

Piattaforma Gestione Spiagge

Applicazioni e Servizi Web

Arianna Soriani - 0001005647 {arianna.soriani@unibo.it}

June 12, 2023

0.1 Introduzione

Il progetto mira alla realizzazione di una web app per la fruizione intelligente delle spiagge.

La web app gestisce 10 diversi stabilimenti balneari e offre agli utenti la possibilità di prenotare postazioni, comprensive di ombrelloni e lettini, e di valutare le diverse schede descrittive di ogni stabilimento.

0.2 Requisiti

Le funzionalità principali che il sistema deve offrire sono:

- Prenotazione postazione a mare (ombrelloni e lettini)
- Schede degli stabilimenti/chioschi

L'applicazione deve fornire, inoltre, opportuni meccanismi di push notification per segnalare l'esito delle interazioni da parte degli utenti con il sistema.

0.3 Design

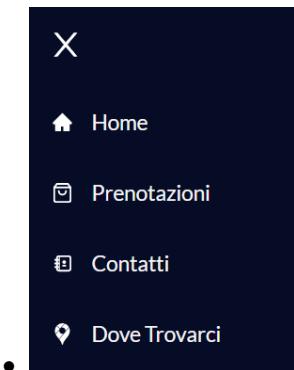
L'utente può interagire con il sistema attraverso un insieme di interfacce diverse. Procedendo secondo la logica dell'applicazione, troviamo:

0.3.1 HomePage

L'interfaccia utente principale è la **HomePage** che viene visualizzata attraverso il caricamento della route iniziale (path=“/”).

Essa è costituita da un flexbox, denominato **HomeWrapper**, che ricopre il 100% dell'area di visualizzazione. Per mezzo della funzione **useEffect** di React, in esso scorrono diverse immagini di sfondo, distanziate l'una dall'altra da un timeout prefissato.

Nella HomePage, inoltre, sono presenti i seguenti elementi:



- **Navigation bar;** un flexbox in posizione fissa, che permette la navigazione tra le diverse pagine della web app.

Il menu è a scorrimento e può essere visualizzato e nascosto facendo click sui rispettivi simboli di apertura e di chiusura della finestra.

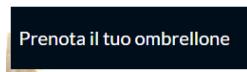
Ogni voce del menu renderizza ai rispettivi percorsi selezionati.

Dal menù è possibile, quindi, accedere direttamente alle pagine di:

- **Home**
- scelta del bagno e **Prenotazioni**
- **Schede tecniche** dei bagni
- **Mappa e Contatti**



- **Bottoni di scorrimento tra slide;** essi permettono di forzare lo scorrimento delle immagini di sfondo della HomePage.



- **Bottone di prenotazione;** permette il renderizzamento diretto alla pagina di scelta del bagno e prenotazione delle postazioni a mare.

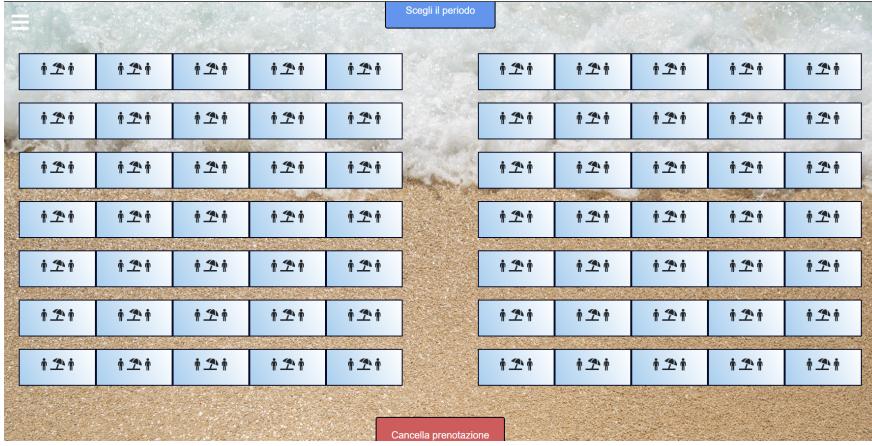
0.3.2 ReservationPage

All'apertura della **Pagina di Prenotazione** si ha la possibilità di selezionare, in prima istanza, il bagno in cui si intende procedere alla prenotazione di una postazione a mare, secondo le preferenze dell'utente.

Gli stabilimenti balneari disponibili sono 8, visualizzabili come segue:



Una volta effettuata la scelta, si apre la scheda di prenotazione corrispondente allo specifico bagno selezionato.



