Eventual

Applicazioni e Servizi Web

Francesco Foschini - 0001033912 {francesco.foschini2@studio.unibo.it}
Alessia Rocco - 0000983123 {alessia.rocco@studio.unibo.it}

03 Febbraio 2023

Indice

Capitolo 1

Introduzione

1.1 Descrizione del progetto

Eventual è un servizio web per la gestione di eventi, simil TicketOne. La piattaforma consente di registrarsi agli eventi come partecipante oppure crearne di nuovi se si è un utente di tipo organizzatore. Eventual permette inoltre di consultare gli eventi più recenti, acquistarli o modificarli, eliminazione compresa, in caso di un utente organizzatore.

Capitolo 2

Requisiti

2.1 Requisiti funzionali

2.1.1 Funzionalità relative all'utente non organizzatore (partecipante)

- Signup e Login;
- Visione del calendario/elenco degli eventi;
- Visione degli eventi a cui partecipa;
- Possibilità di acquistare un biglietto per un evento;
- Gestione del proprio profilo;

2.1.2 Funzionalità dell'utente organizzatore di eventi

- Signup e Login;
- Visualizzazione calendario/elenco degli eventi;
- Visione degli eventi a cui partecipa;
- Possibilità di acquistare un biglietto per un evento;
- Visione degli eventi che ha organizzato;
- Creazione/modifica/rimozione di un proprio evento;
- Gestione del proprio profilo;

2.1.3 Funzionalità utente admin

- Login;
- Visione del calendario/elenco degli eventi;
- Creazione/modifica/rimozione di un qualsiasi evento;
- Gestione e rimozione di un qualsiasi utente non admin;
- Gestione del proprio profilo;

2.2 Requisiti non funzionali

- Rendere il sistema facile da utilizzare anche ad utenti non esperti;
- Interfaccia utente responsive;
- Rendere il sistema sicuro mediante meccanismi di autenticazione;
- Rendere il sistema estendibile a nuove funzionalità;

2.3 Requisiti di implementazione

L'applicazione verrà sviluppata utilizzando lo stack **MERN**: sarà quindi utilizzato React per lo sviluppo dell'applicativo di frontend e bootstrap e SCSS per la grafica. Per il deploy dell'applicazione si sfrutterà il supporto docker.

Capitolo 3

Design

3.1 Design dell'architettura del sistema

Applicazione è stata costruita come stack MERN, acronimo di:

- Mongo DB
- Express
- React
- Node.js

MERN è una variante dello stack MEAN (dove lato frontend il framework Angular.js viene sostituito con React.js) che ha permesso l'uso di Javascript e JSON in tutta l'architettura, divisa in backend e frontend.

3.1.1 Componenti funzionali (React Hooks)

React Hooks sono una nuova feature introdotta in React 16.8 che forniscono un modo di utilizzare lo *state* e altre funzionalità di React senza dover scrivere una classe.

Gli *Hooks* permettono di agganciare il comportamento di un componente ad una funzione, rendendo più semplice condividere il codice tra diversi componenti e scrivere componenti più leggeri e facilmente comprensibili. Gli *Hooks* più comuni, utilizzati nel progetto, includono `useState` e `useEffect`. Queste funzioni prendono in input valori e restituiscono uno stato che può essere utilizzato all'interno del componente.

- `useState`: restituisce un array di due elementi: un valore che rappresenta lo stato corrente e una funzione che permette di modificare lo stato.
- `useEffect`: utilizzato per gestire gli effetti collaterali all'interno di un componente, ovvero tutte le operazioni che hanno un impatto sull'ambiente esterno al componente (come ad esempio la richiesta di dati da un API).

Gli `Hooks` sono progettati per essere utilizzati solo all'interno di componenti React o di funzioni custom Hook, e non possono essere utilizzati all'interno di classi o fuori dalla libreria di React.

3.1.2 JSX

Un'altra sintassi utilizzata all'interno del progetto è quella JSX fornita da React. E' una sintassi estesa per JavaScript che viene utilizzata per la descrizione della struttura dell'interfaccia utente in React. JSX appare come HTML all'interno di un componente React, ma di fatto viene tradotto in chiamate JavaScript alla libreria React.

3.1.3 Axios

Axios è una libreria promised-based JavaScript che consente di effettuare richieste HTTP (come GET, POST, PUT, DELETE, etc.) a un'applicazione, ad esempio, in Node.js. Fornisce una vasta gamma di funzionalità, tra cui la gestione dei dati in formato JSON, la gestione degli errori, la gestione della cache e molto altro. E' stata utilizzata lato frontend, per la richiesta di dati al backend. Si basa tutto sul concetto di Promise, ovvero la rappresentazione di un'operazione asincrona differita, nel caso di Axios, che non è ancora completata e che verrà risolta in futuro. Axios infatti mette a disposizione dei metodi asincroni a cui possono essere associate handlers di gestione del successo o fallimento delle azioni richieste.

