

UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

Progetto WebRTC

GoodChat

A.A 2021/2022

Buono Andrea M63001193

Sommario

1 Introduzione	3
1.1 Strumenti utilizzati	3
1.2 Diagramma dei casi d'uso.....	4
2 Backend	5
2.1 Setup dell'ambiente	5
2.2 Implementazione.....	6
2.2.1 Server.js	6
2.2.2 Model.....	7
2.2.3 Controller.....	9
2.2.3 Route	10
2.3 Testing	10
3 Front-end.....	12
3.1 Setup dell'ambiente	12
3.2 Implementazione.....	12
3.2.1 DOM.....	12
3.2.2 Hook.....	13
3.2.3 React Router	14
3.2.4 Redux.....	15
3.2.5 Cloudinary.....	17
4 Realizzazione chat	19
4.1 Client.....	19
4.2 Server.....	20
5 Demo.....	22

1 Introduzione

Questo documento andrà a presentare GoodChat, una web application derivata dal progetto open source WebRTC, il quale permette la comunicazione real time da browser web attraverso l'utilizzo di API.

GoodChat è un'applicazione di messagistica istantanea che, previa registrazione, offre comunicazioni private tra due utenti e anche stanze pubbliche con cui comunicare con tutti gli utenti della piattaforma.

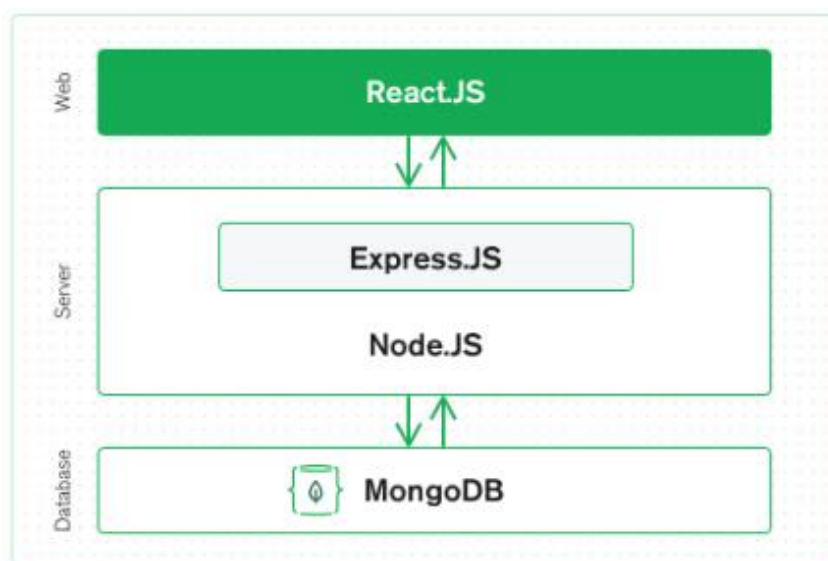
Nei prossimi capitoli verranno mostrati nel dettaglio strumenti e metodologie utilizzati per la realizzazione dell'applicazione. In particolare, verrà mostrata prima la parte relativa al backend e successivamente la parte relativa al frontend.

Prima di fare ciò, si riserva un paragrafo per mostrare quali sono gli strumenti utilizzati sia per il backend che per il frontend.

1.1 Strumenti utilizzati

L'applicazione è stata realizzata utilizzando lo stack MERN¹, ovvero sfruttando:

- **MongoDB**, un database distribuito e non relazionale. Utilizzato per l'archiviazione dei dati.
- **Express.JS**, un framework lato server basato su JavaScript. Utilizzato come strato middleware e fornire API di routing.
- **React.JS**, una libreria JavaScript utilizzata per realizzare le interfacce necessarie.
- **Node.JS**, un framework che permette la realizzazione di applicazioni web in JavaScript.

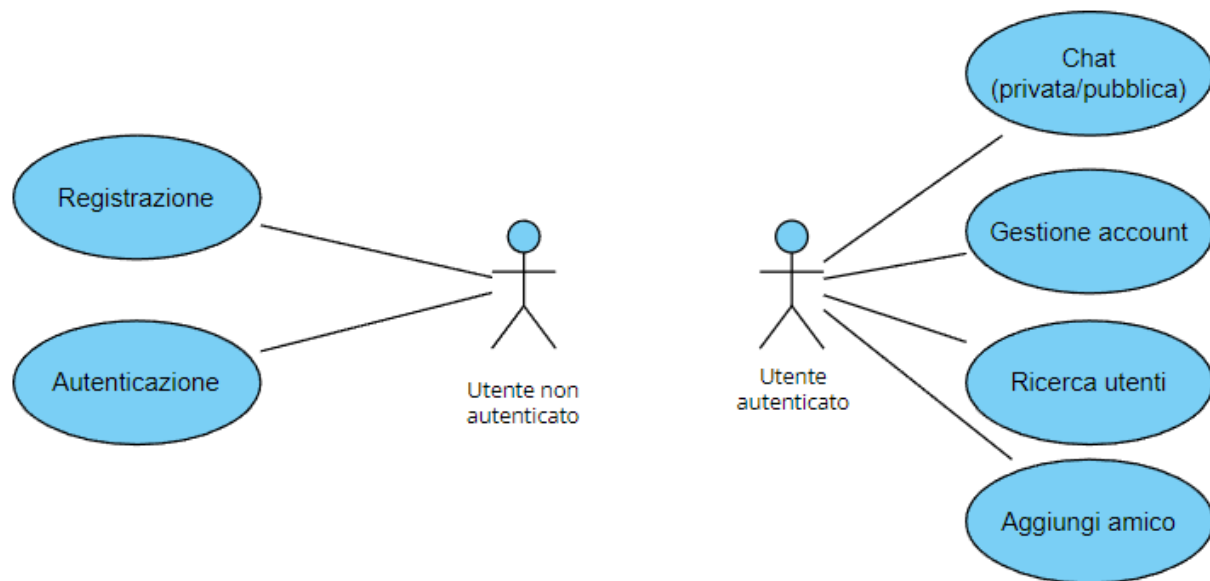


¹ Una variante dello stack MEAN, dove al posto di ReactJS si utilizza Angular

Il gestore di pacchetti utilizzato per l'installazione dei moduli necessari è *npm* , sebbene probabilmente una scelta migliore sarebbe stata l'utilizzo di *yarn*. Infatti, quest'ultimo fu sviluppato dal team di Facebook proprio per risolvere alcuni noti problemi di *npm*.

1.2 Diagramma dei casi d'uso

Si riporta un semplice diagramma dei casi d'uso che mostra ad alto livello le funzionalità offerte dall'applicazione.



Per 'Gestione account' si intende sia la modifica di dati personali, come nome o password, sia la modifica dell'avatar, ovvero una foto associata all'utente che gli altri utenti potranno vedere.

Una volta registrati, non sarà più possibile modificare l'email.