Al centro dell'interfaccia si trova la rappresentazione tabellare delle diverse postazioni, dal bagno alla riva. In questa pagina, l'utente ha la possibilità di:

- Effettuare una prenotazione
- Cancellare una prenotazione

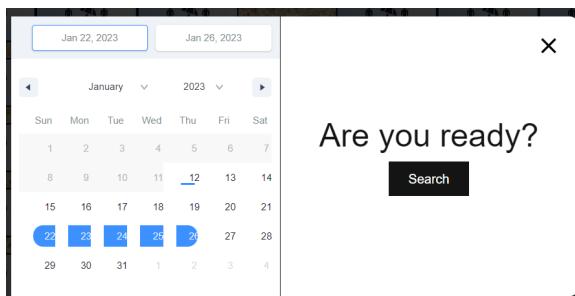
Per **Effettuare una prenotazione** è necessario selezionare in sequenza:

- Il periodo richiesto
- La postazione desiderata
- Confermare la prenotazione

E' possibile fare click sulla postazione desiderata solo dopo aver selezionato il periodo di interesse e solo se non risulta già riservata ad un altro utente per le date selezionate.

Questa interfaccia permette all'utente di gestire il sistema di prenotazione delle postazioni balneari. Essa è composta dai seguenti componenti:

- **Scegli il periodo** Bottone scelta del periodo; in seguito al verificarsi dell'evento OnClick mostra il component **Calendar**.



- **Calendario**; permette di selezionare il range di date, corrispondente all'inizio e alla fine del peri-

odo desiderato. In seguito al verificarsi dell'evento OnClick, in corrispondenza del bottone Search, il software effettua una ricerca di disponibilità di postazioni balneari, per il periodo richiesto e specificatamente per lo stabilimento selezionato; al termine, visualizza sullo schermo il risultato della ricerca.

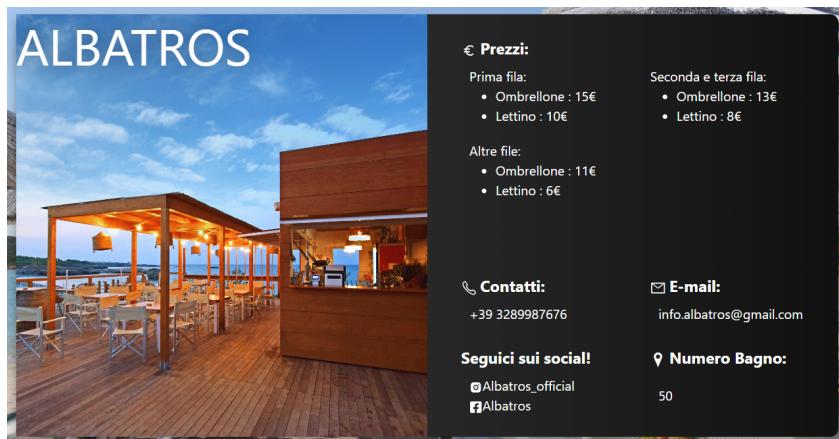


- **Postazioni balneari;** ogni flexitem rappresenta una specifica postazione, a cui corrispondono due coordinate: la prima, relativa alla fila, e la seconda, relativa al numero dell'ombrellone.

0.3.3 CardsPage

Questa interfaccia racchiude le diverse cards, ciascuna rappresentativa e descrittiva di uno specifico stabilimento balneare. Essa è costituita da un box centrale, nowrap, costituito da due porzioni:

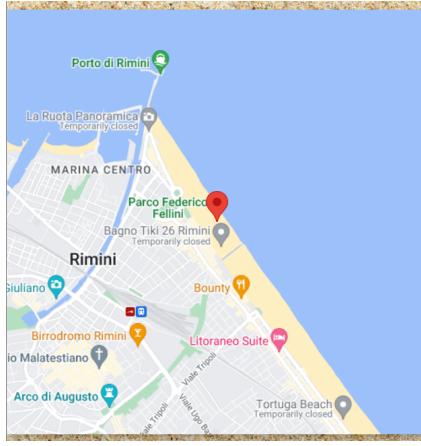
- L'immagine e il nome dello stabilimento, sulla sinistra
- Le informazioni relative al corrispondente stabilimento, sulla destra.



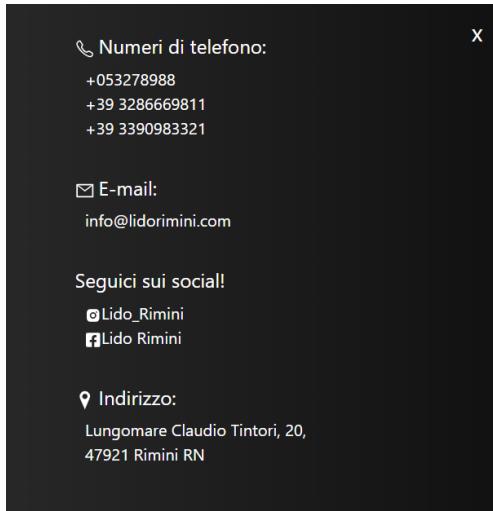
Per lo scorrimento delle diverse cards, sull'interfaccia sono presenti i corrispondenti **Bottoni a scorrimento**.