3.1.4 Design finale dell'architettura

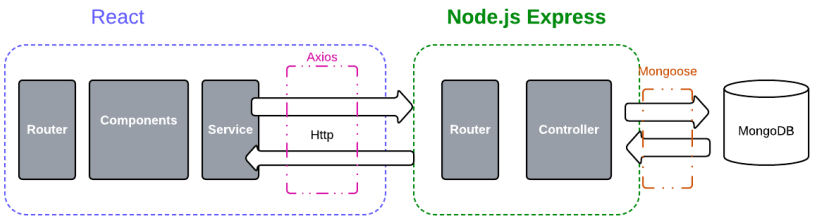

Figura 3.1: MERN architecture and interactions

3.2 Metodologie di sviluppo

Il modello utilizzato è di tipo iterativo e basato su User Centered Design con utenti virtualizzati. La scelta è ricaduta su questa metodologia in quanto sin

dalle prime fasi di progettazione, il team ha messo in primo piano la HCI (Human Computer Interaction), cercando di focalizzarsi sulle esigenze degli utenti e sull'ottenimento di una buona usabilità. Sono stati individuati gli utenti target dell'applicazione e su di essi si sono sviluppate le *Personas* che hanno contribuito alla comprensione delle funzionalità del sistema e come possano esse rispondere ai requisiti utente. Il team ha gestito tutto il progetto tramite una dashboard Trello che ha permesso di organizzare lo sviluppo in vari sprint; in ognuno dei quali sono state individuate le funzionalità da implementare. Il team ha seguito quindi un framework di sviluppo ispirato a quello AGILE. Nello svolgimento del progetto e del design delle interfacce si è sempre cercato di rispettare i seguenti principi:

- Responsive Design: sono state realizzate pagine Web in grado di adattarsi graficamente e in modo automatico ai dispositivi coi quali vengono visualizzati;
- Mobile First: sono state progettate prima le interfacce per i dispositivi più limitanti (smartphone e tablet) per poi passare alla progettazione di interfacce per i PC;
- KISS: le interfacce sono state realizzate per contenere solamente gli elementi fondamentali ed in modo facilmente accessibile all'utente.

3.3 Mockup iniziale

Nella prima fase di design si sono realizzati i seguenti mockup progettando l'app mobile first e con la scelta di una grafica pulita, organizzata e intuitiva all'uso. In seguito ai feedback da parte di alcuni utenti selezionati si è arrivati ai seguenti prototipi:

Ogni link nella navabar conduce alla pagina associata. Nella Home inizialmente abbiamo pensato di far vedere, come una sorta di vetrina, i 10 eventi più recenti. Lo Store, invece reindirizza tutti gli eventi che si svolgeranno dalla data odierna in poi. Ogni evento dallo Store è possibile aggiungerlo nel carrello ma solo una volta loggati si potrà procedere all'acquisto.

Un utente si può registrare come partecipante o come organizzatore di eventi, scegliendo tale opzione al momento del Signup. Solo gli utenti registrati possono quindi loggarsi e vedere i propri eventi, comprare biglietti per eventi o nel caso di organizzatori crearne di nuovi. Alcune delle funzionalità descritte sono riportate nei mockups di seguito.