2 Backend

Una volta avviato l'ambiente di esecuzione (in questo caso il server Node.js), il server sarà raggiungibile in locale sul numero di porta 3000, quindi digitando nella barra degli indirizzi di qualsiasi web browser² : <http://localhost:3000>.

2.1 Setup dell'ambiente

Nella directory relativa al lato backend del progetto si lancia il comando:

```
npm init
```

Questo va a generare il file `package.json`, contenente vari metadati e necessario per gestire le dipendenze dell'applicazione. Inoltre, in questo file, è possibile andare ad inserire degli script che possono essere lanciati da terminale per facilitare le operazioni.

Ogni qual volta si è rivelato necessario andare ad aggiungere nuove dipendenze (ad esempio per aggiungere mongoose per la connessione a MongoDB), questo è stato fatto utilizzando da terminale il comando:

```
npm install dipendenza_necessaria
```

In questo modo verrà generato anche il file `package-lock.json`, nel quale si troverà un elenco di tutte le dipendenze installate con le loro specifiche versioni.

La differenza tra i due file è che `package.json` conterrà le dipendenze richieste dall'applicazione (nel senso di requisiti minimi), mentre `package-lock.json` conterrà quelle effettivamente installate. Bisogna prestare particolare attenzione alla versione delle dipendenze poiché versioni diverse possono creare problemi di compatibilità.

Nel file `package.json` si è andato ad aggiungere tra gli script il comando *nodemon server.js*, che permette di avviare il server in modalità sviluppo, in modo tale che le modifiche siano subito visibili senza la necessità di riavviare il server. Lo script può essere lanciato con:

```
npm run dev
```

Un'altra parte fondamentale per la corretta esecuzione dell'ambiente è il setup di un particolare file che conterrà quelle che si possono definire variabili d'ambiente, ovvero il file `.env`. Oltre a questo dovrà anche essere regolato l'accesso al database nel file `connection.js` : si rimanda al paragrafo 2.2.2.

Tale file conterrà i dati di accesso (quindi anche la password segreta) al database MongoDB.

² Sono stati rilevati alcuni problemi (grafici) nella formattazione della pagina utilizzando Edge. Per un funzionamento totalmente corretto si consiglia l'uso di Opera o Chrome.

2.2 Implementazione

Poiché Node.js, su cui si basa l'applicazione, supporta il pattern architetturale Model-View-Controller, il back-end dell'applicazione è stato suddiviso in tre parti:

- Model, quindi le collezioni che saranno memorizzate sul database MongoDB.
- Controller, quindi le funzioni che permettono la modifica del Model (attraverso la View).
- Route, quindi i percorsi necessari a permettere che i client possano fare richiesta alle API.

Nei seguenti sottoparagrafi andremo ad analizzare come ciascuna di queste parti è stata effettivamente realizzata. Andremo inoltre ad analizzare il primo file che viene lanciato all'avvio del server, ovvero *server.js*.

2.2.1 Server.js

Server.js è il primo file ad essere eseguito (lanciando il comando speciale definito in precedenza) e mette il server Express in ascolto sulla porta 5001.

È questo il file che si occupa dell'avvio e dell'importazione di moduli e funzioni necessari all'esecuzione dell'applicazione.

All'interno del file sono inoltre definite alcune funzioni che si rendono necessarie una volta che l'utente ha effettuato il login ed è pronto ad iniziare delle chat con altri utenti. In particolare, alcuni eventi che coinvolgono le socket sono gestiti qui ed è compito di questo file anche recuperare le vecchie conversazioni e ordinare i messaggi in base alla data.

A titolo di esempio vediamo una parte del codice relativa alle importazioni necessarie e alla creazione del server:

```
const express = require('express');
const app = express();
const cors = require('cors');

app.use(express.urlencoded({extended: true}));
app.use(express.json());
app.use(cors());

const server = require('http').createServer(app);
const PORT = 5001;
const io = require('socket.io')(server, {
  cors: {
    origin: 'http://localhost:3000',
    methods: ['GET', 'POST']
  }
})
```

2.2.2 Model

Per realizzare il model è stato creato un database sul cloud offerto da MongoDB.

Per poter accedere a tale database è stato quindi necessario creare un file apposito, *connection.js*, nel quale si va a sfruttare Mongoose, una libreria di programmazione JavaScript che crea una connessione tra MongoDB e l'ambiente di runtime Node.js.

Mostriamo il codice per intero:

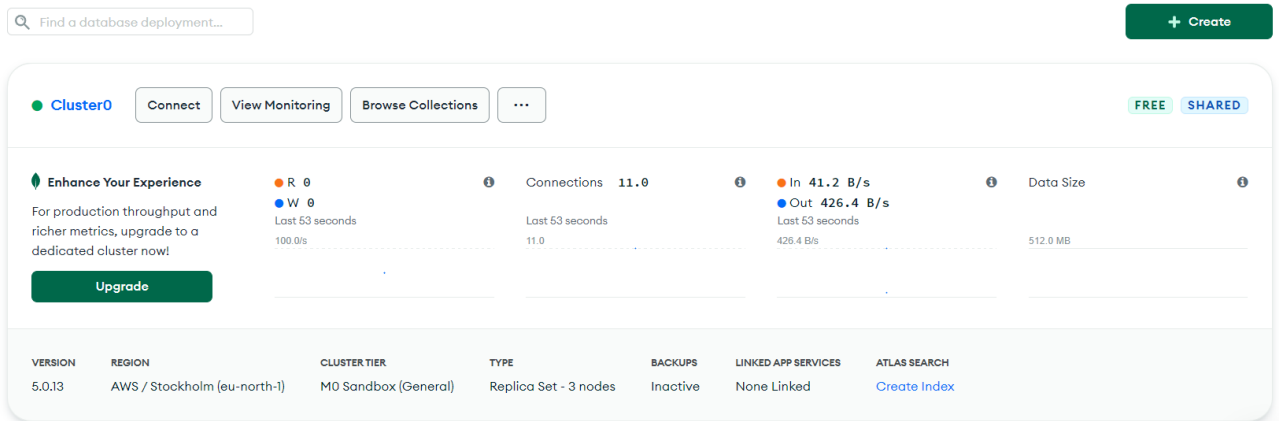
```
const mongoose = require('mongoose');
require('dotenv').config();

mongoose.connect(`mongodb+srv://${process.env.DB_USER}:${process.env.DB_PW}@cluster0.xxxx.mongodb.net/?retryWrites=true&w=majority`, () => {
  console.log('connected to mongodb')
})
```

L'argomento della funzione connect è una stringa indicata direttamente da MongoDB in seguito al click su "Connect", mostrato in figura successiva. Si noti come nella stringa siano presenti i due parametri che sono stati, per motivi di sicurezza, inseriti all'interno del file .env.

ANDREA'S ORG - 2022-07-22 > PROJECT 0

Database Deployments



Come si può notare in figura, è stato quindi creato un cluster nominato Cluster0 all'interno del quale è stato creato un database nominato mern-chat.