0.3.4 PlacePage

Questa interfaccia è costituita da un box centrale suddiviso in due sezioni:



- **Mappa;** descritta dal component **GoogleMap**, ritorna una porzione di mappa e un Marker che fissa una specifica posizione localizzata su Google Maps.



- **Contatti,** insieme di item che racchiudono le informazioni di contatto del lido.

0.4 Tecnologie

L'applicazione Web è stata realizzata basandosi sull'architettura MERN.
MERN è l'acronimo di:

- **MongoDB** - database documentale (NoSQL)
- **Express** - web framework di Node.js
- **React** - client-side Javascript framework
- **Node** - piattaforma Javascript server-side

MERN è un full-stack Javascript framework ottimizzato che permette di velocizzare e rendere più semplice la creazione di applicazioni Web robuste e facilmente mantenibili.

Il livello superiore dello stack MERN è occupato da **React.js**, framework Javascript dichiarativo per la creazione di applicazioni dinamiche lato client in HTML.

React è una libreria Javascript per costruire web application, che separa la UI in elementi componibili e che lavora a frontend. E' stato scelto per i diversi vantaggi che offre, tra i quali:

- Componenti Riusabili. React permette la separazione degli elementi UI in componenti riusabili e facilmente testabili.
- Alte Performance garantite dal Virtual Dom grazie al quale, React, riesce ad ottimizzare enormemente le performance sul web, diminuendo drasticamente gli aggiornamenti diretti sul DOM, in quanto agisce prima sul DOM virtuale e poi, con una serie di confronti tra lo stato precedente e quello corrente, calcola il modo migliore per apportare le modifiche.
- Chiara separazione del codice in base alle diverse funzioni.
- Gestione di interfacce data-driven, con stato, che richiedono una minima quantità di codice a bassissimi problemi; ottimo supporto per moduli, gestione degli errori, eventi, elenchi e altro.

Il vantaggio principale che React.js offre è la scomposizione in componenti funzionali che favorisce la modularità del codice. Infatti, per ogni componente HTML che si vuole renderizzare, è possibile definire il corrispondente componente, creato da una rispettiva funzione. React, inoltre, si basa su Flux (alternativa al pattern MVC), che supporta il concetto del data flow unidirezionale; le action sono inviate ad uno store attraverso un dispatcher, le modifiche allo store sono propagate fino alla view.

Il livello successivo dello stack MERN è occupato dal framework lato server **Express.js**, in esecuzione all'interno di un ambiente **Node.js**.

Express.js è un web application framework, minimale e flessibile. Fornisce un insieme di feature per costruire web app facilitando l'uso di Node.js e l'implementazione di API REST. Inoltre, supporta diversi template engine, senza oscurare mai le funzionalità di Node.js.

Express.js è, dunque, un framework server-side, in esecuzione all'interno di un server Node.js. Esso consiste in un framework web veloce, con potenti modelli per il routing degli URL (ricercando il match tra un URL in entrata e la corrispondente funzione del server) e per la gestione delle richieste e delle risposte HTTP.

Effettuando delle richieste HTTP XML, o GETs o POSTs dal front-end React.js, ci si può connettere alle funzioni Express.js che alimentano l'applicazione. Queste funzioni, a loro volta, usano i driver Node.js di MongoDB, attraverso callbacks o promises, per accedere e aggiornare i dati nel database MongoDB. Tra le funzionalità principali che offre Express:

- Consente di definire middleware per rispondere a richieste HTTP
- Consente di definire una tabella di routing per eseguire azioni differenti in base al Metodo HTTP e all' URL.

Node.js, invece, è una piattaforma software cross-platform che permette di creare il proprio Web server e su cui si possono creare Web application. Alcune sue caratteristiche sono:

- esegue codice javascript server side
- è single-threaded
- possiede una event driven architecture
- ha un'esecuzione asincrona
- è un non blocking I/O model

Node.js permette di avere Web application con connessioni real-time, two-way, dove anche il server può iniziare la connessione, permettendo di trasferire liberamente i dati. Tra i benefici che offre si evidenziano:

- Permette di scrivere codice in Javascript
- Estremamente veloce e leggero
- Uso efficiente delle risorse
- Non c'è bisogno di eseguire un web server separato
- Permette di avere un forte controllo della logica dell'app e dell'ambiente
- Permette di gestire diversi utenti con poche risorse

MongoDB è un database NON relazionale, classificato come NoSql database. MongoDB è un database cross-platform, orientato ai documenti, json-like document con schema dinamico. La scelta del database segue l'utilizzo di framework come React, Express e Node, con cui risulta più semplice lavorare se si utilizzano database non relazionali.

