Bookit

Applicazioni e Servizi Web

Jacopo Corina - 000929808 {jacopo.corina@studio.unibo.it}
Filippo Nardini - 000950342 {filippo.nardini@studio.unibo.it}

Settembre 2022

Indice

Capitolo 1

Introduzione

L'obiettivo del progetto consiste nella realizzazione di un'applicazione web che consenta ad un gestore di aule studio di aggiungere le proprie strutture, le aule presenti ed i servizi offerti all'interno di una piattaforma online. I fruitori della piattaforma potranno poi ricercare le aule studio, visualizzare gli orari ed i servizi offerti. Inoltre potranno effettuare la prenotazione di uno specifico posto direttamente all'interno dell'app. Infine i gestori potranno autenticare l'utente mediante un QR code generato durante la prenotazione.

Capitolo 2

Requisiti e analisi dei target users

La fase iniziale dell'elaborato ha previsto più sessioni di *brainstorming* al fine di elencare in maniera chiara e succinta tutti i requisiti funzionali e non.

2.1 Requisiti

2.1.1 Requisiti funzionali

Le sottosezioni che seguono descrivono i requisiti funzionali.

Autenticazione

- Login: l'applicativo richiederà agli utenti di effettuare l'accesso mediante i campi *email* e *password*;
- Registrazione: l'applicativo permetterà agli utenti di effettuare una registrazione utilizzando un'*email* non ancora utilizzata e una *password*. Questa fase prevederà anche che l'utente registrato scelga un tipo di utente *gestore* o un tipo di utente *fruitore*.

Gestione dei locali

- Elenco dei locali: permette al gestore di visualizzare l'elenco dei locali a propria disposizione;
- Visualizzazione di un locale: permette al gestore di visualizzare nello specifico le caratteristiche di un locale e l'elenco delle aule al suo interno
- Modifica di un locale: permette al gestore di modificare le caratteristiche di un locale;

- Rimozione di un locale: permette al gestore di rimuovere dall'elenco un locale.

Gestione delle aule

- Elenco delle aule di un locale: permette al gestore di visualizzare l'elenco delle aule di un locale;
- Visualizzazione di un'aula: permette al gestore di visualizzare le caratteristiche di un'aula;
- Modifica di un'aula: permette al gestore di modificare le caratteristiche di un'aula;
- Rimozione di un'aula: permette al gestore di rimuovere un'aula dall'elenco di un locale;
- Creazione e modifica della piantina di un'aula: permette al gestore di disegnare approssimativamente la piantina 2D di un locale, specificando per ciascun posto i servizi disponibili (per esempio Wi-Fi o presa Ethernet).

Conferma di una prenotazione

Funzionalità che permette ad un gestore di confermare una prenotazione e quindi consentire l'accesso all'aula. Per realizzare questa funzionalità sarà necessario effettuare l'uso di tecnologie come QR code.

Ricerca aule studio

- Ricerca della aula studio attraverso il nome oppure la città
- Possibilità di applicare filtri alla ricerca, quali giorni di apertura, requisito di accessibilità, presenza di determinati servizi.

Visualizzazione aula studio

L'utente può visualizzare informazioni relative ad un' aula studio, come specificate dal gestore in fase di creazione

Prenotazione di un posto

L'utente può scegliere la sua postazione, in una determinata giornata e fascia oraria, visualizzare le caratteristiche della postazione ed essere bloccato se questa non è più disponibile

Elenco prenotazioni

L'utente ha a disposizione l'elenco delle prenotazioni effettuate, in modo che il gestore possa svolgere le opportune verifiche di controllo.

2.1.2 Requisiti non funzionali

Tra i requisiti non funzionali emersi i principali sono:

- Eventuali errori devono essere nascosti all'utente finale, mediante una pagina di *graceful exit*.
- Le informazioni gestite all'interno della piattaforma devono essere erogate tenendo conto dei permessi dell'utente che le richiede. Dovranno essere quindi utilizzati dei meccanismi di autorizzazione per implementare questo comportamento.
- Utilizzo di un layout *mobile first* per la web app, ma flessibilità tale da essere utilizzato anche da dispositivi con diverse dimensioni.
- Utilizzo di interfacce semplici, per risultare il più possibile *user-friendly*.
- Le API dovranno essere implementate seguendo il modello RESTful e non dovranno esporre informazioni sensibili.

2.2 Analisi dei target users

Al fine di produrre delle interfacce semplici, chiare ed usabili in questa sezione verrà effettuata un'analisi di quelli che sono i *target user* di questo applicativo. Infatti definendo quelle che sono classi di utilizzatori finali del servizio, sarà più semplice rispettare le necessità di ciascuna di esse.

Data l'età media dei frequentatori delle aule studio e biblioteche, il gruppo principale di *target user* è stato individuato in ragazzi giovani, entro i 35 anni di età, sicuramente possessori di smartphone, che sono entrati in contatto fin da giovani con la tecnologia e le interfacce web. La loro mente è flessibile e sicuramente capace di comprendere in breve tempo l'interfaccia del servizio.

Un gruppo secondario di *target user* è stato individuato invece in frequentatori over 35, senza un limite specifico di età. Questi non hanno la stessa esperienza del gruppo precedente con la tecnologia e le sue interfacce. Se un'interfaccia non è sufficientemente chiara c'è il rischio concreto che questi si allontanino dal servizio. Sono il gruppo critico, sul quale devono essere concentrate le attenzioni. Infatti se le interfacce sono sufficientemente intuitive da essere comprese da questo gruppo, allora il gruppo principale non avrà problemi a fruirne.

Infine il gruppo di gestori è stato identificato in persone dall'età che può variare, dai 30 anni, senza un limite superiore. Dovranno essere in grado di inserire all'interno della piattaforma tutti i locali e aule che hanno in gestione, utilizzando l'editor delle piantine. Dato che questa tipologia di utente lavora solitamente in una postazione fissa, dotata di computer, le interfacce di questa sezione saranno orientate all'utilizzo desktop. In particolare l'editor, in quanto si tratta di un componente con molte funzionalità, sarà più facile da utilizzare da un dispositivo delle dimensioni almeno di un tablet.

2.2.1 Personas

Le *personas* sono finti utenti che vanno a concretizzare delle determinate caratteristiche, utili per la rifinitura dei requisiti. Ne sono state individuate tre.

Marco Rossini

Marco ha 21 anni ed è uno studente della facoltà di Lingue e Culture Moderne di Urbino. Vive a Pesaro, vicino alla stazione e frequenta l'università da pendolare, prendendo un autobus tutte le mattine alle 7:30, per poi rientrare solitamente entro le 15:00. Non avendo un vero e proprio spazio dove studiare, egli è solito, prima di rincasare, appoggiarsi per qualche ora in una piccola aula studio vicino a casa sua, in modo da poter riordinare le note appuntate durante le lezioni del giorno. L'aula si trova all'interno di una biblioteca e fornisce servizi per lui fondamentali, come Wi-Fi e postazioni con computer.

Recentemente il suo orario è cambiato e le lezioni che frequenta terminano più tardi. Ora quando rientra a Pesaro sono solitamente le 16:30 e quando raggiunge la solita aula studio la trova sempre piena. A Marco piacerebbe se ci fosse un modo per prenotare un posto nella propria fascia oraria di interesse, senza dover occupare una postazione per l'intera giornata.

Rosa Mancini

Rosa è una donna di 42 anni, professoressa di italiano e storia al Liceo Scientifico G. Marconi di Pesaro. Vive poco fuori città e raggiunge il proprio luogo di lavoro in auto. Rosa non è un'appassionata di tecnologia, anzi nel tempo ha sviluppato una vera e propria repulsione per essa; infatti non possiede un computer personale e utilizza uno smartphone obsoleto, di 5 anni fa. Invece ama i libri, la biblioteca e il suo silenzio. Per questo motivo ogni volta che può cerca di ritagliare dello spazio nel suo tempo libero per passare del tempo nell'aula studio della biblioteca accanto alla scuola, dove può correggere i compiti dei suoi studenti o dedicarsi alla lettura.

A volte, raramente, per motivi di lavoro necessita di utilizzare un computer. Non possedendone uno è solita utilizzare una postazione fornita dalla biblioteca. Capita a volte però che le postazioni con computer siano tutte occupate e che sia costretta a cercare un altro posto dove lavorare. A Rosa sarebbe molto utile una piattaforma semplice e intuitiva, dove poter riservare un posto che offre un determinato servizio.

Alberto Costa

Alberto è un signore di 58 anni, diplomato in ragioneria. Dopo aver passato anni a lavorare in uno studio commerciale, negli ultimi 8 ha lavorato come impiegato nella biblioteca "Louis Braille" di Pesaro. I suoi compiti sono: gestire i prestiti dei libri (utilizzando un sistema informatico), riordinare gli scaffali e, durante il periodo della pandemia di Covid-19, gestire gli accessi contingentati

per rispettare le normative vigenti. Alberto per questioni anagrafiche non ha la stessa confidenza con smartphone e interfacce mobile che potrebbe avere un teenager. Nonostante ciò la sua lunga esperienza lavorativa precedente e quella attuale lo hanno messo quotidianamente di fronte ad interfacce desktop per lavorare con sistemi informatici gestionali.

Da quando sono presenti le normative anti Covid-19, il numero di posti all'interno delle aule sono stati ridotti notevolmente, aumentando la quantità di chiamate che la biblioteca riceve per prenotare i posti, spesso anche quando questi sono ormai esauriti, riducendo l'efficienza di Alberto, che deve passare una grande quantità di tempo al telefono. Inoltre le prenotazioni sono gestite "a mano" mediante un sistema informativo cartaceo, sistema inefficiente e prono ad errori. La produttività di Alberto migliorerebbe se le prenotazioni fossero gestite da un sistema informatico e se le prenotazioni fossero effettuate direttamente dagli utenti, mediante una piattaforma web.

2.3 Scenari

2.3.1 Prenotazione di un posto

Obiettivo: L'utente A vuole prenotare un posto in un'aula, fornendo data e ora di interesse.

Requisiti:

- l'utente deve essere autenticato.

Passi:

1. L'utente A utilizza la barra di ricerca per cercare una libreria;
2. l'utente A seleziona una libreria tra i risultati;
3. l'utente A preme *Prenota Ora* e seleziona: data, ora, stanza e posto di interesse;
4. l'utente A preme *Conferma* e ottiene un codice di conferma della prenotazione.

2.3.2 Creazione di una stanza

Obiettivo: Il gestore B vuole aggiungere una nuova stanza X all'interno del proprio locale Y.

Requisiti:

- il gestore deve essere autenticato;
- il locale Y deve essere già inserito all'interno della piattaforma.

Passi:

1. Il gestore B utilizza la *Dashboard* per visualizzare l'elenco dei locali in sua gestione;
2. il gestore B seleziona il locale Y e preme *Aggiungi una stanza*;
3. il gestore B inserisce il nome della stanza e le sue caratteristiche;
4. infine il gestore B utilizza l'editor per modellare la piantina della stanza, inserendo i posti a sedere e i servizi per ciascuno di essi.

