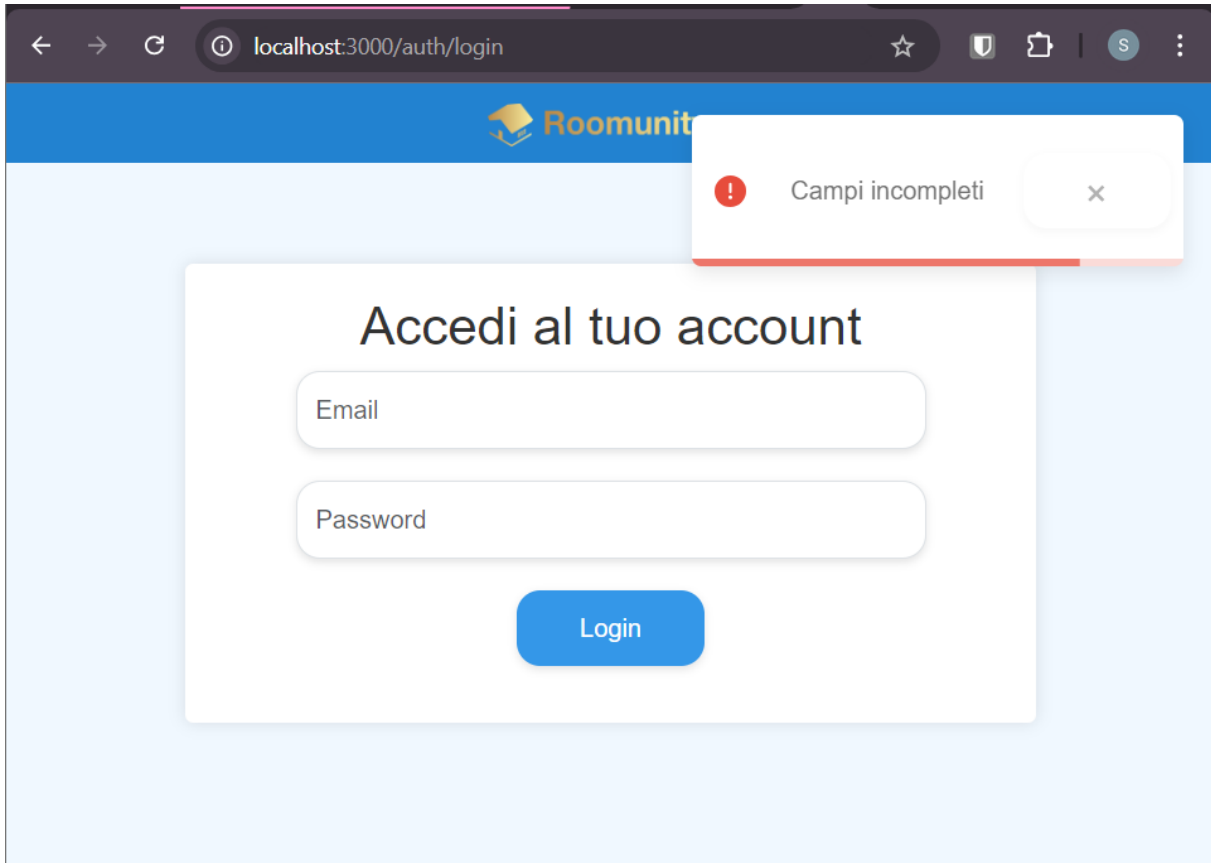
Quantità

Prezzo

[Aggiungi Prodotto](#)

Infine, vengono mostrati alcuni esempi di toast.





The screenshot shows a web browser window with the address bar displaying `localhost:3000/auth/login`. The page header features the RoomUnity logo. A central white card contains the heading "Accedi al tuo account" and two input fields labeled "Email" and "Password". Below these fields is a blue "Login" button. A red error message box in the top right corner of the card area displays a red exclamation mark icon and the text "Campi incompleti".

## 7) DEPLOYMENT

All'interno dell'organizzazione GitHub del progetto è possibile consultare i documenti nella cartella **Deliverables**, oltre all'effettivo codice sviluppato nella cartella "roomunity". Per segnare storicamente il percorso svolto dal team, si è scelto di lasciare il vecchio progetto nella repository sotto il nome di *old\_project*.  
Link al progetto:

**RoomUnity GitHub**

### 7.1) ESECUZIONE IN LOCALE

In questa parte della documentazione verranno fornite tutte le informazioni necessarie per clonare, configurare ed eseguire in locale il progetto dal repository GitHub, nonché per configurare correttamente l'ambiente di sviluppo.

#### Prerequisiti

Prima di eseguire il progetto, è necessario avere installato i seguenti strumenti:

- Git
- Node.js (versione consigliata: 14.x o successiva)
- npm o yarn come gestore di pacchetti

Passi per l'esecuzione in locale:

#### 1. Clonare la Repository

Il primo step è clonare o scaricare la repository del progetto GitHub.

#### 2. Installare le dipendenze

Il progetto, utilizzando Node.js, utilizza dipendenze gestite da npm o yarn. Dopo aver clonato il progetto bisogna installare le dipendenze eseguendo il comando all'interno della cartella del progetto.

#### 3. Impostare il file .env

Il progetto richiede (come specificato nel punto 2.3) che vengano usate delle variabili di ambiente che non sono inserite all'interno della repo github. Creare un file .env nella directory principale e inserire il seguente testo:

```
MONGODB_URI="mongodb+srv://room-unity-user:not-really-safe-  
pass@cluster0.tscu9gp.mongodb.net/?retryWrites=true&w=majority&  
appName=Cluster0"  
DB_NAME="roomunity"  
JWT_SECRET="fsdfjdsiojfisojdio"
```

#### 4. Costruire il progetto Next.js

Prima di eseguire il progetto è necessario costruire i file necessari per generare la cartella .next con i file precompilati.

#### 5. Avviare il server in modalità sviluppo o test

Eeguire:

```
npm start → per lanciare il progetto  
npm run test → per eseguire in modalità test  
npm run dev → per eseguire in modalità sviluppo
```

## 7.2) DEPLOYMENT CON VERCEL

Il deployment è stato eseguito tramite Vercel, con una connessione diretta alla repository di GitHub del progetto. Questa integrazione semplifica il processo di distribuzione, automatizzando il rilevamento delle modifiche nel repository. Vercel fornisce un URL pubblico per accedere immediatamente all'ultima versione dell'applicazione tramite questo link:

**RoomUnity**