I documenti JSON creati a front-end con React.js, possono essere inviati al server Express.js, dove possono essere elaborati e (se validi) archiviati direttamente in MongoDB per un successivo recupero.

0.5 Codice

L'applicazione si sviluppa su due lati: la componente client side e la componente server side.

0.5.1 Server Side Components

Il software lato server si occupa principalmente di gestire il reindirizzamento delle pagine web e dell'interazione con il database.

Esso è costituito da un insieme di routes, controllers e models.

Il **routing** è una parte fondamentale di una qualunque applicazione web. Grazie ad esso è possibile manovrare un'applicazione in esecuzione attraverso la richiesta di specifici path nell'url. Con il server-side routing, un utente fa click su un link, il quale richiede una nuova pagina o nuovi dati dal server e questi nuovi dati vengono serviti da esso all'utente.

```
app.route('/api/bookingsAlbatros/:id')
    .delete(bookingsController.delete_Albatros);

app.route('/api/bookingsHakunaMatata')
    .get(bookingsController.list_bookings_hakunamatata)
    .post(bookingsController.create_booking_hakunamatata);
```

Come si può vedere dall'immagine, grazie a questa suddivisione, il sistema di routing collega direttamente uno specifico link alla corrispondente funzione richiesta, gestita dal Controller.

Il **Controller** racchiude l'insieme delle funzioni che vengono eseguite lato server e consistono nella ricezione di una richiesta lato client, elaborazione di una risposta ed, infine, restituzione di quest'ultima al client. La struttura base di queste funzioni, infatti, prevede la ricezione di una richiesta, lato client, l'elaborazione della richiesta nel corpo della funzione, e la restituzione di un risultato. Un esempio può essere:

```
exports.delete_Albatros = function(req, res) {
    BookingAlbatros.deleteOne({_id: req.params.id}, function(err, result) {
        if (err)
            res.send(err);
        else{
            if(result.deletedCount==0){
                res.status(404).send({
                    description: 'Booking not found'
                });
            }
            else{
                res.json({ message: 'Task successfully deleted' });
            }
        }
    });
}
```

La funzione riportata nell'immagine si occupa di ricevere una richiesta di eliminazione di una prenotazione, da parte di un utente. Dal corpo della richiesta viene estratto l'ID della prenotazione e, se presente nel database, viene eliminata la corrispondente tupla, viceversa, se non presente, viene restituito un messaggio di errore 404.

Ogni funzione lavora su un modello di dati rappresentativo delle tuple contenute all'interno del corrispondente database. L'insieme di questi modelli è racchiuso nella directory models.

Un **modello** è uno schema rappresentativo del corrispondente database a cui fa

riferimento.

```
module.exports = function(mongoose) {
  var Schema = mongoose.Schema;
  var BookingAlbatrosSchema = new Schema({
    dataInizio: Date,
    dataFine: Date,
    fila: Number,
    postazione: Number,
  });
  return mongoose.model('BookingAlbatros', BookingAlbatrosSchema),
};
```

L'immagine rappresenta il modello Albatros: ossia lo schema del database "bookingalbatros".

0.5.2 Client Side Components

Il software lato client si occupa di gestire interamente l'interazione con l'utente e quindi l'interfaccia lato client.

Grazie alla forte modularizzazione di React, il software client side si scomponete in 3 subdirectory principali: components, data e pages.

Il file che viene lanciato a momento di esecuzione è index.js, il quale, grazie all'utilizzo del render di ReactDOM, richiede l'esecuzione del component App. **App.js** è un component React che racchiude l'insieme delle routes richieste dall'applicazione e il component "Navbar".

```
<div class="row-container">
  <Navbar/>
  <Route path="/" exact><HomePage/></Route>
  <Route path="/booking"><ReservationPage/></Route>
  <Route path="/booking/Albatros"><AlbatrosPage/></Route>
  <Route path="/booking/HakunaMatata"><HakunaMatataPage/></Route>
  <Route path="/booking/Faro"><FaroPage/></Route>
  <Route path="/booking/Marrakech"><MarrakechPage/></Route>
  <Route path="/booking/Granchio"><GranchioPage/></Route>
  <Route path="/booking/FloridaBeach"><FloridaBeachPage/></Route>
  <Route path="/booking/Oasi"><OasiPage/></Route>
  <Route path="/booking/Mexico"><MexicoPage/></Route>
  <Route path="/schede"><CardsPage /></Route>
  <Route path="/Luogo"><PlacePage/></Route>
</div>
```

- **Navbar** è la barra di navigazione, costituita da un menu a tendina che dirige l'utente nelle diverse pagine di navigazione fornite dall'applicazione.
- Tutte le **routes** presenti collegano uno specifico path alla rispettiva pagina. Ogni pagina corrisponde ad un component presente nella subdirectory pages.