Su questo database sono state descritte due collezioni: User, che conterrà appunto gli utenti iscritti alla webapp, e Message, che conterrà i messaggi scambiati dagli utenti.

Mostriamo quindi prima lo schema della collezione User:

```
const UserSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Can't be blank"]
  },
  email: {
    type: String,
    lowercase: true,
    unique: true,
    required: [true, "Can't be blank"],
    index: true,
    validate: [isEmail, "invalid email"]
  },
  password: {
    type: String,
    required: [true, "Can't be blank"]
  },
  picture: {
    type: String,
  },
  newMessages: {
    type: Object,
    default: {}
  },
  status: {
    type: String,
    default: 'online'
  },
  friends: {
    type: Array,
    default: []
  }
}, {minimize: false});

const User = mongoose.model('User', UserSchema);

module.exports = User
```


E poi quello relativo a Message:

```
const MessageSchema = new mongoose.Schema({
  content: String,
  from: Object,
  time: String,
  date: String,
  to: String
})

const Message = mongoose.model('Message', MessageSchema);

module.exports = Message
```

2.2.3 Controller

La parte Controller contiene quelle funzioni che permettono di modificare il Model a seguito di un input dal front-end.

In JavaScript queste funzioni sono realizzabili attraverso due speciali keyword:

- *async*, utilizzata nella definizione della funzione poiché l'interazione con il database è un'operazione asincrona;
- *await*, valida esclusivamente all'interno delle funzioni asincrone. Mette la funzione in attesa che l'operazione asincrona venga risolta e venga restituito un risultato (detto *Promise*). È utile quando si vuole che il codice sincrono attenda quello asincrono.

I parametri da fornire ad una funzione asincrona sono tre:

- *req*, la richiesta inviata dal client;
- *res*, la risposta che sarà ritornata al client (in formato JSON);
- *next*, la funzione di callback. Viene richiamata nel caso in cui la Promise venisse rifiutata o se si verificasse un errore.

A titolo di esempio si riporta una funzione asincrona estratta dal codice, in particolare dal file *server.js*. Tale funzione svolge il ruolo di recuperare i messaggi contenuti in una particolare chat:

```
async function getLastMessagesFromRoom(room){
  let roomMessages = await Message.aggregate([
    {$match: {to: room}},
    {$group: {_id: '$date', messagesByDate: {$push: '$$ROOT'}}}]
  )
  return roomMessages;
}
```

2.2.3 Route

Questa parte si occupa di fornire al client un modo per poter fare richiesta alle API viste prima.

Express semplifica molto questo compito, poiché dà la possibilità di definire delle rotte e mette a disposizione metodi che vanno a creare degli oggetti Router. Questi oggetti sono necessari per la gestione delle richieste del client.

Una volta creato un oggetto Router, Express fornisce dei metodi per gestire le richieste HTTP. In particolare, con:

- `router.get(path, handler)`, si gestiscono le richieste HTTP GET;
- `router.post(path, handler)`, si gestiscono le richieste HTTP POST;

Vediamo come ad entrambi i metodi dovrà essere passato l'handler, ovvero la funzione che viene richiamata quando c'è una richiesta su quel path. Gli handler all'interno del codice sono spesso funzioni asincrone.

Le rotte sono definite all'interno del file `userRoutes.js`, contenuto in una apposita cartella `routes`.

Si riporta un esempio relativo alla funzione che il backend esegue quando un client fa richiesta di login:

```
router.post('/login', async(req, res)=> {
  try {
    const {email, password} = req.body;
    const user = await User.findByCredentials(email, password); //metodo del file
User.js
    user.status = 'online';
    await user.save();
    res.status(200).json(user);
  } catch (e) {
    res.status(400).json(e.message)
  }
})
```

2.3 Testing

Per il testing delle API è stato utilizzato *Postman*, uno strumento che permette appunto di testare le API comodamente.

Questo strumento permette di generare richieste HTTP verso le API attraverso una semplice interfaccia grafica e in seguito permette di visualizzare la risposta fornita dal server, in modo da poter subito identificare eventuali errori.

Vediamo un semplice esempio del testing della API che gestisce la modifica dei dati:

http://localhost:5001/users/update

Save

</>

POST

http://localhost:5001/users/update

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	name	demo			
<input checked="" type="checkbox"/>	email	demo@demo.it			
<input checked="" type="checkbox"/>	password	demo			
<input type="checkbox"/>	picture				
	Key	Value	Description		

Notiamo come sia possibile deselezionare i parametri che non dovranno essere inseriti all'interno della richiesta.

3 Front-end

In questo capitolo si andrà ad analizzare il front-end dell'applicazione.

Prima di passare ai dettagli implementativi, vediamo brevemente come è stata inizializzata la directory relativa a questa parte dell'applicazione.

3.1 Setup dell'ambiente

Avendo scelto di utilizzare lo stack MERN, il front-end è stato sviluppato grazie a React.js, una libreria JavaScript che permette la creazione di UI e lo sviluppo di applicazioni dinamiche che possano visualizzare i dati modificati senza avere l'obbligo di dover ricaricare la pagina.

Per poter iniziare lo sviluppo con React, nella cartella relativa al lato front-end, sono stati prima aggiunti i pacchetti necessari (con npm) e poi creato il progetto con il comando:

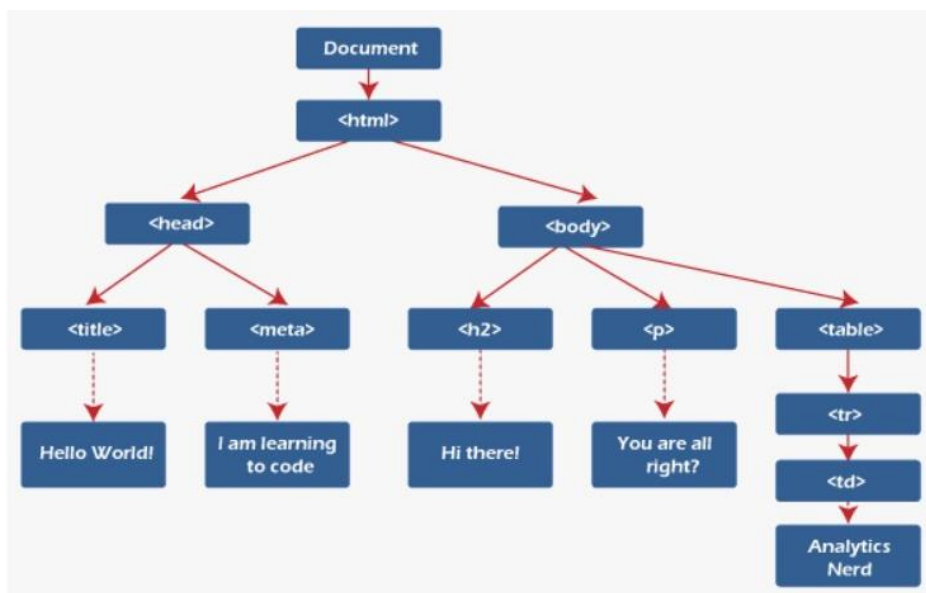
```
npx create-react-app mern-chat
```

3.2 Implementazione

3.2.1 DOM

L'approccio seguito da React per la realizzazione delle interfacce utente complesse è quello di scomporle in componenti più semplici, che verranno composti per poter arrivare al risultato finale.