2.3.3 Conferma di una prenotazione

Obiettivo: Il gestore B vuole confermare la prenotazione dell'utente A, validando il suo token.

Requisiti:

- sia il gestore che l'utente devono essere autenticati;
- il locale X deve contenere al proprio interno la stanza Y;
- l'utente deve possedere un token, ottenuto al termine di una prenotazione;

Passi:

1. L'utente A apre l'elenco delle prenotazioni e seleziona quella per la stanza Y del locale X;
2. l'utente A mostra il proprio token al gestore B (utilizzando per esempio un QR code);
3. il gestore utilizza la funzione *Conferma prenotazione* e inserisce il token dell'utente A (utilizzando per esempio uno scanner QR), per confermare la prenotazione.

2.4 Focus group

Al fine di produrre delle interfacce di alta qualità sono stati organizzati dei *focus group*, ossia discussioni alla quale tra le 8 e 12 persone sono invitate a discutere e confrontarsi riguardo ad un tema o un'idea. Nello specifico, data la scarsa disponibilità di persone che si sono offerte, sono stati organizzati due *focus group* di dimensioni minori rispetto a quelle standard. Le interfacce sviluppate (che verranno mostrate nel capitolo successivo) sono state realizzate grazie alla collaborazione con questi gruppi.

2.4.1 Gruppo A

Descrizione del gruppo: Gruppo di 6 ragazzi tra i 20 e i 26 anni, studenti e non, di entrambi i sessi.

Tema: Il gruppo è stato utilizzato come rappresentante dei potenziali utenti della piattaforma. Tra i vari concetti emersi durante il confronto, emerge la necessità di avere un elenco di locali preferiti, che deve essere facilmente accessibile. Un altro requisito espresso dalla maggioranza dei partecipanti è stato quello di utilizzare un metodo rapido per lo scambio del token, come un codice QR.

2.4.2 Gruppo B

Descrizione del gruppo: Gruppo di 3 adulti tra i 38 e i 52 anni, di diversi impieghi, di entrambi i sessi.

Tema: Il gruppo, di un numero molto ridotto a causa di scarsa disponibilità di partecipanti, è stato coinvolto per quello che riguarda la parte di amministrazione della piattaforma, in particolare riguardo all'editor. Il suggerimento principale, condiviso tra tutti i partecipanti, è stato riguardante al numero di *strumenti* forniti al suo interno: meglio pochi strumenti di facile utilizzo (come selezione, rimozione, spostamento di un posto alla volta), piuttosto che molti strumenti, che potrebbero effettuare operazioni complesse (come la selezione multipla di un'area rettangolare), ma dalla funzionalità poco intuitive.

Capitolo 3

Design

Di seguito vengono mostrati alcuni mock-up che illustrano l' idea di massima delineata durante i focus group. In particolare sono stati realizzati mock-up sia per l'area utente che l'area gestore.

3.1 Autenticazione

Arrivati alla pagine iniziale (figura 3.1), è possibile cliccare sul bottone del menu per mostrare le due possibilità: login o registrazione. La schermata 3.2 di login e quella di registrazione sono analoghe: è necessario inserire username (cioè la e-mail) e la password per accedere, ed è presente un link per navigare verso l'altra pagina, nel caso si fosse sbagliato a cliccare sulla voce del menù. Una volta creato l'account sarà necessario decidere a quale profilo associarlo: "utente" o "gestore. Nelle figure 3.3 e 3.4 vengono mostrati i campi da completare per concludere la fase di registrazione.

3.2 Profilo "utente"

Per il profilo "utente", cliccando sull' icona in alto viene mostrata la barra laterale (figura 3.5). Essa include alcune informazioni del profilo, i collegamenti verso le pagine di ricerca, prenotazioni, ed elenco delle aule studio preferite. Quest' ultimo punto è stato incluso perchè è chiaramente emerso durante i focus group. La pagina principale che apparirà all' utente sarà quella di ricerca, ed una volta inserito il valore opportuno della barra di ricerca appariranno gli eventuali risultati (figura 3.6). Cliccando su uno dei risultati, appariranno più informazioni collegate ad esso, quali giorni e fasce orarie di apertura, eventuali servizi disponibili. Sempre da questa schermata, come si può evincere dalla figura 3.7, è possibile aggiungere l' aula studio nell' elenco dei preferiti cliccando sul bottone vicino all' immagine. Scegliendo di prenotare una postazione, si viene re-indirizzati ad una pagina che permette la scelta della data (figura 3.8) e successivamente ad un' altra (figura 3.9) la quale permette di scegliere la stanza

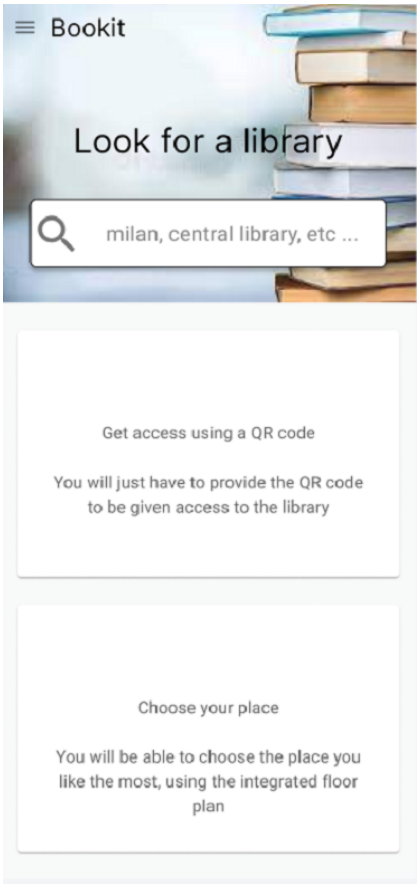


Figura 3.1: Schermata di Landing page

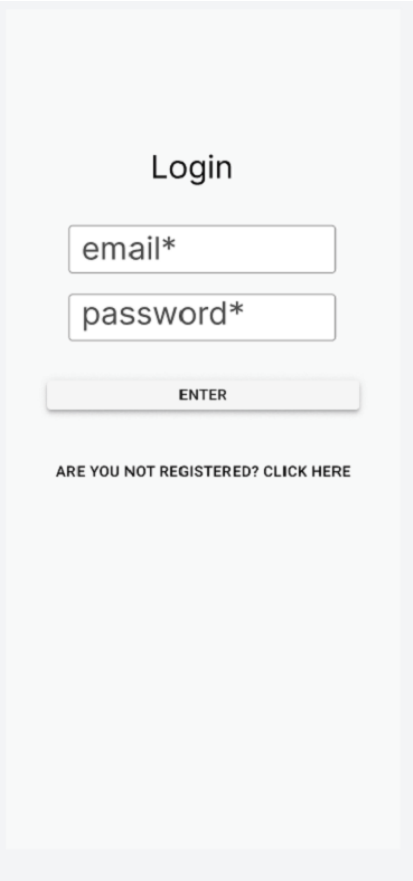


Figura 3.2: Schermata di Login

dell'aula studio, la fascia oraria e la postazione, interagendo con la mappa. L'ultima fase della prenotazione consiste nel mostrare un riassunto delle azioni compiute (figura 3.10), e dando la conferma finale viene generata una ricevuta (figura 3.11) contenente un opportuno codice (es. QR code), da presentare in futuro al gestore. Dalla schermata con la ricevuta è possibile visualizzare nuovamente la disposizione grafica della postazione prenotata (VIEW) oppure eliminarla (DELETE). Per recuperare la ricevuta è possibile utilizzare la voce contenuta nella barra laterale.