La subdirectory **pages** è l'insieme tutti i components Pages che, semplicemente, renderizzano alla rispettiva pagina da visualizzare.

La struttura del file è standard e semplice, ma si differenzia per alcuni aspetti.

- **HomePage**

```
function HomePage() {
  return (
    <Home slides={HomeData}/>
  )
}
```

Al component **Home**, che rappresenta la prima pagina che viene visualizzata all'avvio dell'applicazione, viene passata la lista **HomeData**. Quest'ultima raccoglie tutte le informazioni necessarie al caricamento della homepage e si struttura come segue:

```
export const HomeData = [
  {
    title:name,
    path:'/booking',
    image: ImageOne,
    label:label
  },
  {
    title:name,
    path:'/booking',
    image: ImageTwo,
    label:label
  },
  {
    title:name,
    path:'/booking',
    image: ImageThree,
    label:label
  }
];
```

Il file è contenuto nella subdirectory data.

- **ReservationPage** ritorna unicamente il component "Reservation" che rappresenta a sua volta la pagina a cui si accede dal menu principale; essa permetterà, poi, di selezionare il bagno desiderato.

- **Pages dei relativi bagni.** Sono 8, una per ogni stabilimento balneare. Ogni pages ritorna il component **General** a cui viene passato il vettore di dati della rispettiva pagina. Il vettore viene prelevato dalla lista **Cards-Data** da cui viene estrapolato unicamente l'insieme dei dati di interesse.

```
let data = [];

useEffect(()=> {
  let i = 0;
  while ((CardsData[i].title != "Albatros")&&(i<CardsData.length)) {
    i++;
  }
  data.push(CardsData[i]);
});

return (
  <General title={"Albatros"} data={data}/>
)
}
```

CardsData è un file che contiene una lista di dati e raccoglie tutte le informazioni riguardanti i singoli stabilimenti balneari. Un elemento si struttura come segue:

```
export const CardsData = [
  {
    title: "Albatros",
    path: '/booking/Albatros',
    label: label,
    image: Albatros,
    telefono: "+39 3289987676",
    email: "info.albatros@gmail.com",
    instagram: "Albatros_official",
    facebook: "Albatros",
    numero: 50,
    prima_ombrellone: "15€",
    prima_lettino: "10€",
    sec_terza_ombrellone: "13€",
    sec_terza_lettino: "8€",
    altri_lettino: "6€",
    altri_ombrellone: "11€"
  },
]
```

Il component **General** rappresenta la base comune di tutti gli stabilimenti balneari e si differenzia unicamente per il vettore di dati che gli viene passato a momento di caricamento. Esso si struttura come segue:

```
<Button onClick={openModal}>Scegli il periodo</Button>

<Calendar showModal={showModal} setShowModal={setShowModal} requestedDates={requestedDates} showBooking={showBooking} setShowBooking={setShowBooking} setDelete={setDelete} onDelete={onDelete}>
  {successDelete ? <Popup text={text} esito={esito} closePopup={() => setSuccessDelete(false)} /> : null}
</Calendar>

<GlobalStyle />

<Table bookings={loadedBookings} requestedDates={requestedDates} showBooking={showBooking} setShowBooking={setShowBooking} setDelete={setDelete} onDelete={onDelete}>
  {success ? <Popup text={text} esito={esito} closePopup={() => setSuccess(true)} /> : null}
</Table>
```

Utilizzando lo useState di React è possibile istanziare delle costanti da settare in seguito al verificarsi di certi eventi. Al momento di caricamento della pagina web, la costante showModal è impostata a false. In seguito al verificarsi dell'evento OnClick del Button di scelta del periodo di interesse per effettuare la prenotazione, la costante viene settata a true.

```
const openModal = () => {
  setShowModal( value: prev => !prev);
};
```

Conseguentemente a questo, viene allora mostrato il component Calendar che permette la selezione delle date richieste.

Grazie all'uso di animazioni viene visualizzato un flexbox suddiviso in due parti contenenti il DateRange così organizzato:

```

<DateRange
    style={{position:"initial"}}
    format="DD-MM-YYYY"
    minDate={new Date()}
    editableDateInputs={true}
    moveRangeOnFirstSelection={false}
    ranges={date}
    onChange={item => {
        setDate( value: [item.selection]);
        console.log(item);
    }}
    startDate={date.startDate}
    endDate = {date.endDate}
/>

```

che cattura in input le date selezionate dall'utente.