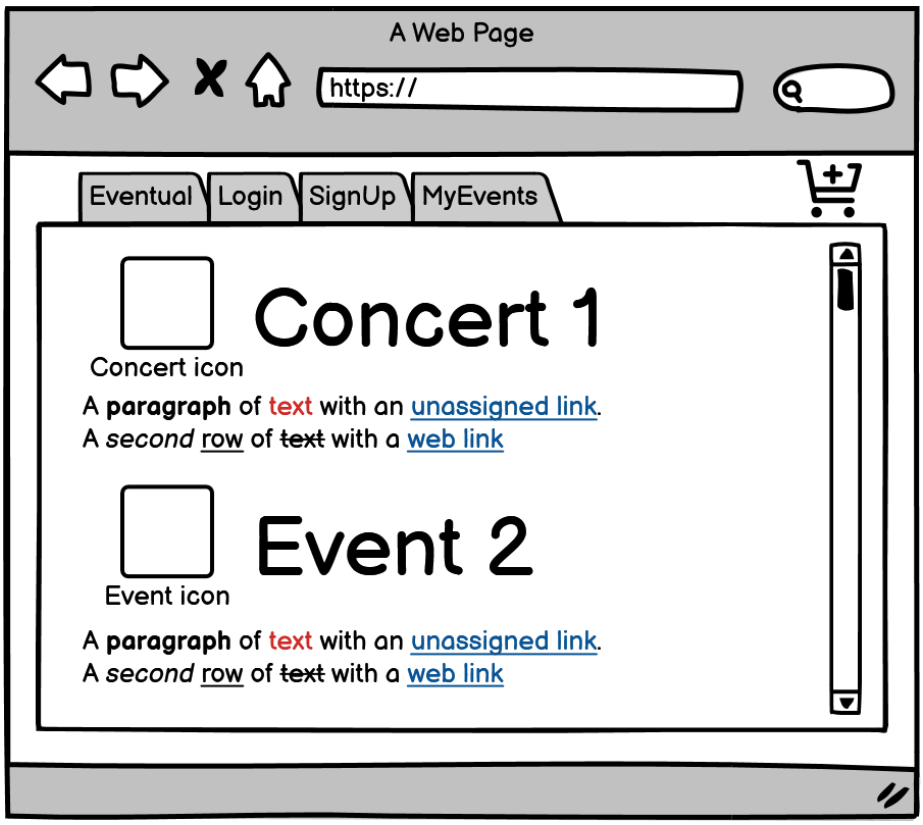

Figura 3.2: Wireframe desktop

Figura 3.3: Signup

Figura 3.4: Login

Figura 3.5: Events Store

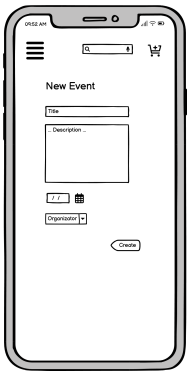

Figura 3.6: New Event

3.4 Target User Analysis

Come target principale del sistema sono state individuate due tipologie di utente: la persona che vuole organizzare un evento (organizzatore) e il partecipante (utente).

Di seguito si sono identificate diverse *personas* differenziate da ciò che si aspettano dal sistema e servizio che esso gli offre. L'idea di business è rivolta infatti a uno specifico gruppo di potenziali clienti, ovvero coloro interessati a eventi come concerti, serate, lauree e compleanni; in quanto si reputa il più adatto a recepire e desiderare l'applicativo.

Si è deciso d'identificare e segmentare il pubblico a una cerchia di potenziali utilizzatori dell'applicazione che avessero esigenze in comune alle caratteristiche dell'applicazione. L'analisi è stata per tanto condotta tenendo in considerazione le seguenti guidelines:

- contenuti specifici: comunicare al pubblico contenuti mirati e appropriati, come per esempio gli eventi vicini all'utente o delle categorie d'interesse;
- ottimizzare il sito: considerare come il target naviga il sito, se da mobile, tablet e pc e se il target ha o meno dimestichezza nel farlo. Favorire l'esperienza ai nuovi clienti attraverso un design intuitivo, pratico e accattivante.
- concentrazione sul potenziale: il focus sarà principalmente su utenti e organizzatori di eventi quali privati e aziende del settore dei concerti, nel futuro l'espansione potrà comprendere anche la considerazione di ristoranti e locali.
- strategie di successo: scegliere quali media e quali tecniche usare per la promozione del prodotto finale.

Le principali guidelines hanno portato quindi a definire due tipologie di personas e scenari d'uso, riportati di seguito.

3.4.1 Personas

Personas: Sofia

Sofia ha appena scoperto un nuovo artista che le piace molto. *Scenario d'uso*: Sofia vuole partecipare a un concerto del suo nuovo artista preferito.

- Sofia accede alla piattaforma;
- Sofia si iscrive come utente, registrando i suoi dati;
- Sofia cerca l'artista e il concerto a cui vuole partecipare;
- Sofia mette nel carrello l'evento ed eventualmente lo acquista;
- Sofia può contattare l'organizzatore e controllare/modificare il suo evento prenotato.

Personas: Matteo e Lorenzo

Matteo e Lorenzo sono due imprenditori adulti e affermati nel loro settore che vogliono rinnovare i loro locali organizzando nuovi eventi e serate ogni settimana. *Scenario d'uso*: Matteo e Lorenzo vogliono sponsorizzare i loro eventi organizzandoli su una piattaforma che gli consenta di tenere traccia del numero di partecipanti, i pagamenti e dell'organizzazione del materiale. In questo modo puntano a raggiungere un pubblico maggiore e incrementare i loro guadagni.

- Matteo e Lorenzo accedono alla piattaforma;
- Matteo e Lorenzo si iscrivono come organizzatori, registrando i dati della loro organizzazione;
- Matteo e Lorenzo creano un nuovo evento;
- Matteo e Lorenzo hanno la necessità di modificare gli eventi o eliminarli;
- Matteo e Lorenzo incassano poi il ricavato dalla vendita dei biglietti di un evento;

Capitolo 4

Tecnologie

All'interno del progetto, essendo sviluppato con stack MERN, sono state utilizzate diverse tecnologie, che differiscono da frontend a backend. MERN ha offerto tutto il necessario per lo sviluppo di un'applicazione cross-platform, indipendente al sistema operativo utilizzato.

4.1 Frontend

Per la parte frontend sono state utilizzate:

- **React**: si appoggia sul pattern MVVM;
- **Hooks e LocalStorage**: per la condivisione di stati e informazioni persistenti;
- **Saas** e **Bootstrap**: per lo stile delle pagine;
- **Axios**: libreria Javascript per richieste http;
- **Jest**: libreria per il test di applicativi react.

4.2 Backend

Per la parte backend sono state utilizzate:

- **Node.js**;
- **Mongoose**: come client per comunicare con MongoDB;
- **Express**;
- **Bcryptjs**: per l'"hashing" delle password;
- **MongoDB**;
- **Mongo-Express**: come interfaccia sul web per avere un supporto nel controllare visivamente la struttura del database.