React, inoltre, implementa un sistema DOM (Document Object Model) indipendente dal browser per massimizzare le prestazioni e la compatibilità con i vari browsers. Il DOM non è altro che una struttura ad albero attraverso la quale è descritta l'intera UI.



Dato l'oggetto che rappresenta il modello dei dati della pagina, per modificare gli elementi della pagina basta cambiare le informazioni all'interno del modello, rappresentato da uno stato. Per tenere traccia dello stato di un componente in React esiste l'apposito Hook

`useState()`). Questo perché tutto in React è visto come un componente e ogni componente ha uno stato.

Ogni volta che c'è una modifica, un Virtual DOM viene creato. Si cerca quindi un modo per attuare quelle modifiche sul DOM effettivo e gli elementi aggiornati verranno renderizzati sulla pagina. Potrebbe sembrare una perdita di tempo, ma in realtà calcolare le differenze tra il precedente stato e quello corrente è estremamente veloce e grazie al Virtual DOM si riducono gli interventi sul DOM reale (la cui modifica è più lenta).

La radice dell'albero del DOM è il file `index.js`, il quale, tolti i vari import, presenta il seguente codice.

```
const persistedStore = persistStore(store);

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistedStore}>
        <App />
      </PersistGate>
    </Provider>
  </React.StrictMode>,
  document.getElementById("root")
);
```

Vediamo come nel codice viene effettuato il rendering dell'elemento `App`, che verrà quindi aggiunto al DOM. Il componente `<Provider>` rende lo `store` Redux disponibile per tutti i componenti innestati che hanno bisogno di accedervi. Tutti i componenti React possono essere connessi allo store, per questo lo si mette nel livello più alto.

3.2.2 Hook

Gli Hook, citati in precedenza, sono delle funzioni integrate di React che permettono di utilizzare le funzionalità della libreria React (come lo stato e il contesto dei componenti) nei componenti funzionali senza doversi preoccupare di riscriverli in una classe.

Tali Hook (ad esempio lo `useState` e lo `useContext`) sono stati ampiamente utilizzati per la realizzazione del front-end. Sono anche stati definiti degli Hook personalizzati, scritti per realizzare determinate funzionalità dell'applicazione.

Vediamo un esempio estratto dal file `Sidebar.js`:

```
const [inputname, setInputName] = useState("");
const [inputname2, setInputName2] = useState("");
const { socket, setMembers, members, setCurrentRoom, setRooms,
privateMemberMsg, rooms, setPrivateMemberMsg, currentRoom } =
useContext(AppContext);
const [newFriend] = useAddFriendMutation();
```

3.2.3 React Router

Nel paragrafo 2.2.3 è stato illustrato come le API vengano realizzate nel back-end, ovvero come vengono descritte delle rotte grazie alle funzionalità offerte da Express.

Per poterle richiamare lato client ci si serve di React Router, una libreria che permette la transizione dinamica tra le varie pagine tramite JavaScript.

Sono forniti diversi tipi di Router, ma quello utilizzato per realizzare questa applicazione è il componente BrowserRouter. Quest'ultimo utilizza la History API di HTML5 per tenere la UI sincronizzata con l'URL.

Le varie rotte che dovrà utilizzare l'applicazione sono state definite nel file *App.js*, di cui si riporta il codice qui di seguito:

```
import './App.css';
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Navigation from './components/Navigation';
import Home from './pages/Home';
import Login from './pages/Login';
import Signup from './pages/Signup';
import Chat from './pages/Chat';
import Update from './pages/Update';
import PasswordReset from './pages/PasswordReset';
import { useSelector } from 'react-redux';
import { useState } from 'react';
import { AppContext, socket } from './context/appContext';

function App() {
  const [rooms, setRooms] = useState([]);
  const [currentRoom, setCurrentRoom] = useState([]);
  const [members, setMembers] = useState([]);
  const [messages, setMessages] = useState([]);
  const [privateMemberMsg, setPrivateMemberMsg] = useState({});
  const [newMessages, setNewMessages] = useState({});
  const user = useSelector((state) => state.user);
  return (
    <AppContext.Provider value={{ socket, currentRoom, setCurrentRoom, members,
setMembers, messages, setMessages, privateMemberMsg, setPrivateMemberMsg, rooms,
setRooms, newMessages, setNewMessages }}>
      <BrowserRouter>
        <Navigation />
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/login" element={<Login />} />
          <Route path="/signup" element={<Signup />} />
        </Routes>
      </BrowserRouter>
    </AppContext.Provider>
  );
}
```

```

        <Route path="/passwordReset" element={<PasswordReset
/>} />
      </>
    )}
    <Route path="/chat" element={<Chat />} />
    <Route path="/update" element={<Update />} />
  </Routes>
</BrowserRouter>
</AppContext.Provider>
);
}

export default App;

```

Il *Context* in React fornisce un modo per passare i dati attraverso l'albero dei componenti senza dover passare manualmente i *props* ad ogni livello.

3.2.4 Redux

Per raggiungere le API vere e proprie ci si è serviti di uno speciale toolkit utilizzato spesso insieme a React: *Redux.js*. Non è sempre indispensabile usarlo, ma torna utile quando lo stato dell'applicazione cambia di frequente.

Redux fornisce la funzione *createApi*, che permette di definire un set di endpoints che descrivono come recuperare i dati delle API del backend e come effettuare il fetching di quei dati.

Le Mutation, invece, sono usate in Redux per inviare messaggi di update contenenti dati al server ed eventualmente apportare modifiche alla cache locale.

Il codice relativo a questa parte è contenuto nel file *appApi.js*, di cui se ne riporta il codice per intero:

```

import { createApi, fetchBaseQuery } from "@reduxjs/toolkit/query/react";

const appApi = createApi({
  reducerPath: "appApi",
  baseQuery: fetchBaseQuery({
    baseUrl: "http://localhost:5001",
  }),

  endpoints: (builder) => ({
    // creazione dell'utente
    signupUser: builder.mutation({
      query: (user) => ({
        url: "/users",
        method: "POST",
        body: user,
      }),
    }),
  }),
});

```

```

    }},

    // login
    loginUser: builder.mutation({
      query: (user) => ({
        url: "/users/login",
        method: "POST",
        body: user,
      }),
    }),

    // logout

    logoutUser: builder.mutation({
      query: (payload) => ({
        url: "/logout",
        method: "DELETE",
        body: payload,
      }),
    }),

    addFriend: builder.mutation({
      query: (payload) => ({
        url: "/users/addfriend",
        method: "POST",
        body: payload,
      }),
    }),

    // modifica dati
    updateUser: builder.mutation({
      query: (user) => ({
        url: "/users/update",
        method: "POST",
        body: user,
      }),
    }),
  }},
});

export const { useSignupUserMutation, useLoginUserMutation, useLogoutUserMutation,
useUpdateUserMutation, useAddFriendMutation } = appApi;

export default appApi;

```

Notiamo come alla fine del codice si vanno ad esportare le Mutation appena definite, in modo da poterle poi richiamare.