Una volta avvenuta la selezione, il component Table che prima non permetteva la selezione delle postazioni balneari, ora viene ricaricato ed, essendo la costante requestedDates diversa dal vettore nullo, mostra ogni postazione incrociando le disponibilità per il dato periodo.

Le disponibilità vengono mostrate in seguito al caricamento dal database di riferimento per il dato stabilimento balneare.

```

useEffect( effect: ()=>{
    axios.get( url: 'http://localhost:3000/api/bookings'+title)
        .then(response =>{
            let bookings= [];
            bookings = response.data
            setIsLoading( value: false);
            setLoadedBookings(bookings);
        })
}, deps: []);

```

Una postazione risulta non disponibile se:

```

for(let z=0; z<bookings.length;z++){
    if( ((requestedDates[0]>=bookings[z].dataInizio) && (requestedDates[0] <=bookings[z].dataFine)) ||
        ((requestedDates[0]<=bookings[z].dataInizio) && (requestedDates[1] <=bookings[z].dataFine) &&
        requestedDates[1] >=bookings[z].dataInizio) ||
        ((requestedDates[0]<=bookings[z].dataInizio) && (requestedDates[1] >=bookings[z].dataFine))) {
            no_available.push(bookings[z]);
        }
}

```

In seguito alla selezione della postazione, e quindi al verificarsi dell' evento onClick in Table, viene eseguita la funzione "openBooking" che modifica il valore di "showBooking". Ques'ultima costante, se settata a true, permette la visualizzazione del component Booking.

Similmente al component Calendar, Booking è un flexbox che mostra i dati riepilogativi della richiesta di prenotazione.

Confermando con l'apposito pulsante, viene eseguita la funzione **On-Added** che richiede l'inserimento di una tupla nel corrispondente database.

```

function OnAdded(booking) {
    setIsLoading( value: true)
    axios.post( url: "http://localhost:3000/api/bookings"+title, booking)
        .then( response => {
            //console.log(response.data);
            const newList = loadedBookings;
            newList.push(response.data)
            setLoadedBookings(newList);
            setIsLoading( value: false);
            setSuccess( value: prev => !prev);
            setEsito( value: true);
            setText( value: "Prenotazione avvenuta con successo!");
        }).catch(error => {
            console.log(error);
            setText( value: 'Qualcosa è andato storto. Riprova più tardi.');
        })
}

```

Un'altra interazione che l'utente può fare in questa pagina è **Cancelare una prenotazione**. Anche in questo caso, in seguito al verificarsi dell'evento onClick del Button **Cancella prenotazione**, viene eseguita la funzione **openDelete** che modifica il valore della costante showDelete e permette la visualizzazione del contenuto del component **Delete** che consiste in un flexbox in cui è possibile inserire l'ID della prenotazione che si richiede di eliminare. Confermando la richiesta, viene eseguita la funzione **OnDelete**, di seguito riportata:

```

function OnDelete(_id){
    axios.delete( url: 'http://localhost:3000//api/bookings'+title+'/${_id}')
        .then(response=>{
            if(response.status==404){
                console.log('Not Found');
                setText( value: 'Prenotazione non trovata!');
                setEsito( value: false);
            }
            else{
                console.log('Deleted');
                console.log(response.status);
                setEsito( value: true);
                setText( value: "Eliminazione avvenuta con successo!");
            }
        })
        .catch(error =>{
            console.log(error);
            setText( value: 'Not Found');

        })
        .setSuccessDelete( value: prev => !prev);
        console.log(_id);
}

```

- **CardsPage** ritorna il component Cards, a cui viene passato il vettore dei dati **CardsData**.

```

function CardsPage() {
    return (
        <Cards cards={CardsData}>
    )
}

```

Il component **Cards** altro non è che un box, nowrap, i cui item a scorrimento sono il risultato di una mappatura sul vettore CardsData.

```

<body style={{overflowX: "auto"}}>
    {cards.map((card, index) => {
        return (
            <>
                {index === current && (
                    <div className="contacts-container" key={index}>

```

Ogni item si suddivide in due parti:

- Il componente di sinistra: costituito da un'immagine, corrispondente allo stabilimento balneare, e dal rispettivo nome.
- Il componente di destra: che riporta tutti i dati relativi allo stabilimento mappando i corrispondenti valori al componente di riferimento nel vettore di dati.

- **PlacePage** riporta il component **Contatti**.