4.3 Linguaggi

I linguaggi utilizzati all'interno del progetto sono:

- Typescript: lato client;
- Javascript: lato server;
- Html: struttura delle pagine web;
- Scss: fogli di stile.

4.4 Altre tecnologie

Si è pensato di utilizzare anche il client Mongo-Express per facilitare la gestione del database su MongoDB via interfaccia grafica. E' importante precisare che MongoDB grazie alla memorizzazione di documenti JSON, ha facilitato notevolmente l'archiviazione, la manipolazione e la rappresentazione dei dati ad ogni livello dell'applicazione.

Capitolo 5

Codice

Di seguito verranno riportate le porzioni di codice che meglio supportano la comprensione del progetto.

5.1 Struttura Database

Le due entità principali del dominio applicativo sono gli utenti del sistema e gli eventi. Gli *Users* modellano gli utenti. Viene mostrato di seguito il modello Mongoose:

```
const userSchema = new mongoose.Schema({
        name:  {
            type: String,
            required: true
        },
        surname: {
            type: String,
            required: true
        },
        email: {
            type: String,
            required: true
        },
        phone: Number,
        password: {
            type: String,
            required: true
        },
        birthday: Date,
        category: {
            type: String,
            required: true
```

```
        },
        inscriptions: [{ type: Schema.Types.ObjectId, ref: 'Events' }],
        my_organizations: [{ type: Schema.Types.ObjectId, ref: 'Events' }]
    })
```

I campi marcati come "required" sono obbligatori per ogni utente registrato nella collezione del database MongoDB. Gli utenti possono appartenere a tre categorie:

- Partecipanti: possono solo partecipare a eventi;
- Organizzatori: possono partecipare e organizzare eventi;
- Admin: possono creare, modificare o eliminare qualsiasi evento e possono eliminare un user dal sistema a patto che non sia un altro admin.

Il campo *inscriptions* raccoglie tutte le iscrizioni agli eventi: è un array di id degli eventi. *organizations*, invece, memorizza tutti gli id degli eventi organizzati dallo specifico utente.

La seconda entità fondamentale del sistema è *Events* che raccoglie tutti gli eventi memorizzati dal sistema. Di seguito è mostrato il modello Mongoose:

```
const eventSchema = new mongoose.Schema({
        title:  {
            type: String,
            required: true
        },
        author: {
            type: Schema.Types.ObjectId, ref: 'Users',
            required: true
        },
        category: {
            type: String,
            required: true
        },
        date: {
            type: Date,
            required: true
        },
        description: {
            type: String,
            required: true
        },
        users: [{ type: Schema.Types.ObjectId, ref: 'Users' }]
    })
```

Gli eventi, al momento, possono appartenere a tre categorie: concerti, feste e compleanni. Come per il modello degli utenti anche in questo caso tutti i campi

obbligatori sono marcati con "required". Il campo `users` memorizza tutti gli id dei partecipanti all'evento. I file `EventRoute.js` e `UserRoute.js` raccolgono tutte le rotte sul quale l'applicazione backend rimane in ascolto di richieste http per gestire e manipolare le due entità di sistema.

Nell'applicazione backend è stata utilizzata la libreria javascript *bcryptjs* per l'inserimento di password salate all'interno della collezione *users*. Di seguito è mostrato il codice eseguito nell'applicazione backend inseguito alla richiesta di registrazione di un nuovo utente nel sistema.

```
exports.sign_user = (req,res)=>{
    const { name, surname, email, phone, password, birthday, category} = req.body;
    // controllo se l'email e' gia' presente nel database
    UserModel.findOne({"email": email}, (err, user) => {
        if (err) {
            res.status(500).json({ error: 'Errore del server' });
        } else if (user != null) {
            res.status(409).json({error: 'Email utente gia' in uso'});
        } else {
            bcrypt.hash(password, 10, (err, hash) => { //10 numero di round di cifratura
                if (err) {
                    res.status(500).json({error: 'Errore del server'});
                }
                const newUser = new UserModel({
                    name,
                    surname,
                    email,
                    phone,
                    password: hash,
                    birthday,
                    category,
                    inscriptions: [],
                    my_organizations: []
                });
                //newUser.birthday = new Date(birthday)
                newUser.save((err) => {
                    if (err) {
                        //res.send("Salvataggio non possibile, alcuni campi non sono stati
                        res.status(500).json({ error: 'Errore del server' });
                    }
                    res.status(200).json({description:`User per ${name} ${surname} creato`})
                });
            })
        }
    })
}
```

Il nuovo utente, in questo modo, viene inserito nel database con tutti i suoi dati personali e con la password salata. Quando l'utente vuole eseguire il login, sfruttando il metodo `compare` della libreria *bcryptjs*, si controlla che la password inserita e quella salvata nel database a seguito della registrazione coincidano:

```
exports.log_user = (req, res)=> {
    UserModel.findOne({"email": req.body.email}, (err, user) => {
        if (err){
            res.status(500).json({ error: "Errore del server" });
        } else if (!user){
            res.status(404).json({error: "Utente non trovato"});
        } else {
            bcrypt.compare(req.body.password, user.password, (err, result) => {
                if (err){
                    res.status(500).json({ error: "Errore del server" });
                }
                if (!result){
                    res.status(401).json({error: "Password errata"});
                }
                res.status(200).json({description: "Benvenuto ${user.name} ${user.surname}"
            });
        }
    })
}
```

La libreria *bcryptjs* è stata utilizzata anche per l'aggiornamento delle credenziali dell'utente.