3.2.5 Cloudinary

Un'ulteriore nota va fatta per spiegare perché si è scelto di utilizzare l'API offerta da Cloudinary.

Cloudinary è un servizio che permette di effettuare l'upload di risorse (in questo caso immagini) e di potervi accedere, previa autenticazione, attraverso una URL. Si è scelto di utilizzare questo servizio in modo tale che gli avatar potranno essere memorizzati su MongoDB semplicemente memorizzando questa URL.

Infatti, in fase di registrazione, all'utente è data la possibilità di scegliere un avatar, ovvero un'immagine che l'utente avrà previamente salvato sul proprio dispositivo.

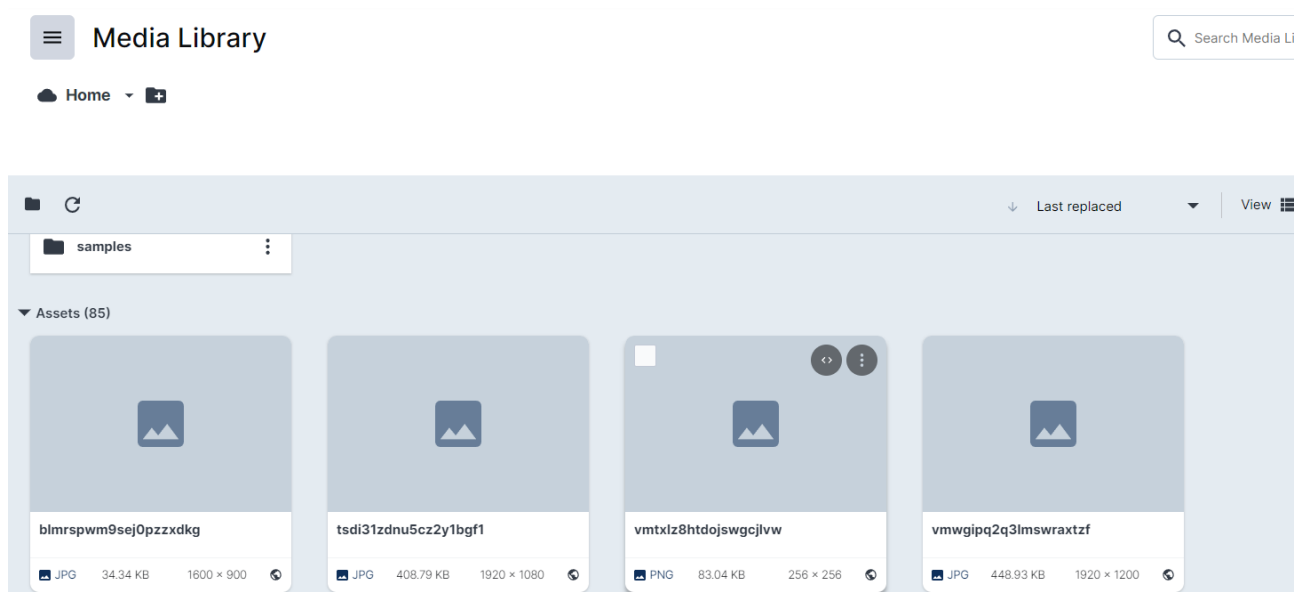
Per poter realizzare questa funzionalità, si è andato prima a richiedere uno spazio dedicato su Cloudinary.

Si è poi andata a scrivere una funzione asincrona apposita nel file Signup.js. Se ne riporta il codice per intero qui di seguito:

```
async function uploadImage() {
  const data = new FormData();
  data.append('file', image);
  data.append("upload_preset", "qtehprwd"); //quello è l'id del preset di
  Cloudinary, si richiama l'API con upload_preset
  try{
    setUploadingImg(true);
    let res = await
fetch('https://api.cloudinary.com/v1_1/webtrc/image/upload', {
      method: 'post',
      body: data
    })
    const urlData = await res.json();
    setUploadingImg(false);
    return urlData.url;
  } catch (error) {
    setUploadingImg(false);
    console.log(error);
  }
}
```

Tale funzione riceve l'immagine caricata dall'utente e invia una POST verso l'API di Cloudinary per richiederne l'upload. Si mette poi in attesa della risposta da parte dell'API. Da tale risposta verrà poi recuperato l'URL per raggiungere l'immagine nel cloud.

Sarà inoltre possibile vedere tutti gli asset caricati su Cloudinary andando nella sezione Media Library.³



³ Lo screen è stato fatto senza aspettare che il browser renderizzasse le immagini.

4 Realizzazione chat

Essendo una chat una particolare applicazione di messaggistica istantanea, per la realizzazione si potevano effettuare diverse scelte. Alla fine la scelta è ricaduta sull'utilizzo della tecnologia delle WebSocket.

Una WebSocket è una socket nel web, quindi realizzata a livello applicativo, che fornisce un canale di comunicazione full-duplex attraverso una singola connessione TCP. Il protocollo WebSocket permette maggiore interazione tra browser e server, facilitando la realizzazione di applicazioni che forniscono contenuti in tempo reale. Questo è reso possibile fornendo un modo standard per il server di mandare contenuti al browser senza dover essere sollecitato dal client.

Non sempre le WebSocket sono supportate. Quando questo accade, la connessione viene comunque stabilita con il meccanismo del long polling.

Per poter utilizzare le WebSocket si è fatto uso della libreria JavaScript *socket.io*, che fornisce tutte le API necessarie. Anche questa libreria è stata aggiunta nelle directory della webapp attraverso npm. In particolare, va aggiunto *socket.io-client* nella cartella relativa al front-end e *socket.io-server* nella cartella relativa al back-end.

Vediamo come viene effettuato uno scambio di messaggi analizzando prima cosa succede a lato client e poi a quello server.

4.1 Client

Il codice discusso in questo paragrafo è quello presente nel file *MessageForm.js*.

Una volta che l'utente avrà scritto un messaggio e cliccato su "invia", verrà invocata la funzione *handleSubmit()*.

All'interno di tale funzione verranno salvati in delle variabili locali *data* e *ora* di invio del messaggio e la stanza nella quale il messaggio è stato scambiato.