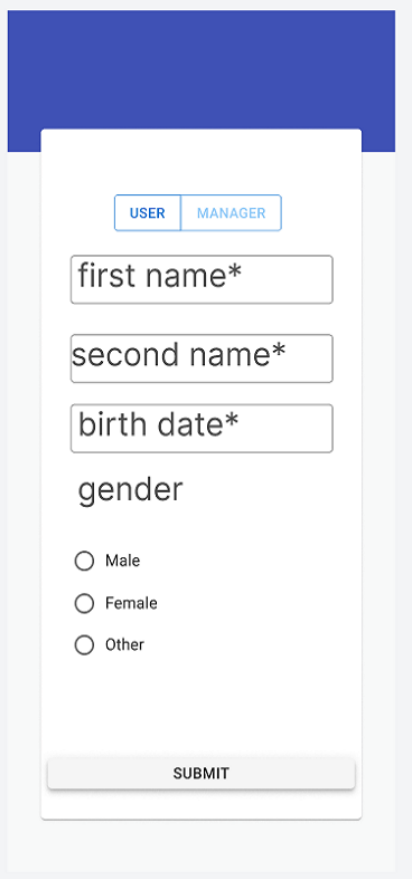


Figura 3.3: Schermata con informazioni richieste per il profilo utente

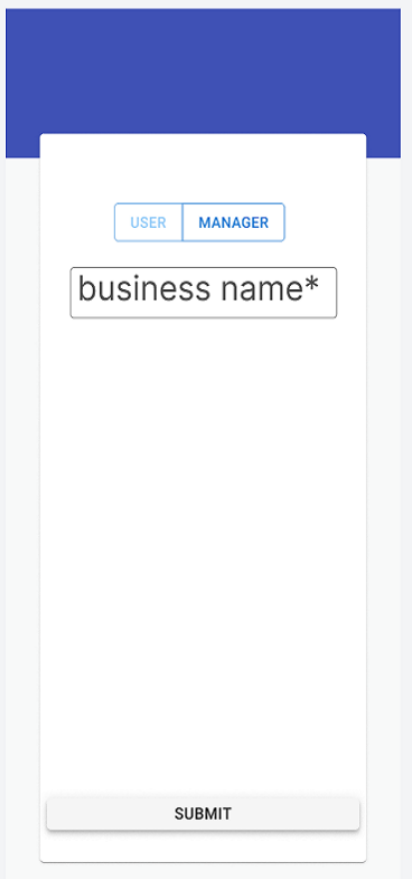


Figura 3.4: Schermata con informazioni richieste per il profilo gestore

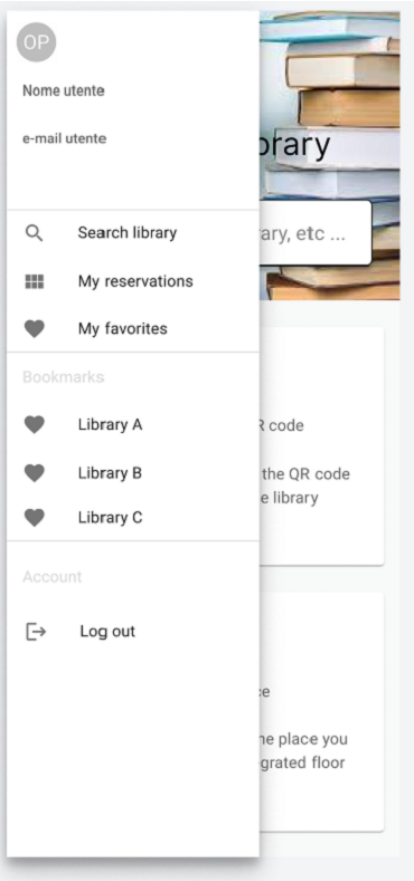


Figura 3.5: Dettaglio di barra laterale con utente loggato

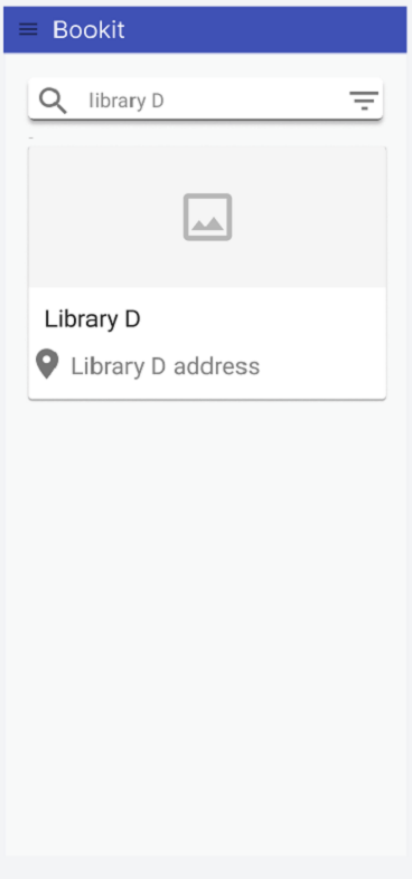


Figura 3.6: Ricerca di un' aula studio

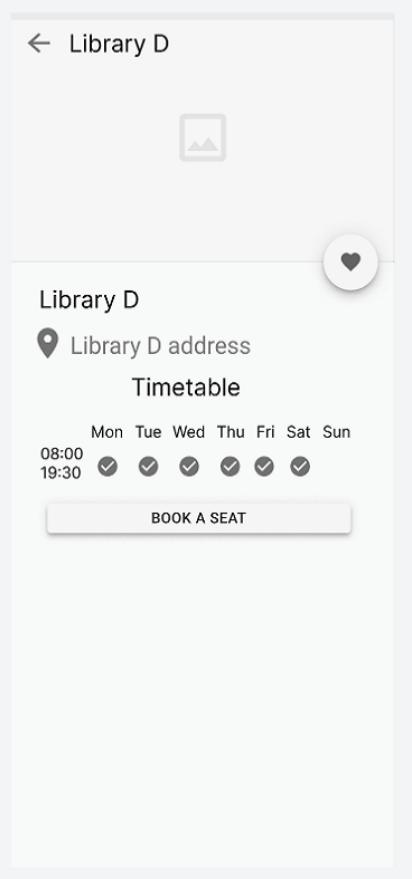


Figura 3.7: Dettagli di un'aula studio

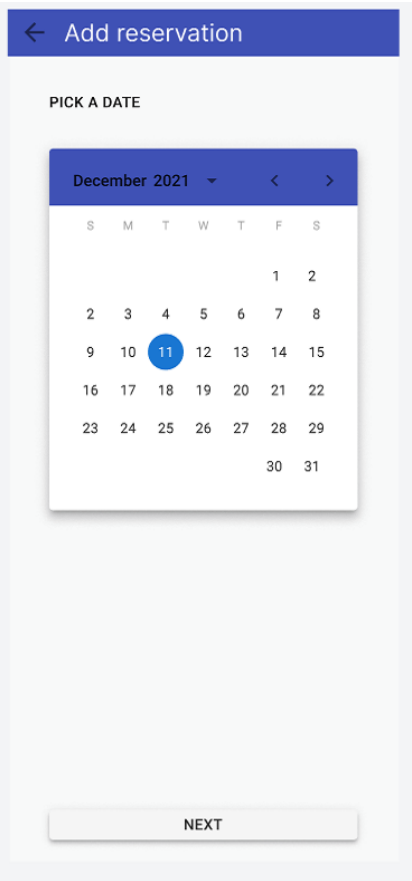


Figura 3.8: Selettore della data della prenotazione

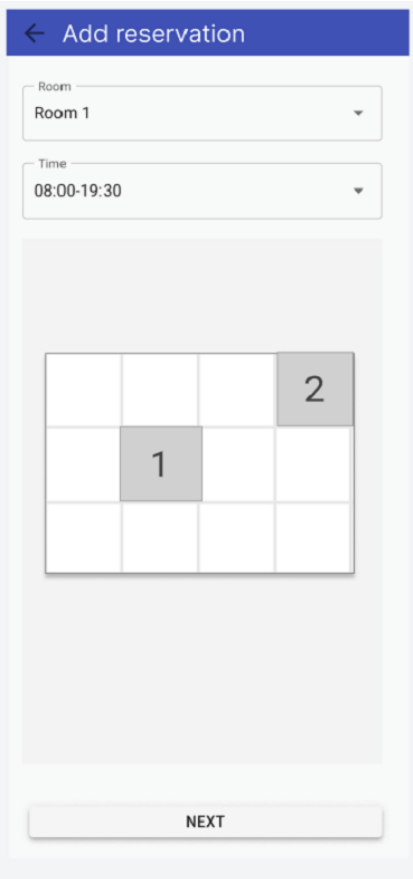


Figura 3.9: Selettori di stanza, fascia oraria e mappa con postazioni disponibili

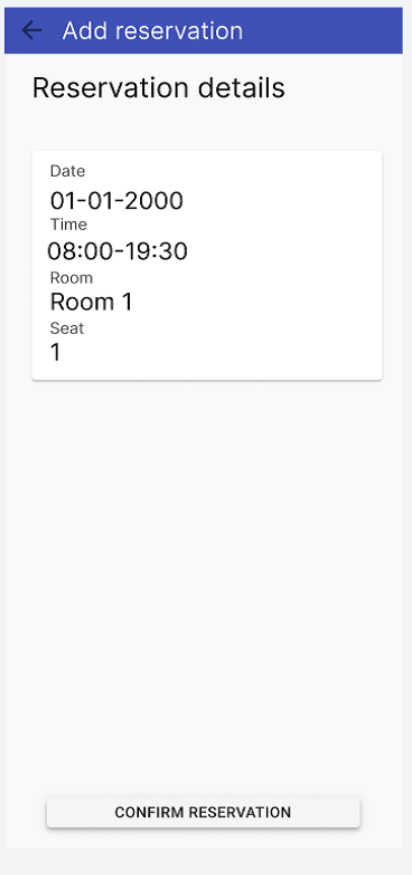


Figura 3.10: Riassunto presentato nella pagina di conferma della prenotazione

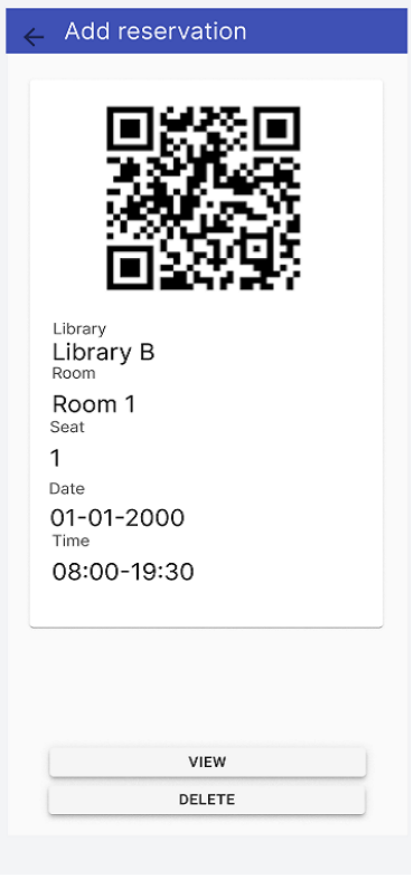


Figura 3.11: Ricevuta di conferma della prenotazione

3.3 Profilo "gestore"

Quando si effettua l'accesso con un account avente profilo "gestore", la pagina principale (figura 3.12) consiste nella "dashboard", il cruscotto che mostra le aule studio create dal gestore. Sempre nella stessa schermata è possibile avviare la procedura di controllo del codice di prenotazione oppure agire per creare una nuova aula studio. La procedura di creazione si svolge in più step: inizialmente vengono richiesti dati essenziali per identificare un' aula studio (figura 3.14), cioè nome, indirizzo e città. Proseguendo è necessario completare una tabella (figura 3.15) nella quale per ogni giorno è possibile definire una fascia oraria. Opportuni controlli dovranno essere implementati per risolvere conflitti di sovrapposizione. Infine è possibile caricare un' immagine (figura 3.16) e salvare il tutto. Queste informazioni possono essere modificate in qualsiasi momento cliccando sull' apposita icona di modifica dell' aula studio. Tornando alla dashboard, quando si clicca sull' aula studio vengono mostrate le stanze presenti in essa (figura 3.13). La creazione di una stanza consiste semplicemente nel definire il nome per essa. Infine, il gestore può utilizzare l'editor sulla stanza appena creata per definire disposizione e caratteristiche delle postazioni (figura 3.17). Ogni cella può contenere una postazione e ad ogni postazione è possibile associare un elenco di servizi predefiniti (figura 3.18).