5.2 Frontend

5.2.1 Struttura organizzativa del codice

Come nelle applicazioni React standard la struttura dell'applicativo frontend è il risultato di una singola pagina html integrata man mano dai componenti react, che a loro volta definiscono sia il comportamento della GUI che la renderizzazione degli elementi html visualizzati. Pertanto nella cartella troviamo le immagini pubbliche, il logo, il manifest e l'`index.html` per la visualizzazione della pagina; mentre nella cartella si trovano tutti i componenti e le pagine principali.

E' importante precisare che il componente principale è `App.js` che contiene la definizione di tutte le rotte ai componenti e pagine del sistema. Per tal proposito si è adottato l'utilizzo della libreria `"react-router-dom"`. Il componente`<App/>` è infatti wrappato all'interno di un `<BrowserRouter>` per poter accedere al sistema di rotte in tutti i punti dell'applicazione.

Tutto il sistema frontend è messo in moto da `index.js` che estrae dal file `html` l'elemento `"root"` e renderizza il primo componente definito in `App.js`.

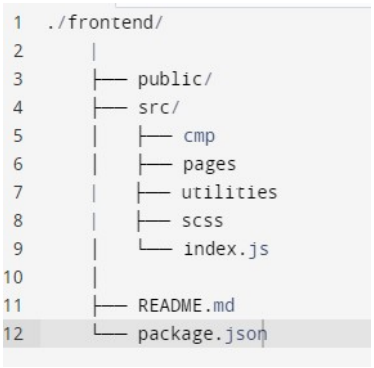

Figura 5.1: Struttura frontend

5.2.2 Gestione stato con Hooks e LocalStorage

Come definito precedentemente si è adottato l'uso degli **Hooks** per l'aggiornamento e renderizzazione automatica degli elementi visivi in seguito al cambiamento del loro stato o dei loro valori. Di seguito è riportato un esempio dell'uso dei React Hooks:

```
export function Home(){
    const [events, setEvents] = useState([])

    useEffect(() => {
        axios.get("http://localhost:8082/events")
            .then(res =>{
                setEvents(res.data)
            })
            .catch(error => console.error(error))
    }, [events])

    return(
            <>
                <Home {...events}/>
            </>
    )
}
```

In più, sempre sfruttando le funzionalità messe a disposizione da React, si è utilizzata la logica readonly del `localStorage` per il salvataggio permanente di alcune informazioni, come per esempio i dati dell'utente loggato o gli elementi salvati nel carrello. Questo ha permesso la creazione solida dell'applicativo perché i dati, anche a seguito di un'eventuale ricarica della pagina o di perdita di connessione, sono mantenuti in locale. Di seguito un esempio del suo utilizzo per la creazione costumizzata del salvataggio degli elementi nel carrello, ma in

generale è sfruttabile per qualsiasi estensione futura grazie all'utilizzo dei tipi generici.

```
import {useEffect, useState} from "react";
export function useLocalStorage<T>(key:string, initialValue: T | (() => T)){
    const [value, setValue] = useState<T>(() => {
        const jsonValue = localStorage.getItem(key)
        if(jsonValue != null) return JSON.parse(jsonValue)

        if(typeof initialValue === "function"){
            return (initialValue as () => T)()
        } else {
            return initialValue
        }
    })

    useEffect(() => {
        localStorage.setItem(key, JSON.stringify(value))
    }, [key, value])

    return [value, setValue] as [typeof value, typeof setValue]
}
```

5.2.3 Sass

Per quanto riguarda i fogli di stile si è utilizzato il super set Sass per sfruttare in particolare le potenzialità della definizione di variabili e `mixins`. Un esempio di `mixins` è stato adottato per la definizione della funzione responsive-height per la gestione automatica dell'altezza del contenuto delle pagine.

```
    // main: index.scss
    @use "sass:math";

    @mixin background-cover($img_url) {
      background: url($img_url) no-repeat center center;
      -webkit-background-size: cover;
      -moz-background-size: cover;
      background-size: cover;
    }

    @mixin responsive-height($x, $y) {
      height: 0;
      padding-bottom: math.div($y, $x) * 77%;
    }
```

Capitolo 6

Test

L'applicativo è stato testato ad ogni chiusura di Sprint dal team insieme mentre individualmente di volta in volta che si aggiungevano funzionalità o venivano apportate modifiche al codice. Principalmente il sistema è stato testato su vari Browser di riferimento (Chrome, Firefox, Edge e Safari) con l'obiettivo di verificare la correttezza e la portabilità del progetto.