Fatto questo verrà richiamata una funzione della libreria *socket.io*: *socket.emit()*. A questa funzione passiamo come parametri l'evento e le variabili locali citate prima.

```
function handleSubmit(e) {
  e.preventDefault();
  if (!message) return;
  const today = new Date();
  const minutes = today.getMinutes() < 10 ? "0" + today.getMinutes() :
today.getMinutes();
  const time = today.getHours() + ":" + minutes;
  const roomId = currentRoom;
  socket.emit("message-room", roomId, message, user, time, todayDate);
  setMessage("");
}
```

4.2 Server

Il server sarà stato messo preventivamente in ascolto dell'evento *message-room* sulla socket.

Alla ricezione del messaggio, il server dovrà gestire l'evento: verrà attivata una funzione che salva il nuovo messaggio, recupera la conversazione e invia a sua volta un evento *message-room* contenente l'intera conversazione di quella stanza.

Il server invia inoltre un secondo evento, *notifications*, che permetterà agli altri utenti di visualizzare l'icona di notifica di fianco al nome della chat dove il messaggio è stato scambiato.

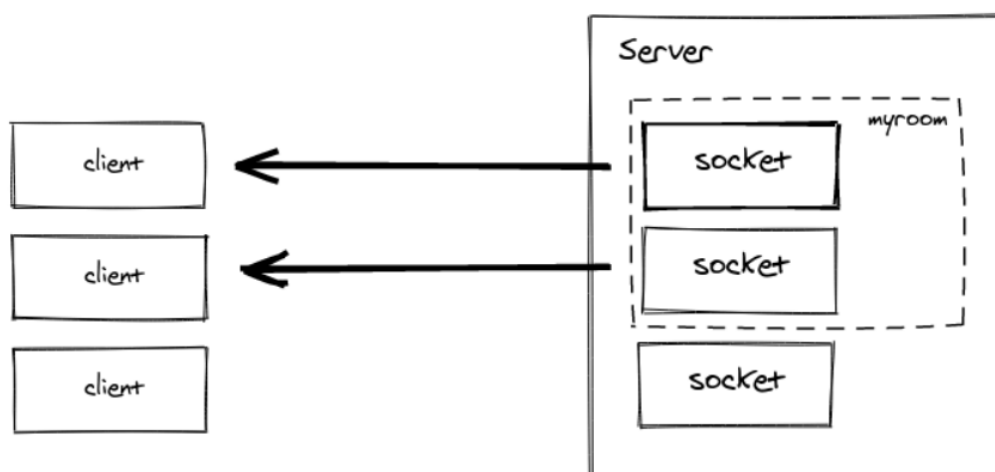
Il codice che realizza quanto appena descritto è il seguente:

```
socket.on('message-room', async(room, content, sender, time, date) => {  
  const newMessage = await Message.create({content, from: sender, time, date, to:  
room});  
  let roomMessages = await getLastMessagesFromRoom(room);  
  roomMessages = sortRoomMessagesByDate(roomMessages);  
  io.to(room).emit('room-messages', roomMessages);  
  socket.broadcast.emit('notifications', room)  
})
```

Notiamo che tra i parametri necessari c'è anche *room*, l'identificativo della stanza (ovvero la conversazione) a cui si fa riferimento.

Quello delle *room* è un concetto molto importante in socket.io: si tratta di canali che possono essere definiti arbitrariamente e a cui le socket possono partecipare (o lasciare). È molto utile soprattutto per inviare in broadcast messaggi solo ad un determinato subset di socket.

Ricordando che ad ogni socket è associato un identificativo univoco randomico SocketID, per convenienza ogni socket partecipa di default ad una room identificata dal suo stesso SocketID automaticamente.

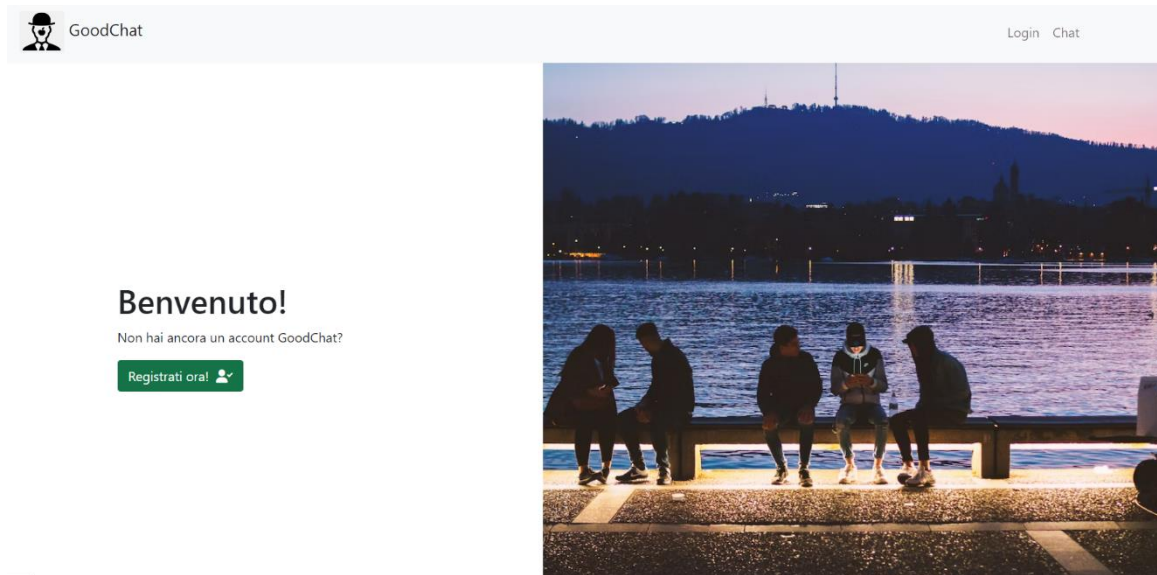


Quello delle room è un concetto che riguarda solo il lato Server: i client non sanno a quale room sono stati assegnati.

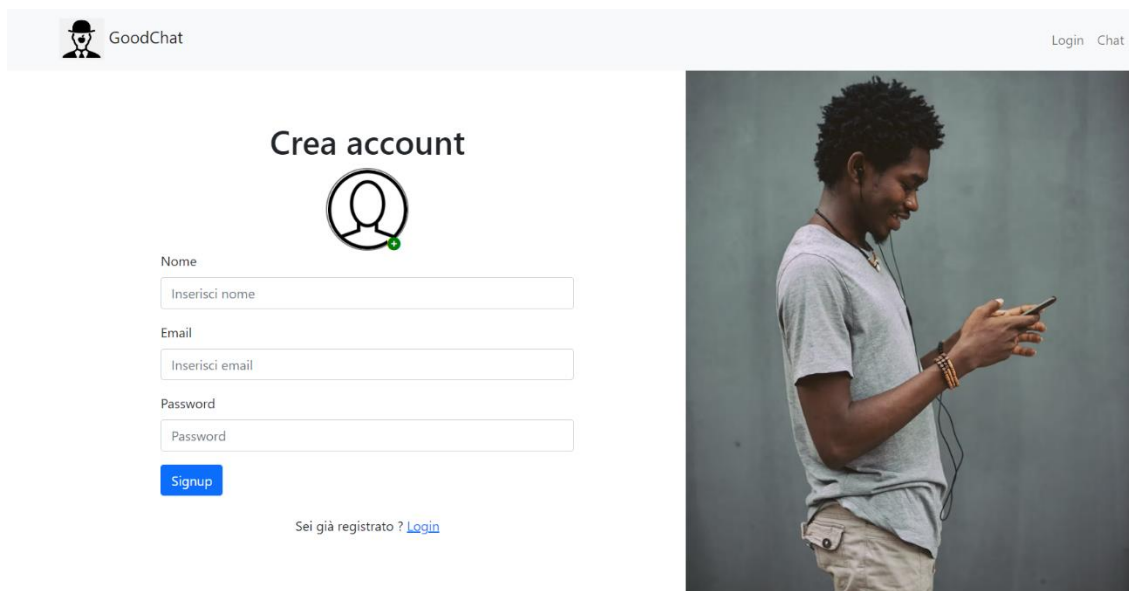
Le room inoltre non vanno create esplicitamente dal programmatore: una room viene creata ogni qualvolta una socket fa la *join* su di essa.