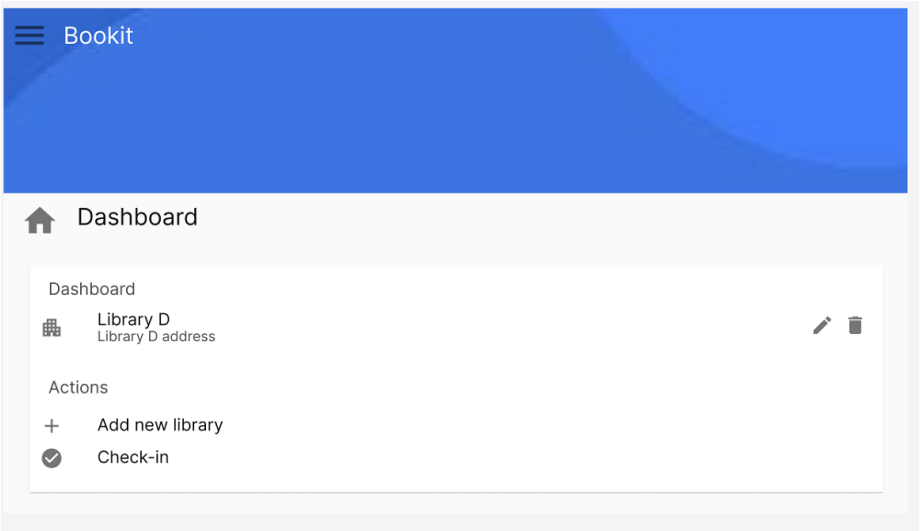


Figura 3.12: Dashboard di un gestore di aule studio

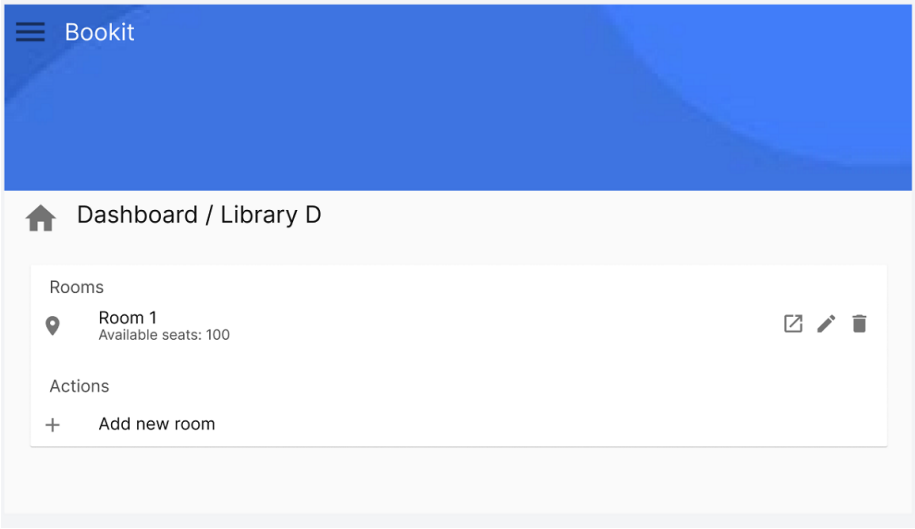


Figura 3.13: Elenco delle stanze all' interno di un' aula studio

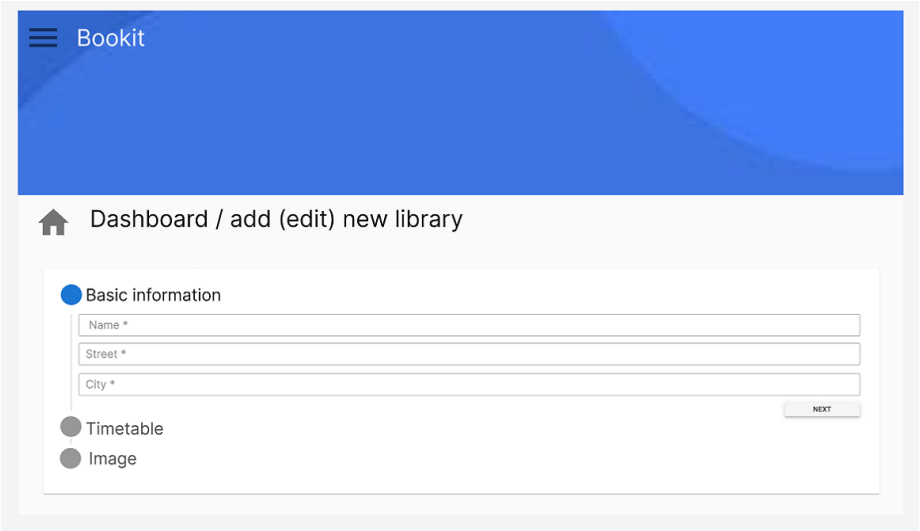


Figura 3.14: Primo step della procedura di creazione/modifica di un' aula studio

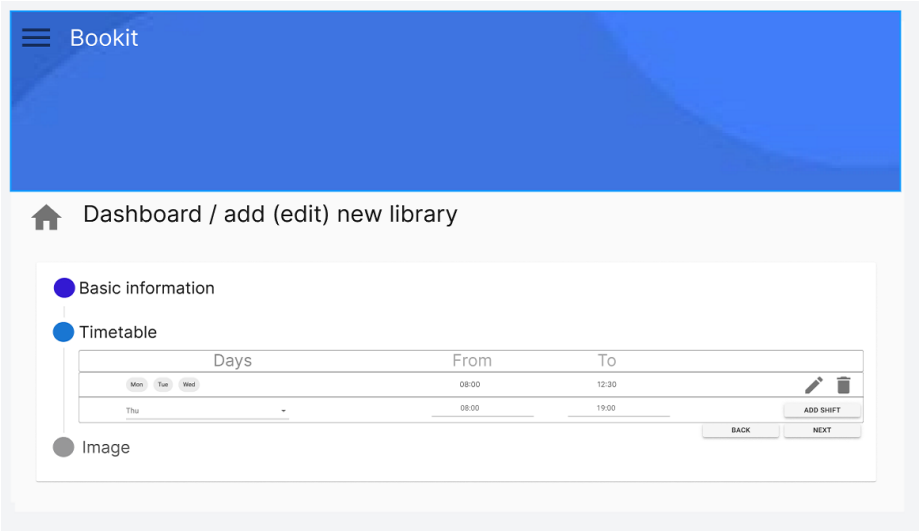


Figura 3.15: Secondo step della procedura di creazione/modifica di un' aula studio

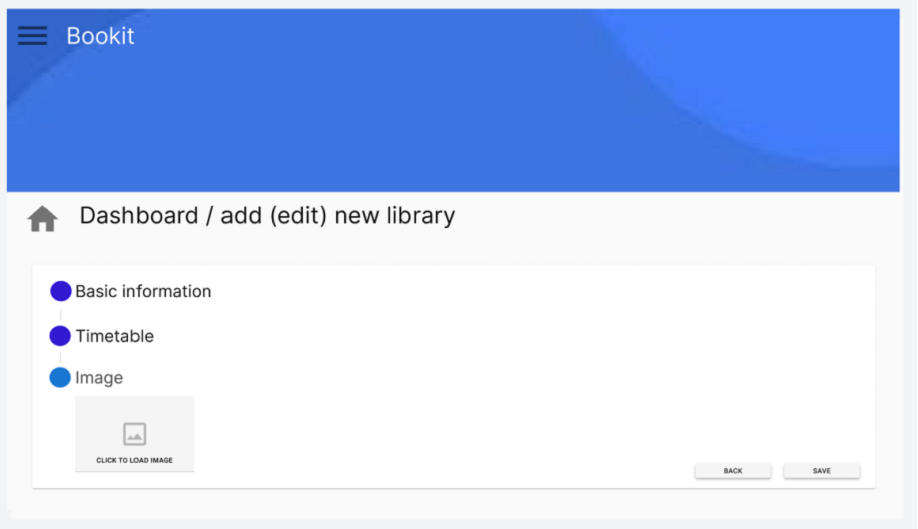


Figura 3.16: Terzo step della procedura di creazione/modifica di un' aula studio

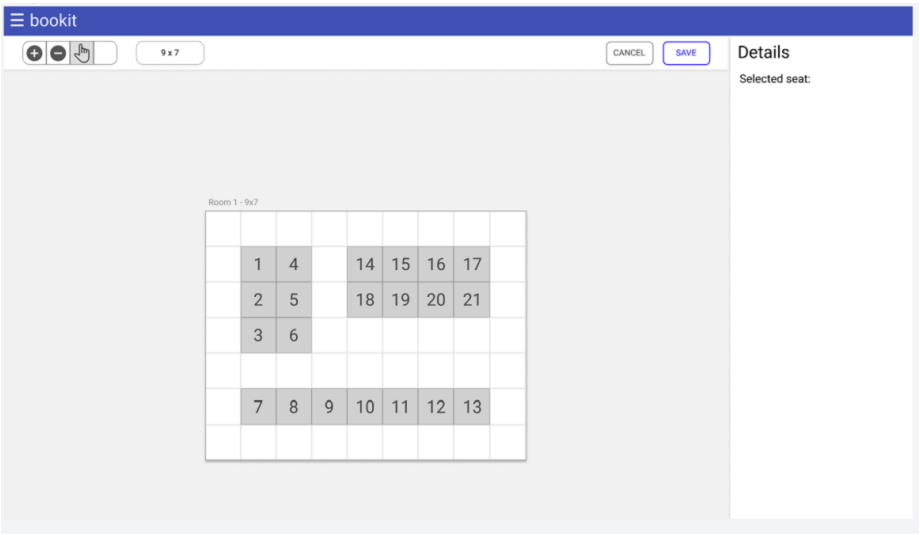


Figura 3.17: Editor della stanza

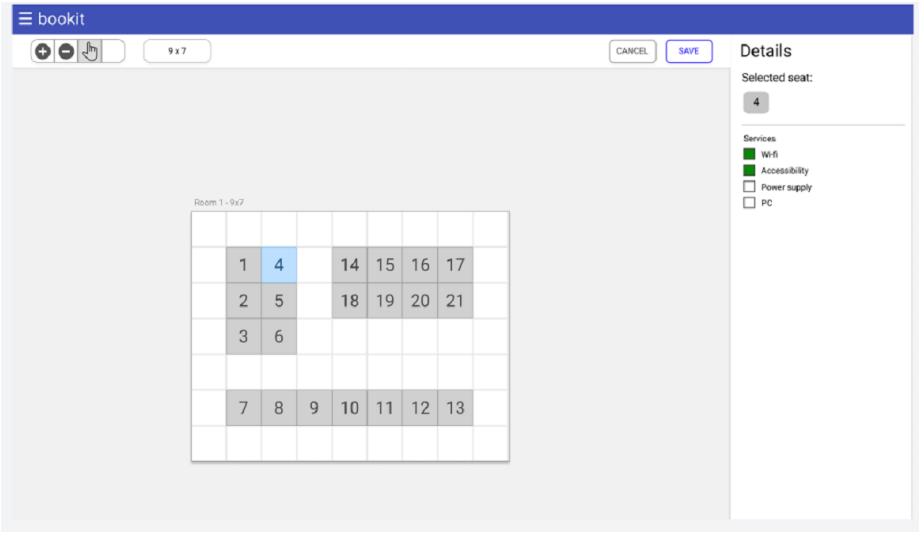


Figura 3.18: Elenco dei servizi associati ad una postazione

3.4 Modellazione del dominio

Una volta identificate quelle che sono le interfacce da implementare, è stato utilizzato il modello E/R per strutturare le istanze del dominio. Il risultato è visible nella Figura 3.19.

In particolare:

- *User*: rappresenta un'istanza di credenziali, identificato dall'*email*;

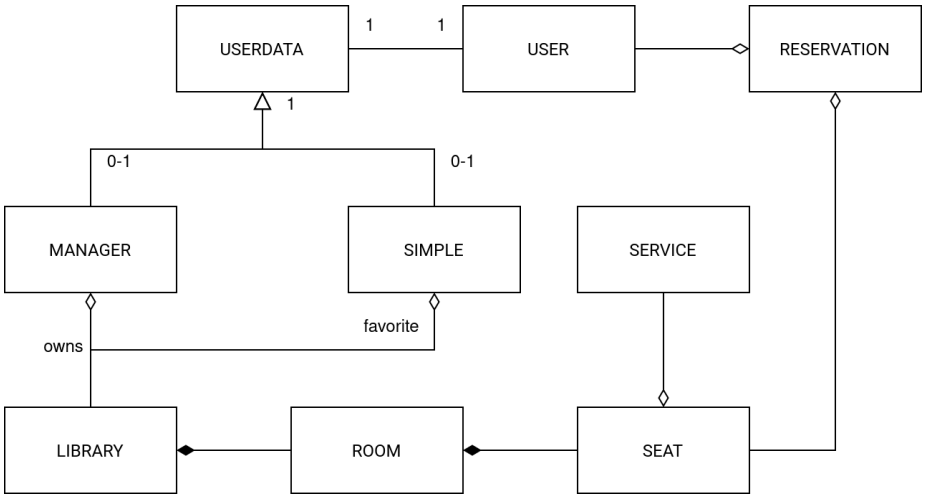


Figura 3.19: Rappresentazione tramite modello E/R delle entità principali di cui è composto il dominio.