Il sistema è stato inoltre sottoposto, a seguito dei test con utenti, all'euristica di Nielsen. Il risultato che ha portato al prodotto finale ha tenuto conto di aspetti quali: la prevenzione di errori, presenza di elementi in aiuto all'utente in tutta l'applicazione, riconoscimento piuttosto che ricordo, corrispondenza con il mondo reale e un'estetica e progettazione minimalista.

6.1 Test Backend

Le API e le rotte sviluppate lato backend sono state testate con l'applicazione PostMan per poter garantire il loro corretto funzionamento. In figura 6.1 è riportato il test per verificare il corretto funzionamento del login di un utente del sistema.

In figura 6.2, invece, viene mostrato il test relativo al signup dell'utente di sistema.

In figura 6.3, infine, viene testato l'aggiornamento dell'utente nel sistema. Questa rotta viene chiamata, ad esempio, a seguito della modifica delle credenziali dell'utente oppure successivamente all'iscrizione a un evento o all'organizzazione di uno nuovo.

6.2 Test Frontend

6.2.1 Testing Gui

Il test per il corretto funzionamento di ogni componente della Gui è stato supportato sia dal controllo tramite l'utilizzoo dei log a console che dalla possibilità

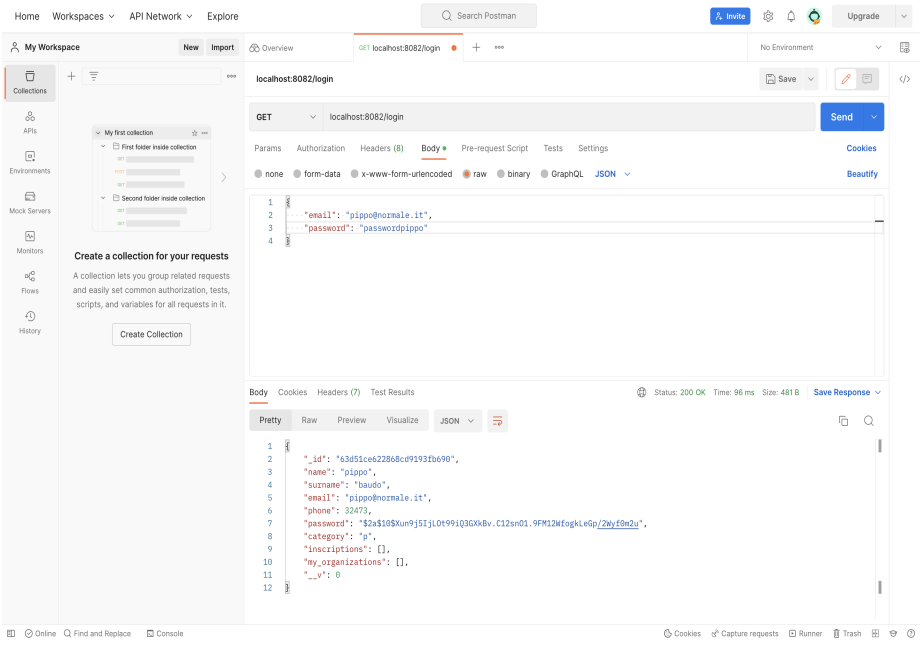

Figura 6.1: test login backend side

di visualizzare in tempo reale lo storage locale, come mostrato di seguito.

6.2.2 Testing Jest

Lato frontend, l'applicazione è stata anche testata utilizzando il framework Jest che mette a disposizione una serie di metodi come `match()` e `expect()` che consentono di riprodurre la renderizzazione di un componente React e testare il suo corretto comportamento. In questo progetto non ci si è focalizzati sull'utilizzo avanzato della libreria ma piuttosto sulla realizzazione di test semplici e immediati per controllare la renderizzazione corretta delle rotte e dei componenti. Un estratto dei test che si possono trovare sono definiti in `App.test.js` è riportato nell'immagine 6.5.