5 Demo

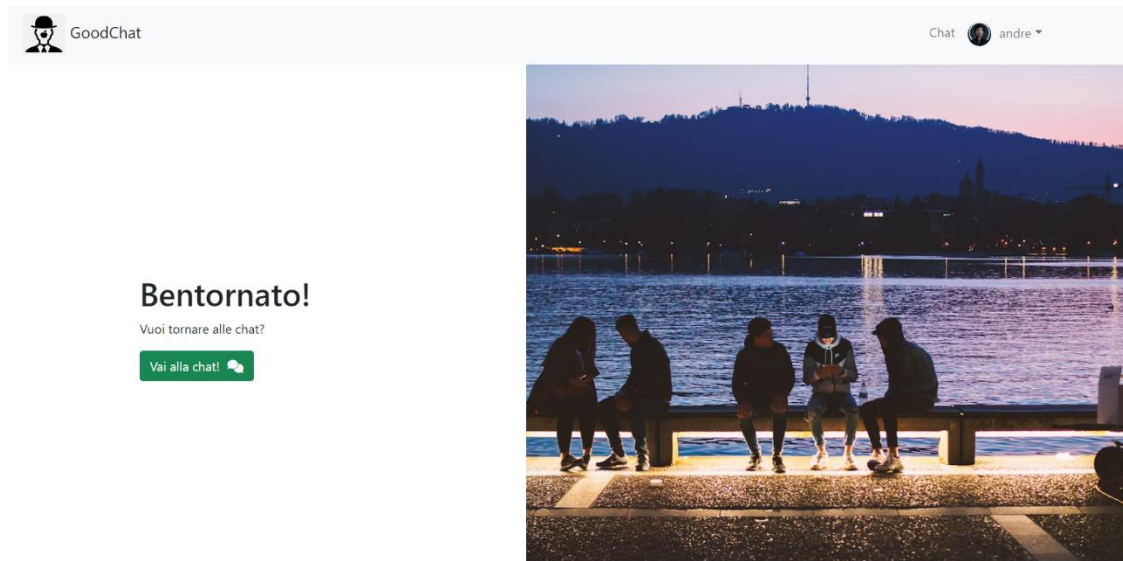
Una volta collegati su <http://localhost:3000>, verrà visualizzata la seguente homepage.



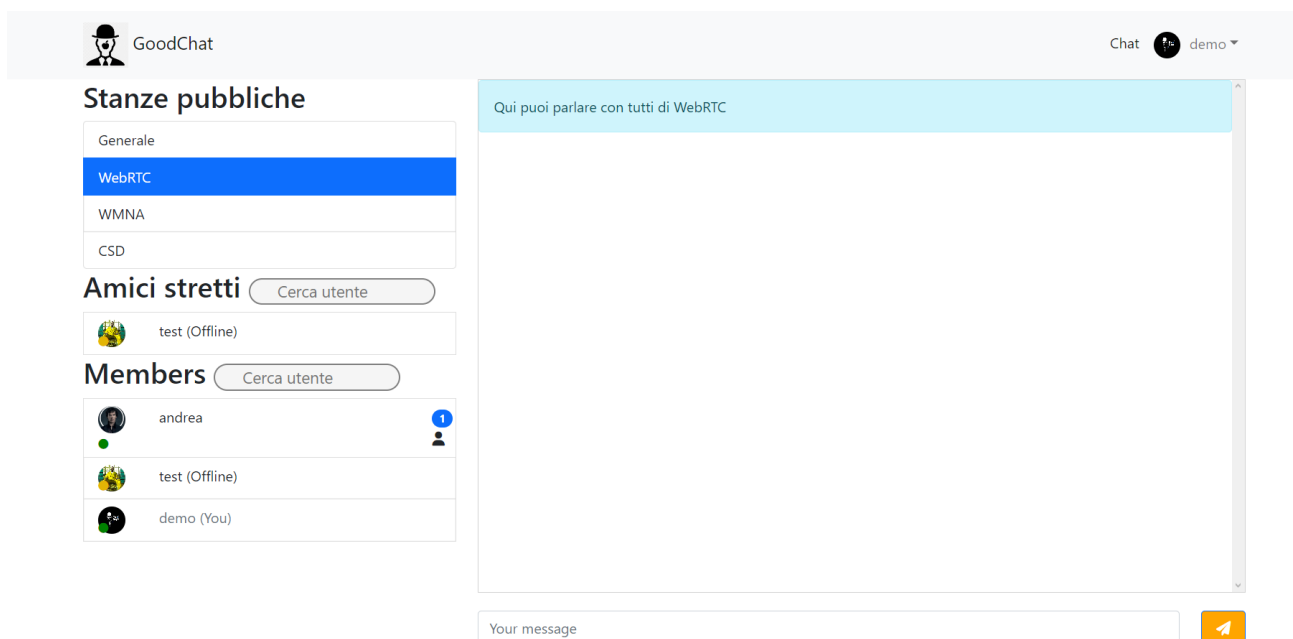
Cliccando su “Registrati ora” si verrà reindirizzati alla pagina dedicata al SignUp.



Inseriti i dati appositi, superati i controlli ortografici e sul formato dell'immagine inserita come avatar, si verrà reindirizzati nuovamente alla homepage. L'utente appena registrato però risulterà già loggato.






Cliccando su “Chat” si verrà reindirizzati all'applicazione vera e propria. Chiaramente le chat sono consentite solo ad utenti già loggati, quindi provando ad accedervi prima di effettuare il login verrà visualizzato un messaggio di errore che invita l'utente ad effettuare l'accesso.