- *UserData*: rappresenta un'istanza di dati associata ad un utente; può essere di tipo *simple* o di tipo *manager*;
- *Simple*: rappresenta la tipologia di utente consumatore del servizio, che frequenta le aule studio;
- *Manager*: rappresenta la tipologia di utente *gestore* di una o più aule studio;
- *Library*: rappresenta un locale (aula studio), modella caratteristiche come nome, indirizzo, città, data e orario di apertura;
- *Room*: rappresenta una stanza, contiene più *seat*;
- *Seat*: rappresenta un posto a sedere;
- *Service*: rappresenta i possibili servizi offerti da un posto a sedere;
- *Reservation*: rappresenta una prenotazione, che associa un utente ad un posto a sedere, in una specifica data e orario.

Capitolo 4

Architettura

In questo capitolo verrà descritta l'architettura del sistema che verrà realizzato. In particolare verranno elencate e motivate quelle che sono le scelte principali che avranno un impatto sulla realizzazione dell'elaborato.

4.1 Architettura del sistema

Il sistema che verrà implementato prevede la realizzazione di due componenti software:

- Sistema *front-end*: Single-page web application servita tramite un hosting statico, che ospiterà le interfacce di interazione degli utenti con il sistema. Questa comunicherà con il Sistema *back-end* attraverso delle API HTTP.
- Sistema *back-end*: Applicazione back-end, che espone attraverso un'interfaccia HTTP l'insieme di operazioni necessarie all'esecuzione del sistema.

Architettura del back-end

L'architettura del sistema di *back-end* realizzata è ispirata a quella offerta da framework popolari come *NestJS*. Essa prevede la separazione della logica applicativa, da quella del modello, da quella delle modalità di erogazione del servizio (REST, SOAP, RPC, GraphQL, ecc.).

I livelli di separazione logica implementati sono:

- *Model*: Rappresenta l'integrazione tra un sistema ad oggetti come JavaScript e un database, relazionale e non. Permette di svolgere semplici operazioni CRUD, senza la necessità di utilizzare specifici *Query Language*.
- *Service*: Rappresenta una porzione della logica di business dell'applicazione. Utilizza e opzionalmente combina istanze specifiche dei modelli per implementare operazioni non banali, come per esempio l'autenticazione di un utente.

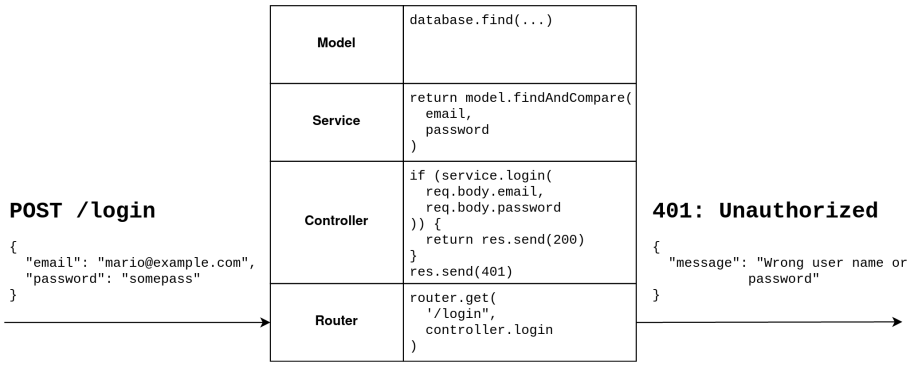


Figura 4.1: Esempio di chiamata HTTP effettuata verso il servizio e dei livelli di astrazione utilizzati per esaudire la richiesta.

- *Controller*: Permette di collegare 1:1 operazioni di un servizio alla logica HTTP. Specifica quali servizi e quali operazioni devono essere chiamati, e come deve avvenire l'eventuale passaggio di parametri.
- *Router*: Permette di collegare la logica dei Controller ad una specifica rotta HTTP. Solitamente i Controller vengono raggruppati all'interno dei Router a seconda delle aree del dominio di cui si occupano.

Quando viene effettuata una richiesta, questa attraversa tutti gli stack di astrazione, per poi tornare indietro e generare una risposta.

La suddivisione del software di *back-end* in questi livelli garantisce numerosi vantaggi, tra i quali:

- maggiore modularità, quindi riusabilità del software prodotto;
- facilità di scrittura di unit test;
- possibilità di usare la stessa logica di business per fornire il servizio mediante un'altra interfaccia (per esempio GraphQL).

All'interno del capitolo 8 verrà mostrato come effettuare un deployment del sistema prodotto insieme ai servizi richiesti.

4.2 Architettura del repository

Dato che sia la componente *front-end* che quella *back-end* dovranno essere realizzate utilizzando il linguaggio Typescript, le due componenti sono state organizzate all'interno di un *monorepo*, ovvero la tecnica di includere due o più applicazioni, anche indipendenti, all'interno dello stesso repository. In questo modo è possibile condividere agevolmente del codice, come per esempio la definizione dei tipi delle risorse. Inoltre in questo modo la configurazione degli

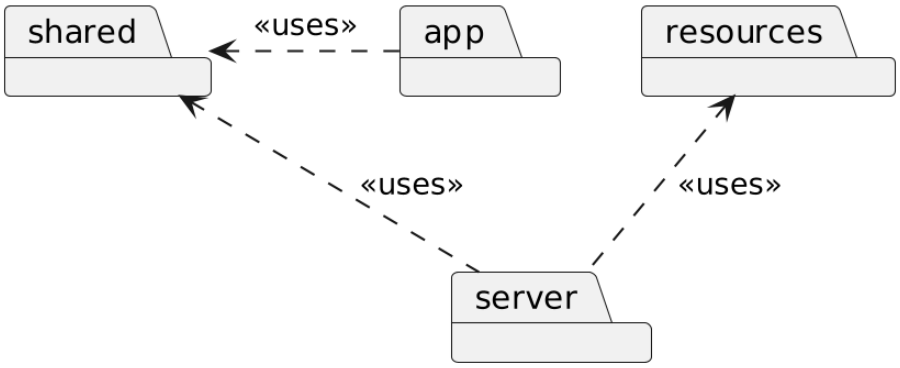


Figura 4.2: Diagramma UML dei package, che rappresenta la suddivisone in package del sistema prodotto e le dipendenze tra di essi.

strumenti di controllo di qualità del codice può essere dichiarata in maniera centralizzata.

Ciascun componente software all'interno del monorepo viene chiamato *package* e viene posto all'interno della cartella *packages*. I package totali risultanti sono quattro (Figura 4.2):

- *shared*: contiene la definizione di tipi e strutture dati comuni tra *back-end* e *front-end*;
- *app*: che contiene il codice dell'applicazione di *front-end*;
- *resources*: una piccola API per la dichiarazione di una risorsa e delle sue rotte all'interno di un router di Express;
- *server*: che contiene il codice del server *back-end*;

4.2.1 Uso di Lerna

L'uso della strategia monorepo è stata agevolata dall'utilizzo di *lerna*, un sistema di build, che permette di gestire la versione e il rilascio di applicazioni dichiarate all'interno di questa tipologia di repository. In particolare è stato utilizzato per l'installazione della stessa versione di una dipendenza in più package, il *link* delle dipendenze tra package e la *build* centralizzata.

4.2.2 Configurazioni in comune

Tramite la strategia del monorepo sono state create delle configurazioni di base, condivise tra tutti i package, dei seguenti strumenti:

- **tsc**: compilatore TypeScript, che permette di generare codice JavaScript da eseguire su diverse piattaforme (in questo caso Node.js e Browser);

- **ESLint**: linter per codice JavaScript e TypeScript;
- **Prettier**: formatter per codice JavaScript e TypeScript;
- **Jest**: piattaforma di esecuzione di test, che tramite l'uso dell'adattatore *ts-jest*, può essere eseguita su *ts-node*, rendendo possibile la scrittura dei test in TypeScript.

Inoltre ciascun package può estendere la configurazione di base (Figura 4.3) e aggiungere regole specifiche, per migliorare il funzionamento dello strumento al proprio interno.

Un esempio di uso di questo tipo di configurazione è visibile nei file *tsconfig.base.json* e *packages/server/tsconfig.json*. Il primo è utilizzato per fornire una configurazione di base, comune a tutti i package; il secondo invece estende il primo (mediante l'istruzione *extends* della configurazione di TypeScript), andando ad aggiungere ulteriori opzioni specifiche per il proprio package.

```
// tsconfig.base.json
{
  "compilerOptions": {
    ...
  },
  "exclude": [
    ".eslintrc.js",
    ".prettierrc.js"
  ]
}
```

```
// packages/server/tsconfig.base.json
{
  "extends": "../../tsconfig.base.json",
  "compilerOptions": {
    ...
  },
  "include": ["./src/**/*"],
  "references": [
    ...
  ]
}
```

4.2.3 Generazione di codice

Durante le fasi iniziali del progetto ci si è resi conto che per una stessa risorsa si arrivavano ad avere fino a tre dichiarazioni:

- modello Mongoose;
- `type` Typescript;
- validator Joi[1].

[1] https://joi.dev/

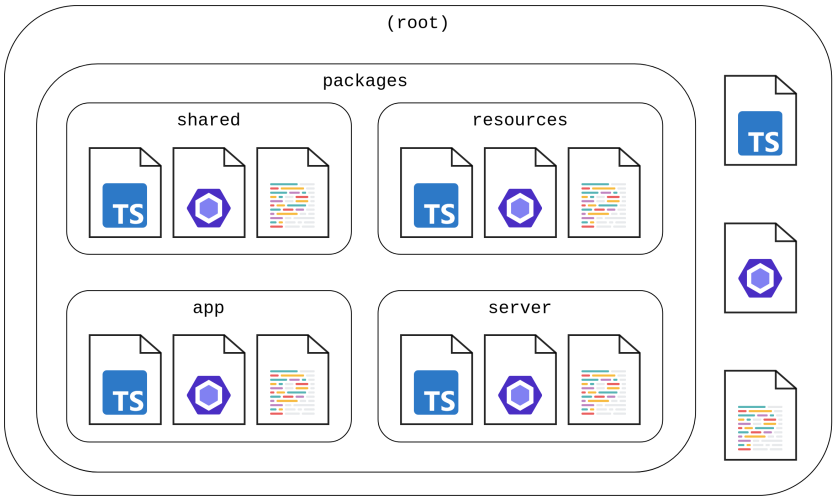


Figura 4.3: Organizzazione delle configurazioni degli strumenti di sviluppo.