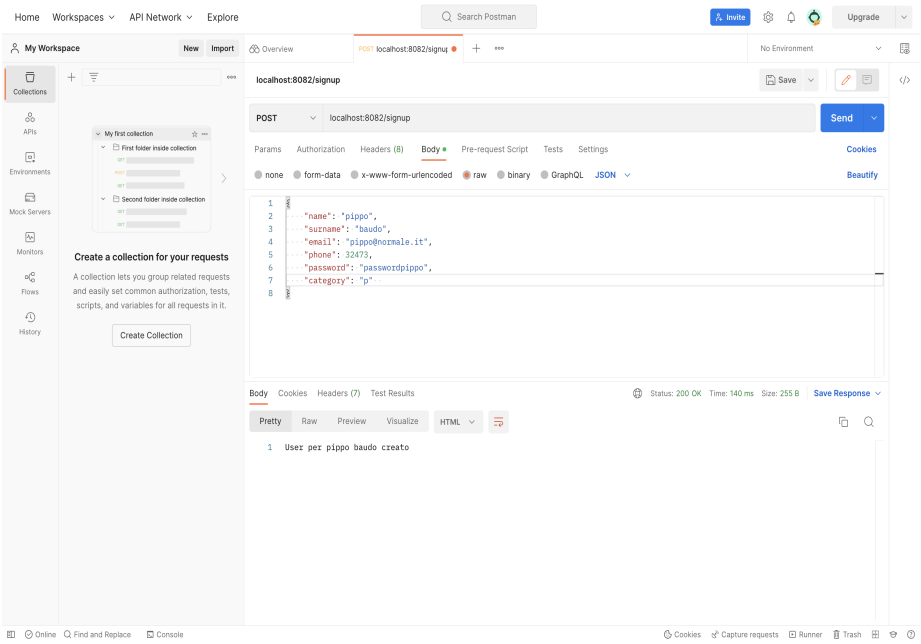

Figura 6.2: test signup backend side

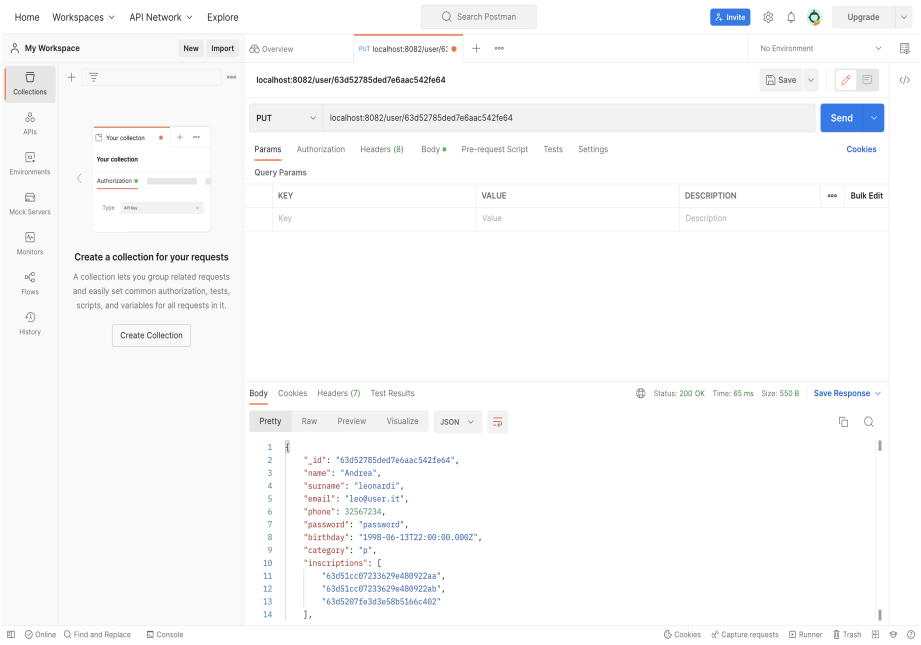

Figura 6.3: test update user backend side

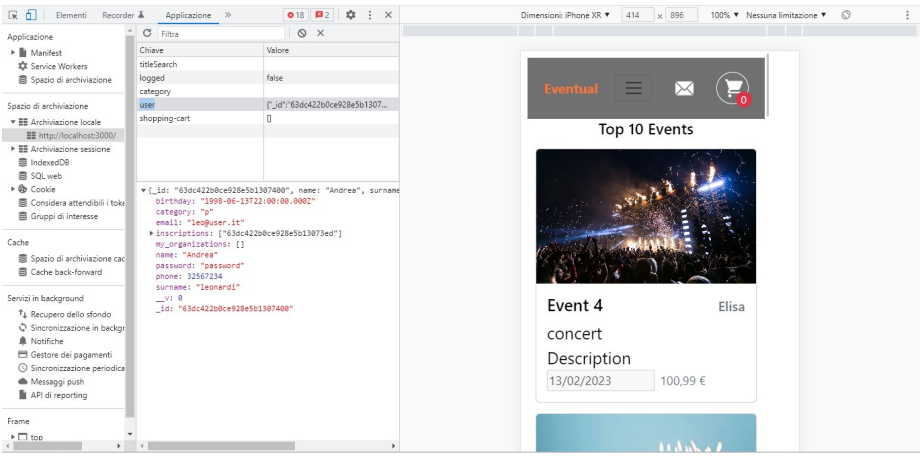

Figura 6.4: test localstorage da ispeziona

```
test( name: 'Application Routes renders correctly', fn: () => {
    const tree = renderer
        .create(
            <BrowserRouter>
                <Link to="/">Home</Link>
                <Link to="/new_event">NewEvent</Link>
                <Link to="/login">Login</Link>
                <Link to="/admin">Admin</Link>
            </BrowserRouter>
        )
        .toJSON();
    expect(tree).toMatchSnapshot();
});

it( name: 'Signup component is loaded', fn: () =>{

    render(
        <BrowserRouter>
            <SignUp />
        </BrowserRouter >
    );

    const element = screen.getByText(/Sign/i);
    expect(element).toBeInTheDocument();
});

it( name: 'Login component is loaded', fn: () =>{
```