Anche in questo caso viene visualizzato un box, nowrap, composto da due sezioni:

- La sezione di sinistra visualizza il component GoogleMap.
GoogleMap utilizza il render di React e le classi offerte dalla libreria "google-maps-react", quali **Map** e **Marker**. Questo componente ritorna l'immagine offerta da google maps relativa alla posizione del lido. Esso lavora come segue:

```

<Map
    google={this.props.google}
    initialCenter={{
        lat: this.state.mapCenter.lat,
        lng: this.state.mapCenter.lng
    }}
    center={{
        lat: this.state.mapCenter.lat,
        lng: this.state.mapCenter.lng
    }}
>
    <Marker
        position={{
            lat: this.state.mapCenter.lat,
            lng: this.state.mapCenter.lng
        }}
        text={{
            this.state.text = "Lido di Rimini"
        }}
    />
</Map>

```

0.6 Test

Questa fase della progettazione dell'applicazione web è parte integrante della progettazione stessa.

Il testing rappresenta uno degli aspetti più importanti all'interno del ciclo di vita di un software: permette di individuare gli errori durante la fase di sviluppo, aumenta la qualità del software e quindi la sicurezza e la fiducia da parte dei cliente e degli stakeholder verso l'azienda, è necessario per applicazioni di alta qualità che vogliono mantenere un basso costo di manutenzione ed è fondamentale per riuscire a mantenersi sul mercato. La difficoltà di individuazione e il costo di risoluzione di un bug aumentano notevolmente con l'avanzare del ciclo di vita del software. Per questo motivo il testing dovrebbe essere effettuato a partire dalle prime fasi di produzione del codice.

Il testing è in ogni caso da considerare non come una singola attività ma come un processo che si articola durante tutto il ciclo di vita del software.

I test che sono stati effettuati si possono suddividere per tipologia:

- **Graphic User Interface Test.**

I Graphic User Interface (GUI) sono quei test che vengono effettuati sull'aspetto estetico del programma. Questi tipi di test verificano che tutti i tasti, le caselle di testo, le immagini, le icone delle interfacce utente vengano visualizzati in maniera corretta e permettano all'applicazione di soddisfare le sue specifiche.

Questo genere di test è principalmente manuale; sono state eseguite una serie di operazioni sull'applicazione ed è stato controllato visivamente se

il risultato ottenuto fosse quello desiderato. Tra i test che sono stati effettuati ritroviamo:

- Verifica della flessibilità dell’interfaccia in seguito al ridimensionamento della finestra di visualizzazione del browser e quindi al suo adattamento. Si è verificato, dunque, il funzionamento dei diversi flexbox.
- Verifica del funzionamento corretto di tutti i button presenti. Ognuno di essi deve riportare correttamente la modifica dell’interfaccia, come preventivamente.
- Verifica del corretto funzionamento dei diversi popup in seguito alla prenotazione o cancellazione avvenuta con successo e, viceversa, ai messaggi di errore.
- Grazie all’interrogazione di alcuni utenti campionari, si è verificato che l’obiettivo di tutte le immagini presenti nelle diverse pagine dell’applicazione venisse colto secondo il significato preventivato. Ad esempio, l’immagine di sfondo nella schermata riservata alle prenotazioni delle postazioni balneari ha l’obiettivo di suggerire all’utente l’ordine delle file rispetto alla riva, quindi di indicargli quale tra queste corrisponde alla prima fila e rispettivamente, all’ultima.

• **Integration Test.**

I test di integrazione vengono effettuati per verificare che, dopo aver unito diversi moduli di un programma, questo funzioni correttamente e non ci siano conflitti tra i suoi componenti. Il test di integrazione è sicuramente uno dei più completi e sicuri, perché verifica una serie di passaggi consecutivi che non riguardano solamente l’aspetto visivo dell’applicazione ma anche quello funzionale.

A tale scopo, partendo dal primo file che viene lanciato a momento di esecuzione (index.js) si è testato che il Render ritornasse, nell’ordine corretto, tutte le Routes definite nel file e che tra una pagina e l’altra si seguisse correttamente il flusso dei files. In altre parole, si è verificato che i file mostrati secondo ogni diverso percorso di navigazione rispettasse l’architettura dell’applicazione definita durante la fase di raccolta dei requisiti.

• **Database Test.**

I test sul database hanno l’obiettivo di verificare il corretto funzionamento dell’interazione con quest’ultimo per tutte le operazioni a lui richieste.

A tale scopo, è stato utilizzato un software di supporto: **Postman**.

Postman è una piattaforma API che consente agli sviluppatori di progettare, costruire, testare e iterare le proprie API. Nello spazio di lavoro principale, è possibile testare diverse tipologie di richieste HTTP, quali: **GET, POST, COPY, DELETE, etc.** Inoltre, è necessario impostare correttamente anche il corrispondente URL di riferimento.

- GET request.

The screenshot shows a POSTMAN interface with a 'GET' method selected. The URL is set to `http://localhost:3000/api/bookingsAlbatros`. The 'Body' tab is highlighted, showing a JSON payload:

```
1
2   - "dataInizio": "2023-01-30",
3   - "dataFine": "2023-01-31",
4   - "fila": "6",
5   - "postazione": "6"
6
```

Questa richiesta HTTP testa la connessione tra il server e il database e, se funzionante, restituisce l'insieme delle tuple contenute nel database di riferimento.