Questa tripla dichiarazione è subito risultata essere un problema, poiché è sarebbe stato improbabile pensare di mantenere il codice di tre configurazioni della stessa risorsa senza incorrere in incongruenze o inconsistenze. La situazione ideale sarebbe stata quella di mantenere solamente una dichiarazione, dalla quale poter derivare tutte le informazioni necessarie per la generazione delle altre configurazioni. Tra le varie possibili soluzioni esplorate, la migliore individuata è stata la seguente. È stato utilizzato come modello base quello di Joi, che specifica la definizione della risorsa e tutte le informazioni necessarie agli altri modelli. Inoltre Joi fornisce il supporto alla compilazione di metadati, utili per memorizzare informazioni non direttamente modellate dalla sua configurazione. In seguito sono state utilizzate due librerie per generare rispettivamente il `type` Typescript e il modello Mongoose.

Typescript

La generazione del `type` ha comportato l'uso di un generatore di codice Typescript, chiamato *joi-to-typescript* [2]. È stato quindi scritto uno script, da lanciare prima della compilazione dei file Typescript, che legge tutti i modelli Joi e che genera opportunamente i `type`.

Per esempio utilizzando il seguente schema Joi:

```
1 // packages/shared/src/data/User.ts
2 export const UserSchema = Joi.object({
3   email: Joi.string().required(),
4   password: Joi.string().required()
5 }).meta({ className: 'User' });
```

[2] https://github.com/mrjono1/joi-to-typescript

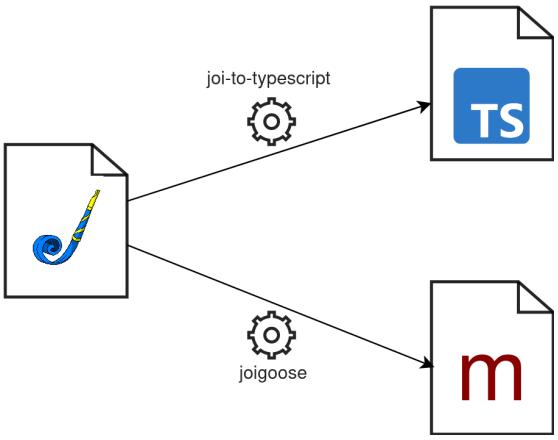


Figura 4.4: Rappresentazione del workflow per la generazione dei modelli realizzata all'interno di questo progetto. A partire da uno schema Joi, viene derivato il tipo TypeScript e lo schema Mongoose.

Lanciando lo script di generazione di codice, verrà generato il seguente file.

```
1 // packages/shared/src/generatedTypes/User.ts
2 export interface User {
3   email: string;
4   password: string;
5 }
```

Mongoose

Per quanto riguarda la creazione del modello Mongoose, questa è resa possibile dalla libreria joigoose [3]. Questa libreria permette di trasformare uno schema Joi in uno Mongoose, importandone anche le regole di validazione.

Un esempio di generazione del modello Mongoose è il seguente:

```
1  // Definizione di uno schema Joi
2  export const UserSchema = Joi.object({
3    email: Joi.string().required(),
4    password: Joi.string().required()
5    bestFriend: Joi.string().meta({
6      _mongoose: { type: "ObjectId", ref: "User" },
7    }),
8  });
9
10 // Conversione dello schema in uno Mongoose
11 const mongooseUserSchema = new Mongoose.Schema(
12   Joigoose.convert(userSchema)
13 );
14 // Creazione del modello Mongoose a partire dallo schema
15 export const User = Mongoose.model("User", mongooseUserSchema);
```

[3] https://github.com/yoitsro/joigoose

4.3 Architettura del frontend

L'architettura con la quale è stata strutturata l'applicazione *front-end* è chiamata Bulletproof React[4]. È un'architettura semplice e facilmente scalabile e sopperisce alla mancanza di un'architettura "standard" di React.

Una spiegazione semplificata della struttura è la seguente:

- *components*: componenti condivisi all'interno di tutta l'applicazione;
- *config*: configurazione condivisa;
- *features*: moduli suddivisi per feature;
- *hooks*: React hook condivisi per tutti i moduli;
- *routes*: configurazione delle rotte;
- *stores*: store per lo stato globale;

All'interno della cartella *features* sono presenti un elenco di moduli suddivisi per dominio applicativo, al cui interno è presente nuovamente la stessa struttura (senza la cartella *features*, per non avere moduli innestati a più livelli). Tramite questa struttura è possibile aggiungere nuove funzionalità e componenti senza avere una struttura orizzontale dei componenti, che con lo scalare della dimensione dell'applicazione diventa poco navigabile.

[4] https://github.com/alan2207/bulletproof-react

Capitolo 5

Tecnologie

Lo sviluppo dell'intero sistema è basato sullo stack MERN (MongoDB, Express, React e Node.js). Il vantaggio principale ottenuto da questa scelta è quello di utilizzare lo stesso linguaggio (JavaScript/TypeScript) sia sul *front-end* che sul *back-end*. Questa caratteristica è stata sfruttata al massimo, creando un *monorepo*, con un package comune contenente le dichiarazioni dei tipi. In questo capitolo verranno descritte le principali scelte tecnologiche effettuate.

5.1 Tecnologie back-end

S3

S3 è un servizio di memorizzazione offerto da Amazon. Poichè la soluzione ufficiale esiste solo in cloud, si è provveduto ad utilizzare altri storage locali ma basati sulle medesime API.

MinIO

MinIO[1] è un object storage, compatibile con le API di S3, e disponibile come immagine per container Docker. Esso ha permesso di memorizzare le immagini utilizzate all' interno dell' applicazione.

Multer-S3

Multer-S3 è un middleware per Node.js, che permette di caricare file attraverso l' uso di form. Questa variante di Multer è in grado di comunicare con S3 invece del file storage locale.

[1] https://github.com/minio/minio

Redis

Redis è uno storage chiave-valore, utilizzato per memorizzare i dati di sessione dell' utente loggato

Purify-ts

Purify è una libreria per scrivere codice funzionale in TypeScript, fornendo dei pattern e strutture dati comuni in questo tipo di linguaggi. A differenza di alternative come *fp-ts* [2], la sua API è molto ergonomica, accessibile tramite *dot notation*, risultando piacevole e facilmente esplorabile durante lo sviluppo.

La struttura dati utilizzata principalmente durante lo sviluppo è stata `EitherAsync`, un dato che modella un'operazione asincrona, che potrebbe fallire. Una sua rappresentazione semplificata potrebbe essere:

```
type Either<L, R> =
  | { type: 'Left', value: L }
  | { type: 'Right', value: R };
type EitherAsync<L, R> = () => Promise<Either<L, R>>
}
```

Tramite un metodo *factory* è possibile effettuare il *wrapping* di una *Promise*, in modo tale da tenere traccia anche dei possibili valori in caso di insuccesso.

Questa struttura dati implementa l'operazione *chain* (chiamata *flatMap* in altri linguaggi come Scala), che permette di concatenare operazioni successive, componendo tra di loro i possibili tipi di fallimento. In questo modo è possibile ottenere caratteristiche fondamentali della programmazione funzionale come *componibilità* e *trasparenza referenziale*, semplificando la composizione di funzioni esistenti, il refactoring.

Questo approccio è stato decisivo per la scrittura del codice di back-end, prevenendo molti eventuali *bug* che si sarebbero potuti verificare in caso di eccezioni inaspettate e non gestite.

5.2 Tecnologie front-end

React

React è una libreria per la scrittura di user interface in maniera dichiarativa, che prende spunto dal paradigma della *programmazione reattiva*. Infatti il flusso dei dati è mono direzionale e ad ogni cambiamento di stato, la UI viene ricomputata e in caso di cambiamenti il DOM viene aggiornato. Grazie a questa sua caratteristica i componenti React possono essere facilmente composti tra di loro, in modo tale da creare interfacce più complesse.

Material UI

Material UI è una libreria di componenti grafici realizzati per React, che implementa le linee *Material Design*, sviluppate da Google. Fornisce una grandissima

[2] https://gcanti.github.io/fp-ts/

varietà di componenti production-ready e un sistema di *theming*, per dichiarare personalizzazioni dello stile in una configurazione centrale, che sarà propagata all'interno di tutta l'applicazione.

Zustand

Zustand è una libreria per la gestione dello stato. Tramite un binding, è possibile integrarla con React, in modo da creare uno *storage* globale e non dipendente da un componenente React. Questa libreria è stata confrontata con la più famosa *redux* e, nonostante la seconda sia più adatta a gestire stati complessi, questa prevede la scrittura di molto codice *boilerplate*. Alla luce di queste considerazioni la scelta è ricaduta su Zustand, che permette di dichiarare uno stato e delle operazioni in poche righe (esempio nel Listing 5.1).

```
1 const useBearStore = create((set) => ({
2   bears: 0,
3   increasePopulation: () =>
4     set((state) => ({ bears: state.bears + 1 })),
5   removeAllBears: () => set({ bears: 0 }),
6 }));
```

Listing 5.1: Esempio di dichiarazione di uno store Zustand, preso dal sito ufficiale.

React Query

React Query è un gestore di stato asincrono per React. Il suo compito è quello di effettuare richieste, caching e aggiornare i dati all'interno dell'applicazione. Seguendo la filosofia di React anche questa libreria utilizza un approccio dichiarativo, dovendo solamente specificare le modalità con le quali i dati devono essere richiesti. La sua API è molto chiara e permette di lavorare con Promise (o funzioni *async*) molto agevolmente. Inoltre supporta uno strumento di ispezione dati, molto utile durante lo sviluppo dell'applicazione, con il quale è possibile ispezionare il valore dello stato, azionare richieste manualmente o annullare richieste in corso.

React Hook Form

React Hook Form è una libreria per la gestione dei form in React. Anche questa libreria utilizza un approccio dichiarativo. È stata confrontata con alternative molto popolari come *Formik*. La differenza tra queste due librerie è nell'approccio alla gestione del form: mentre in *Formik* si utilizzano dei *controlled component*, nei quali ogni cambiamento di un input viene intercettato da un evento e lo stato aggiornato, con *React Hook Form* lo stato degli input viene catturato solamente quando il form viene inviato, permettendo di ottenere più responsiveness utilizzando form più complessi. In caso di necessità è comunque possibile implementare dei *controlled component*.

Tra le varie funzionalità spicca quella di poter validare il form utilizzando uno schema Joi, utile per non dover scrivere nuovamente uno schema per le risorse modellate.

React Draggable e React Resizable

React Draggable e React Resizable sono due componenti React che aggiungono ad un componente gli handler *onDrag* e *onResize*, utili per creare delle UI particolari. Questi componenti sono stati fondamentali per realizzare l'editor 2D.

Capitolo 6

Codice

In questo capitolo verranno mostrate alcune porzioni rilevanti di codice, suddivise per sistema.

6.1 Front-end

6.1.1 Hook useNotification

Dalla versione 16.8 di React sono stati introdotti gli *hook*, un modo semplice e dichiarativo per aggiungere funzionalità o stato ad un componente, senza dover scrivere un *class component*.