Come si nota nell'immagine, è possibile: chattare su delle stanze pubbliche, chattare privatamente con utenti iscritti al sito, aggiungere un utente alla lista amici stretti, visualizzare se un utente è online o meno e vedere se ci sono messaggi in arrivo. Sono inoltre presenti dei filtri sulla ricerca (case-insensitive) che permettono di trovare più velocemente l'utente desiderato.

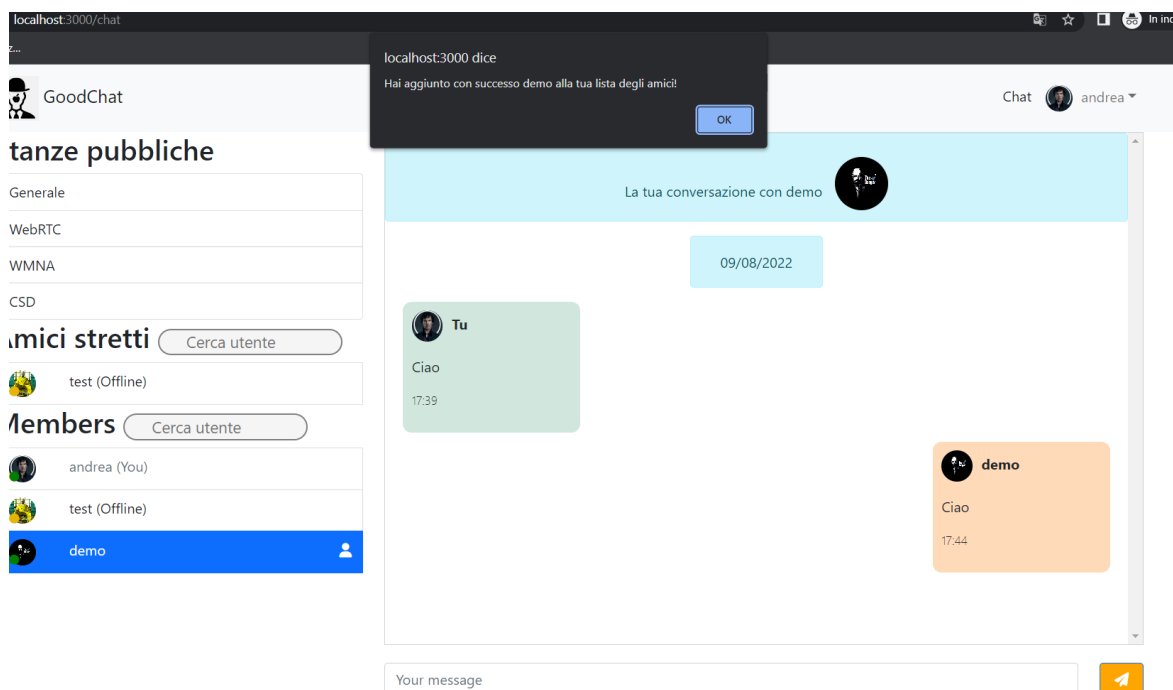
Members

	andrea (You)
	test
	demo

Members

	test
---	------

Cliccando sull'icona alla destra del nome utente è possibile aggiungere un utente alla lista amici stretti. Verrà visualizzato un alert di conferma se l'utente è stato aggiunto con successo.



The screenshot shows the GoodChat application running on localhost:3000. The interface includes a sidebar with navigation links: "tanze pubbliche" (General, WebRTC, WMNA, CSD), "amici stretti" (Cerca utente), and "Members" (Cerca utente). The main chat area displays a conversation with "demo". A confirmation alert is shown: "localhost:3000 dice: Hai aggiunto con successo demo alla tua lista degli amici!" with an "OK" button. The chat history shows messages from "Tu" and "demo".

localhost:3000/chat

GoodChat

tanze pubbliche

- Generale
- WebRTC
- WMNA
- CSD

amici stretti

Members


andrea (You)


test (Offline)

demo

localhost:3000 dice
Hai aggiunto con successo demo alla tua lista degli amici!

OK


Chat  andrea

La tua conversazione con demo 

09/08/2022

Tu
Ciao
17:39

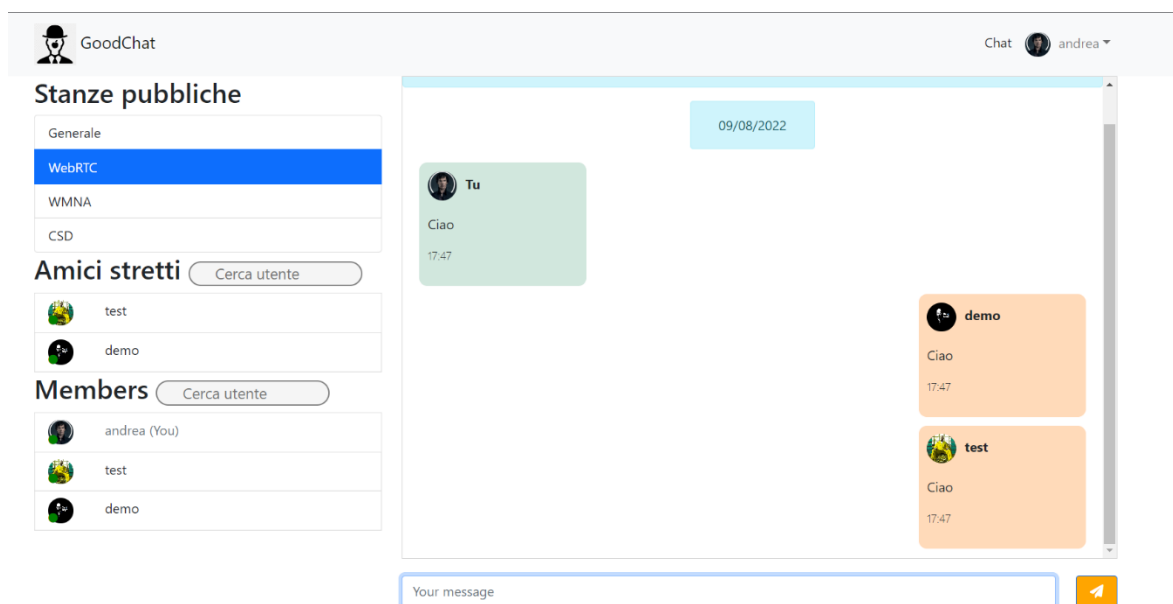
demo
Ciao
17:44

Your message 

In seguito al messaggio di alert il nuovo amico verrà visualizzato nell'apposita lista e scomparirà l'icona affianco al suo nome. Non è necessario refreshare la pagina (grazie ad un Hook React).

Nell'immagine precedente si può anche notare come si presenta una conversazione tra due utenti.

È anche possibile partecipare a 4 dei canali pubblici offerti dalla piattaforma. In questo caso i messaggi scambiati arriveranno a tutti gli utenti e tutti gli utenti potranno partecipare alla conversazione.



Un'ultima funzionalità fornita è la possibilità di modificare i propri dati. Non sarà possibile modificare l'email associata all'account, ma solo: nome, password e avatar.