- POST request.

The screenshot shows a POSTMAN interface with a 'POST' method selected. The URL is set to `http://localhost:3000/api/bookingsAlbatros`. The 'Body' tab is highlighted, showing a JSON payload:

```
1
2   - "dataInizio": "2023-01-30",
3   - "dataFine": "2023-01-31",
4   - "fila": "6",
5   - "postazione": "6"
6
```

Questa richiesta HTTP testa, invece, la connessione tra il server e il database sulla riuscita della richiesta POST. L'obiettivo di questo test è l'inserimento di una tupla all'interno del database di riferimento. La tupla può essere definita secondo diversi formati, quello riportato è di tipo JSON.

- DELETE request.

The screenshot shows a POSTMAN interface with a 'DELETE' method selected. The URL is set to `http://localhost:3000/api/bookingsAlbatros/:id`. The 'Params' tab is highlighted, showing a 'Query Params' section with a 'Key' column and a 'Value' column. Below it is a 'Path Variables' section with a 'KEY' column and a 'VALUE' column.

| KEY | VALUE |
|-----|-------|
| Key | Value |

| KEY | VALUE |
|-----|--------------------------|
| id | 63f32ac47579174130d19587 |

Questa richiesta HTTP testa la connessione tra il server e il database sulla riuscita della richiesta DELETE. L'obiettivo di questo test è l'eliminazione di una specifica tupla all'interno del database di riferimento. A tale scopo, deve essere perciò indicato l'identificativo della tupla, come mostrato nell'immagine.

0.7 Deployment

Una volta realizzata e ultimata l'applicazione è possibile passare all'ultima fase di rilascio, installazione e messa in funzione.

A tale scopo, è stato utilizzato un software di supporto chiamato **Docker**.

Docker è un progetto open source per automatizzare la distribuzione di app come contenitori portabili e autosufficienti che possono essere eseguiti nel cloud o in locale.

Docker è, quindi, un popolare software libero, progettato per eseguire processi informatici in ambienti isolabili, minimali e facilmente distribuibili chiamati **container**, con l'obiettivo di semplificare i processi di deployment di applicazioni software.

L'idea alla base è quella di costruire diversi container che, insieme, descrivono l'applicazione in maniera isolata rispetto alla macchina fisica che li ospita; l'obiettivo è quello di costruire all'interno dei container, uno scenario che sia isolato/separato dalla macchina fisica in cui risiede l'applicazione, al fine di ovviare ai problemi di collisione che si potrebbero riscontrare con le applicazioni, librerie e software generale.

Per questa applicazione, i container necessari sono tre:

- frontend
- backend
- mongodb

Il container **frontend** contiene il servizio basato su React. Questo componente racchiude il software necessario alla visualizzazione dell'applicazione e all'elaborazione e invio delle richieste http al server.

Il container **backend** contiene il servizio basato su Node.js ed Express.js. Questo componente, invece, racchiude il software necessario alla ricezione di una richiesta da parte del componente precedente, alla sua elaborazione e invio di una specifica risposta http al client.

Infine, il container **mongodb** racchiude il servizio basato sul software MongoDB. Esso permette l'archiviazione e l'interazione con i database necessari al corretto funzionamento dell'applicazione.

Questi container espongono il loro servizio su specifiche porte tcp:

- frontend su porta 3000
- backend su porta 3000
- mongodb su porta 27017

Come sintetizzato nel file **compose** presente nella root del progetto, per permettere la comunicazione tra i tre container è indispensabile il mappaggio tra gli stessi in modo tale da renderli visibili nelle rispettive porte tcp.

0.8 Conclusioni

Dopo aver eseguito la pacchettizzazione dell'applicativo con dentro le immagini dei container, è possibile eseguire ed utilizzare in ambiente isolato la piattaforma per la gestione delle spiege del Lido di Rimini, comprensiva dei suoi

8 stabilimenti balneari. L'applicazione permette, quindi, la prenotazione di una postazione mare, previa disponibilità dello specifico bagno, ed altre visualizzazioni base, quali: le schede tecniche descrittive dei diversi stabilimenti, la localizzazione dei bagni e la pagina di riferimento dei contatti.

Una possibile estensione dell'applicativo potrebbe includere:

- Gestione Utenti
- Possibilità di selezionare numero di lettini desiderato per postazione
- Gestione dei pagamenti online

0.9 Bibliografia e sitografia

- <https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04>
- <https://react-icons.github.io/react-icons/>
- <https://www.mongodb.com/mern-stack>
- <https://www.youtube.com/@javascriptmastery>