Use notification è un *hook* custom realizzato per gestire le notifiche all'interno dell'applicazione. Permette di inviare notifiche tramite un alert in un angolo dello schermo. Queste notifiche possono avere diversi livelli di severità e hanno una durata prestabilita.

```
1  // useNotifications.ts
2  const notificationState = ( set, get ) => ({
3    notifications: [],
4    pushNotification: (latest) =>
5      set({
6        notifications: [...get().notifications, latest],
7      }),
8    dismissNotification: () =>
9      set({
10       notifications: get().notifications.slice(1),
11     }),
12 });
13
14 // components/Notifications.ts
15 function Notifications() {
16   const [open, setOpen] = useState(false);
17   const [messageInfo, setMessageInfo] = useState<{
18     message: string;
19     severity: Color;
20     timestamp: number;
```

```
  }>();

  const notifications = useNotification((s) => s.notifications);
  const dismissNotification = useNotification((s) => s.
      dismissNotification);

  useEffect(() => {
    if (notifications.length && !messageInfo) {
      const { message, severity } = notifications[0];
      setMessageInfo({
        message,
        severity,
        timestamp: new Date().getTime()
      });
      dismissNotification();
      setOpen(true);
    } else if (notifications.length && messageInfo !== undefined &&
        open) {
      setOpen(false);
    }
  }, [notifications, open, messageInfo]);

  return (
    <Snackbar open={open} autoHideDuration={6000}>
      <Alert elevation={6} severity={messageInfo?.severity}>
        {messageInfo?.message}
      </Alert>
    </Snackbar>
  );
}
```

Un esempio di utilizzo di questa API è il seguente:

```
export const SomeForm = () => {
  const { pushNotification } = useNotification();
  return (
    <form onSubmit={() => {
      someAsyncCall()
        .then((res) => {
          pushNotification({
            severity: 'info',
            message: res
          })
        })
        .catch((err) => {
          pushNotification({
            severity: 'error',
            message: err.message,
          });
        });
    }}>
      ...
    </form>
  );
}
```

6.1.2 Uso di React Hook Form

React Hook Form permette di creare di form con una API dichiarativa, utilizzando degli *uncontrolled components*, ovvero componenti che non generano un evento ad ogni modifica, ma solamente quando il form viene sottomesso. Questo permette di avere dei form più responsive, specialmente con l'aumentare del numero di input.

```
interface FormValue {
  email: string;
  password: string;
}

export const Login = () => {
  const {
    register,
    handleSubmit,
    formState: { errors, isSubmitting },
  } = useForm<FormValue>();

  return (
    <form
      onSubmit={handleSubmit(({email, password}) => {
        attemptLogin(email, password)
      })}
    >
      {errors.map((e) => (
        <Alert key={e} body={e} />
      ))}
      <TextField
        id="email"
        label="Email Address"
        autoComplete="email"
        autoFocus
        {...register('email')}
      />
      <TextField
        id="password"
        label="Password"
        type="password"
        autoComplete="password"
        {...register('password')}
      />
      <Button isLoading={isSubmitting}>Log in</Button>
    </form>
  );
}
```

6.2 Back-end

6.2.1 Caricamento di un' immagine con Multer-S3

Di seguito vengono riportati alcuni pezzi di codice che mostrano come viene caricata l'immagine dal form utente verso lo storage S3. Si parte dalla definizione

dell'indirizzo al quale si può contattare lo storage

```
// s3.ts
export const s3 = new S3({
  accessKeyId: S3_ACCESS_KEY_ID,
  secretAccessKey: S3_SECRET_ACCESS_KEY,
  endpoint: `http://${S3_HOST}:${S3_PORT}`,
  s3ForcePathStyle: true,
  signatureVersion: 'v4',
});
```

Si istanzia un oggetto nel quale vengono specificati l'indirizzo dello storage, il nome del bucket, il tipo del contenuto che si intende trasmettere (in questo caso saranno solo immagini, quindi con una estensione certa), il metodo per operare sul file ricevuto e completare la scrittura (in questo caso si estrae l'estensione del file, tipicamente .jpg, e si genera un nuovo nome univoco seguito da tale estensione)

```
// multer.ts
export const libraryImagesBucketName = 'library-images';

export const libraryImageUpload = multer({
  storage: multers3({
    s3,
    bucket: libraryImagesBucketName,
    contentType: multers3.AUTO_CONTENT_TYPE,
    cacheControl: 'max-age=31536000',
    key(_: any, file: any, cb: (error: any, bucket?: string) =>
        void) {
      const extension = file.originalname.split('.').slice(-1);
      cb(null, `${uuidv4()}.${extension}`);
    },
  }),
});
```

L' utente trasmette l'immagine attraverso un data form che poi sarà letto dall' oggetto multer definito in precedenza e quindi processato

```
//library.ts (dashboard API)
async function updateLibraryImage(
  imageFile: File,
): Promise<FormDataImageResult> {
  const formData = new FormData();
  formData.append('imageFile', imageFile);

  return ky
    .post('libraries/libraryImage', {
      body: formData,
    })
    .json();
}
//library.ts (router)
router.post(
  '/libraryImage',
  libraryImageUpload.single('imageFile'),
  (req: any, res: any) => {
    if (!req.file) {
      return res.sendStatus(400);
```

```
    }

    return res.json({ key: req.file.key });
  },
);
```

6.3 Resources

Il package **resources** contiene un'API object-oriented, sviluppata per agevolare la creazione di risorse all'interno del *back-end*. Fornisce delle interfacce e mix-in per dichiarare tutte le funzionalità offerte da un servizio. Si basa sulla struttura dati *EitherAsync*, offerta dalla libreria *Purify-ts*, per esprimere in maniera precisa sia il tipo di ritorno, che il tipo di errore. Inoltre fornisce un modo per aggiungere le rotte ad un *router* Express.

La classe *core* è `AbstractService`, che al proprio interno contiene un modello Mongoose, sopra al quale possono essere aggiunte funzionalità opzionali (mediante mix-in).

```
export abstract class AbstractService<T> {
  constructor(protected model: Model<T & Document>) {}

  getModelName() {
    return this.model.modelName;
  }
}
```

L'aggiunta dei mix-in ad una classe esistente è possibile tramite la seguente funzione, importata direttamente dalla documentazione ufficiale di TypeScript[1], che permette di aggiungere funzionalità ad un costruttore esistente.

```
function applyMixins(derivedCtor: any, constructors: any[]) {
  constructors.forEach((baseCtor) => {
    Object.getOwnPropertyNames(baseCtor.prototype)
      .forEach((name) => {
        Object.defineProperty(
          derivedCtor.prototype,
          name,
          Object.getOwnPropertyDescriptor(baseCtor.prototype, name)
              ||
            Object.create(null)
        );
      });
  });
}
```

Di seguito sono mostrati due esempi di funzionalità aggiuntive, che estendono `AbstractService`, rispettivamente *findAll* e *findById*.

```
export class FindAll<T> extends BaseService<T> {
  findAll(
    filterQuery: FilterQuery<T & Document<any, any, any>>,
    projection?: any | null,
```

[1] https://www.typescriptlang.org/docs/handbook/mixins.html

```
    options?: QueryOptions,
  ): EitherAsync<UnexpectedError, WithId<T>[]> {
    return EitherAsync(() => this.model.find(filterQuery,
        projection, options))
      .map((res) => res as any)
      .mapLeft(unexpectedError);
  }
}

export class FindById<T> extends BaseService<T> {
  _definedOrNotFound<T>(
    document: T | null,
  ): EitherAsync<ExpectedError<NotFoundKind>, T> {
    return EitherAsync.liftEither(
      document === null
        ? Left({
            kind: Kinds.NotFound,
          })
        : Right(document),
    );
  }

  findById(
    id: any,
    options?: ProtectedFindByIdOptions,
  ): EitherAsync<Error<FindByIdError>, T & Document> {
    return EitherAsync(() => this.model.findById(id))
      .mapLeft((err: any) => {
        if (err.name === 'CastError') {
          return {
            kind: 'CastError',
          } as const;
        }
        return unexpectedError(err);
      })
      .chain(_definedOrNotFound);
  }
}
```

Infine un esempio di utilizzo di questa API si ha nel package *server*, nella dichiarazione della risorsa *Library*.

```
export class LibraryService extends BaseService<Library> {
  constructor() {
    super(LibraryModel);
  }
}

export interface LibraryService
  extends FindById<Library>,
    Create<Library>,
    FindAll<Library>,
    Remove<Library>,
    Update<Library> {}

applyMixins(LibraryService, [
  SimpleFindById,
  Create,
```

```
17    FindAll,
18    ProtectedRemove,
19    ProtectedUpdate,
20  ]);
```

Capitolo 7

Test

Durante la realizzazione del progetto, sono stati adottati due approcci principali per testare le funzionalità realizzate:

- Realizzazione di test con *Jest*
- Realizzazione di test con *Postman*

7.1 Jest

Il framework *Jest* permette di realizzare unit test su codice Javascript. È stato utilizzato per testare il controller del primo componente con cui ci si interfaccia, ossia quello relativo alla registrazione dell' utente ed alla successiva autenticazione. Nello specifico, si verifica se l'autenticazione avviene con successo con un utente registrato, e viceversa che non avvenga, ed infine che la registrazione di un utente fallisca nel caso si scelga una email già utilizzata da un altro utente.

7.2 Postman

Nella realizzazione degli endpoint, si è verificato puntualmente che essi rispondessero nel modo corretto, effettuando simulazioni tramite *Postman*, con body di dati corrispondenti a quelli che avrebbe inviato la parte client. In figura 7.1 e figura 7.2 vi sono due esempi di test eseguiti con *Postman*.