Figura 6.5: test GUI

Capitolo 7

Deployment

Per gestire il deployment dell'applicativo si è deciso di utilizzare Docker. Per maggiori dettagli sui comandi specifici da eseguire per il deploy dell'intera applicazione fare riferimento al file *IstruzioniPerDeploy.md* più esterno. Come prima cosa, per sicurezza, entrare nelle due cartelle dei progetti *frontend* e *backend* e eseguire il comando *npm install* per scaricare tutte le dipendenze del progetto. Successivamente, è necessario importare in locale le immagini docker presenti su Dockerhub di mongodb, mongo-express e eseguire la *build* per la costruzione delle immagini delle applicazioni del backend e del frontend. In questo modo, in locale, saranno presenti le seguenti quattro immagini docker:

- immagine applicativo backend;
- immagine applicativo frontend;
- mongo (immagine docker di MongoDB);
- mongo-express (immagine docker di Mongo-Express).

Partendo da queste quattro immagini docker è possibile eseguire il deploy dell'applicazione in due modi: con *docker-compose classico* oppure in modalità *swarm*. Si è pensato di tenere in considerazione il deploy dell'applicativo anche in modalità swarm perché fornisce in automatico funzionalità di bilanciamento di carico e repliche dei servizi. Docker Swarm, infatti, utilizza un algoritmo di bilanciamento del carico per distribuire equamente le richieste alle risorse e assicurare una distribuzione uniforme del carico di lavoro. Questo migliora la disponibilità e la tolleranza alle avarie del sistema, poiché le richieste possono essere gestite da più nodi in caso di problemi con un singolo nodo. I servizi del sistema in esecuzione in modalità swarm sono:

- applicativo backend, replicato 3 volte e in ascolto su porta 8082;
- applicativo frontend, 1 replica in ascolto su porta 3000;
- MongoDB, 1 replica in ascolto su porta 27017;

- mongo-express, 1 replica in ascolto su porta 8081;

A causa delle repliche, della pesantezza dell'applicazione e essendo tutto in locale, il primo avvio dell'applicativo in modalità *swarm* potrebbe risultare non immediato. Sopratutto bisognerà aspettare che il container di MongoDb si attivi. Per questi motivi abbiamo notato che, la prima volta che si mette in esecuzione l'applicativo, è consigliabile farlo in modalità *Docker compose classico* in modo che venga costruita l'infrastruttura di rete sottostante che permette ai diversi servizi di comunicare. Dal secondo avvio in poi si potrà eseguire l'applicativo in modalità swarm agilmente.

Se una delle istanze dei servizi va in down sarà instanziato automaticamente un nuovo container dello stesso servizio perché è stato impostato *"restart: always"* in tutti i *services* del file *docker-compose.yaml* Se, invece, si volesse eseguire solamente un container specifico dell'applicativo frontend o dell'applicativo backend fare riferimento ai file *IstruzioniPerDeploy.md* dentro alle rispettive cartelle.

Di seguito è presente il frammento di *docker-compose.yaml* da cambiare in base al tipo di esecuzione scelto. Quando si vuole eseguire l'applicativo in modalità *swarm* decommentare (togliere i) *deploy*, *mode* e *replicas*. Viceversa, commentarli se si vuole eseguire in modalità *Docker compose classico*.

```
eventual-backend:
  depends_on:
    - mongodb
  image: node-eventual
  environment:
    MONGO_HOST: mongodb #nome immagine che corrisponde all'IP grazie al dns di docker
  ports:
    - "8082:8080"
  restart: always
  #deploy:
    #mode: replicated
    #replicas: 3
```

Avendo impostato, come mostrato di seguito, il parametro *volumes* nel servizio *mongodb* i dati nel database rimangono persistenti in locale:

```
mongodb:
  image: mongo
  restart: always
  volumes:
    - ./data/db:/data/db
    - ./data/configdb:/data/configdb
  environment:
    MONGO_INITDB_DATABASE: test
  ports:
    - "27017:27017"
```

```
command: --journal
user: "1000:1000"
```

Successivamente all'avvio dell'applicativo, è possibile andare nella cartella del progetto di backend e eseguire:

- *seed.js*: per popolare il database con alcuni dati di partenza;
- *poll-localhost.sh* : per testare la connessione all'applicativo backend. Se sarà stampata su console la stringa "ciaoooo" la connessione è andata a buon fine.

Capitolo 8

Sviluppi futuri

In futuro potrebbe essere aggiunta una sezione che permetta all'utente di essere notificato degli eventuali aggiornamenti degli eventi a cui si sono iscritti (es. cambio del giorno). Un altra funzionalità che potrebbe essere aggiunta è la ricerca degli eventi in base alla data e alla città.

Infine si potrebbe estendere il sito con una chat in tempo reale e la possibilità di connettere profili utente tra di loro. Creando per esempio entità famiglie o amici che possano comprare e condividere biglietti per gli eventi.