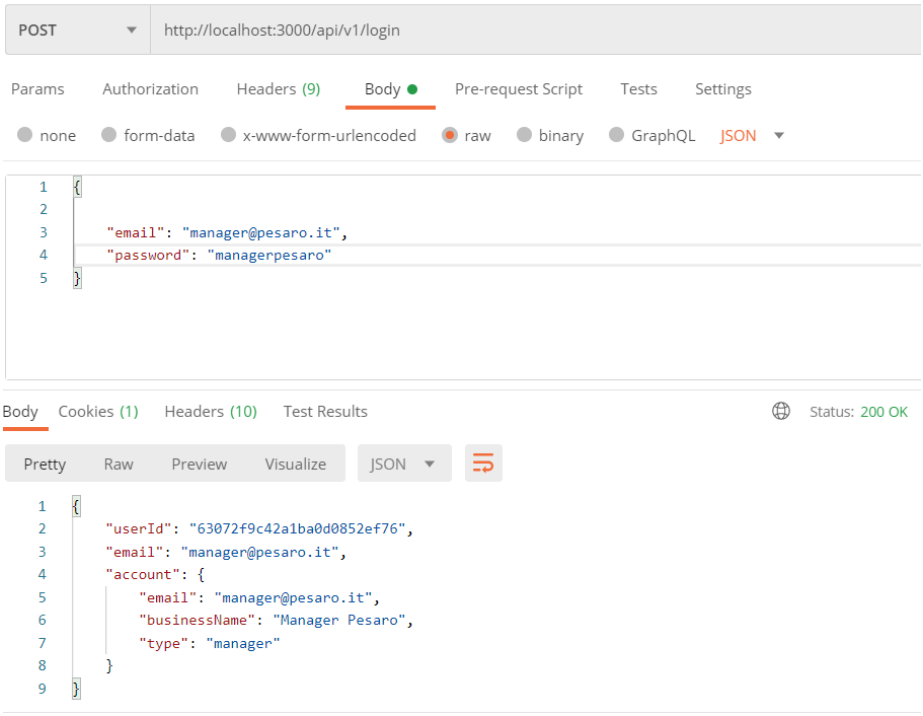


Figura 7.1: Login con account di tipo Manager

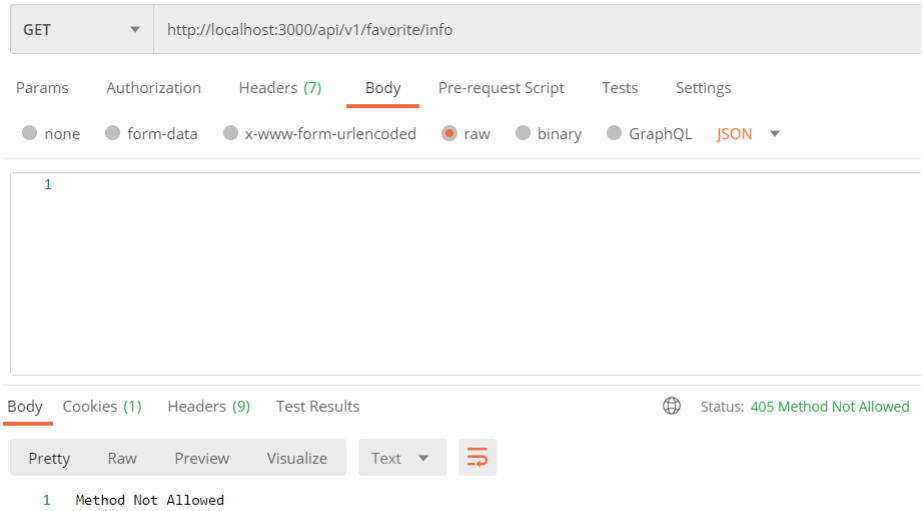


Figura 7.2: Errore di metodo non consentito su account di tipo Manager

Capitolo 8

Deployment

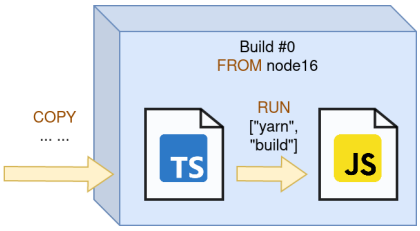
Una volta che l'applicazione è stata testata è stato necessario sviluppare una configurazione di deployment. Questa è stata agevolata dall'uso di strumenti come Docker e *Docker Compose*. Docker è un sistema di virtualizzazione, che permette di eseguire processi Linux all'interno di quelli che vengono definiti *container*, ovvero ambienti isolati dall'host; *Docker Compose*, invece è un sistema che permette di scrivere configurazioni per il deployment di applicazioni multi-container Docker.

Durante tutto il ciclo di vita del progetto è stato fatto uso estensivo di questi strumenti, dalle fasi di sviluppo a quella di messa in produzione. Infatti durante lo sviluppo sono stati utilizzati *container* per il deployment di servizi come MongoDB, Redis, ecc.

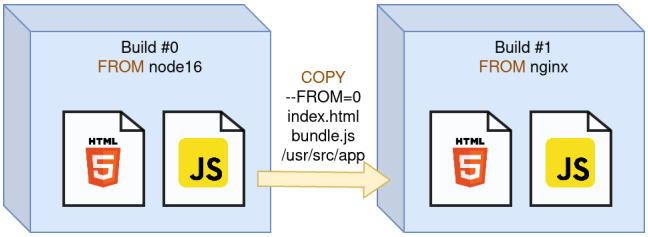
8.1 Build multi-stage

Per poter essere eseguiti all'interno di un *container*, i due artefatti prodotti necessitano di essere configurati in un'immagine Docker. Questo è possibile mediante un Dockerfile: un file di istruzioni finite e riproducibili che illustra a Docker come costruire l'immagine.

Al fine di produrre immagini più leggere e che contenessero solo il minimo indispensabile all'esecuzione, sono state realizzate, sia per il *back-end* che per il *front-end*, delle *build multi-stage*. Questa tipologia prevede che la fase di costruzione dell'immagine avvenga in passi che sono eseguiti in container diversi. In questo modo è possibile effettuare la compilazione del codice TypeScript in JavaScript all'interno di un container. Quando questa è terminata si crea un nuovo container, che sarà quello finale, nel quale verranno copiati solo gli artefatti prodotti. Nel caso del *back-end*, sarà necessario copiare anche tutte le dipendenze, in quanto queste verranno importate durante il *runtime*; mentre nel caso del *front-end* sarà sufficiente copiare il *bundle* prodotto da Webpack, che include al proprio interno tutte le dipendenze utilizzate dall'applicazione.



(a) Compilazione dei file TypeScript tramite Webpack.



(b) Copia degli artefatti prodotti all'interno di un container con un server HTTP statico.

Figura 8.1: Rappresentazione semplificata della procedura di creazione dell'immagine del *front-end*.

8.2 Configurazione in produzione

Per quanto riguarda la configurazione in produzione, questa beneficia della risoluzione dei nomi messa a disposizione da Docker Compose, per mettere in comunicazione tra di essi i servizi. Segue le indicazioni fornite dalla Figura 8.2. In particolare i container dispiegati sono:

- *mongo*: contiene un'istanza di MongoDB;
- *mongo-seed*: container che esegue un processo per popolare il database con dei valori fittizi per poi terminare;
- *s3-storage*: contiene un'istanza di MinIO, un *object storage* gratuito e open-source, compatibile con lo standard utilizzato da AWS S3; espone pubblicamente la porta 9000, poiché le immagini caricate dagli utenti gli verranno richieste direttamente.
- *s3-seed*: container che esegue un'inizializzazione dello storage di MinIO, creando un utente non *root* con delle credenziali predefinite, concordate con il container *server*; quando ha terminato il suo lavoro termina;
- *server*: container che ospita l'immagine Docker dell'applicazione *back-end* realizzata; espone la porta 3000, poiché dovrà essere raggiungibile dall'applicazione front-end, che essendo eseguita all'interno del browser

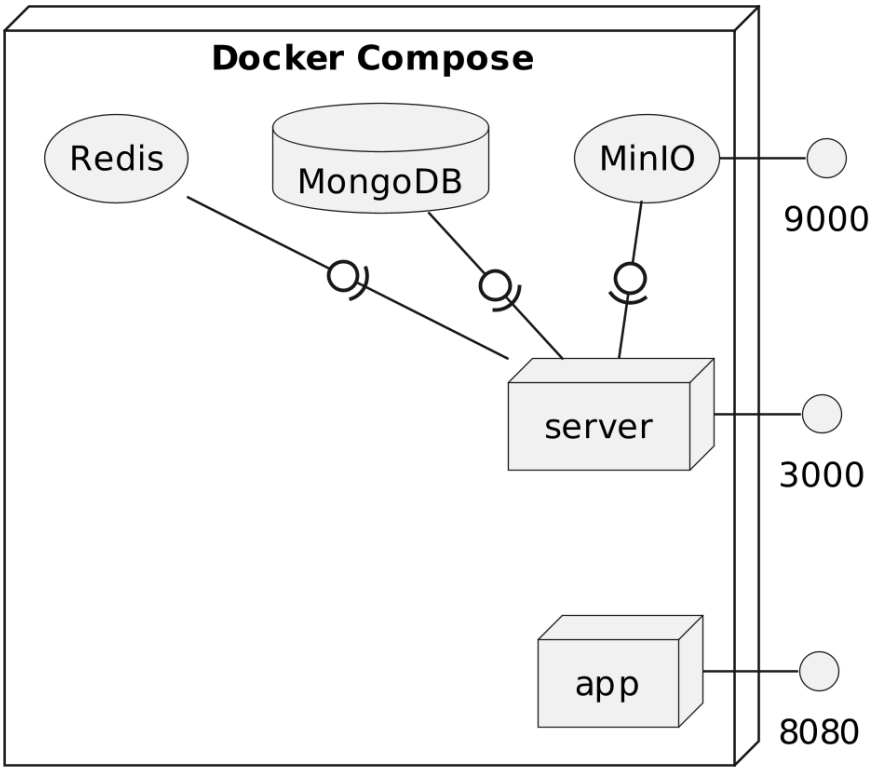


Figura 8.2: Diagramma UML di deployment che rappresenta la configurazione in produzione del sistema realizzato.

dell'utente finale, non potrà beneficiare della risoluzione dei nomi fornita da Docker Compose;

- *app*: container che ospita l'immagine Docker dell'applicazione *front-end*; espone la porta 8080.

Capitolo 9

Conclusioni

È stato realizzato un sistema di prenotazione di posti all'interno di aule studio. Questo sistema è stato realizzato tramite una *single page web application* e un sottosistema *back-end*, che espone le proprie funzionalità mediante un'interfaccia HTTP. Per supportare quest'ultimo sono state impiegate tecnologie moderne e rilevanti nel mercato attuale, come Redis, AWS S3 e MongoDB.

L'attenzione posta nella scelta della tipologia di repository, nella sua creazione e nella suddivisione in package ha permesso di avere un *workflow* di sviluppo agevole, permettendo di sviluppare utilizzando al massimo i vantaggi di TypeScript, garantendo ad entrambi i sistemi *type safety*.

Al termine della fase di sviluppo il sistema è stato presentato agli utenti del focus group A, per ottenere dei *feedback*. Questi, generalmente positivi, hanno manifestato le seguenti critiche:

- inserimento di una cella di testo al posto del QR code, per i dispositivi che non integrano una fotocamera;
- inserimento di una sezione per il cambio delle credenziali d'accesso (non attualmente implementato per mancanza di risorse);
- visualizzazione da parte del gestore dello stato attuale di una stanza;

9.1 Considerazioni finali

Il team si ritiene soddisfatto del progetto realizzato, dalle fasi iniziali, di analisi del dominio e dei requisiti, fino alla fase di sviluppo, che si è svolta agevolmente, senza mai riscontrare grossi problemi tecnologici. Questo è stato possibile grazie alle conoscenze acquisite durante il corso, sia di design delle interfacce, che tecniche, degli strumenti per lo sviluppo di applicazioni web, si sono rivelate fondamentali per procedere senza rallentamenti.

Infine sviluppare un sistema e metterlo in produzione, ci ha fatto confrontare con alcuni aspetti che a volte vengono posti in secondo piano: testing, configura-

zione di sviluppo/produzione, utilizzo di Docker per scrivere una configurazione di deployment.