Indice

Overview

hAPPy è una web application che sfrutta il progetto open-source WebRTC (*Web and Real Time-Communication),* quest'ultimo permette ad un browser web la comunicazione real-time tramite l'uso di API.

L'applicazione realizzata implementa due funzionalità principale: messaggistica istantanea e videochiamate tra due utenti iscritti all'app. L'utente quindi dopo essersi registrato e dopo aver effettuato il login potrà iniziare a scambiare messaggi in tempo reale e avviare videochiamate con altri utenti, anch'essi registrati e presenti nella sua lista contatti.

La progettazione ha previsto la divisione del software in due parti:

-un'applicazione *server* (*back-end*) che offre servizi;

-un'applicazione *client* (*front-end*) che gestisce l'interfaccia con l'utente e permette di richiedere servizi al server.

Gli strumenti utilizzati sono stati *Node.js*, utile per scrivere le applicazioni lato server, e *React*, per l'implementazione dell'interfaccia grafica. Per la comunicazione real-time si è serviti delle librerie *Javascritpt*: *socket.io e simple-peer*. Il database scelto per l'archiviazione dei dati è stato *MongoDB*.

Infine, è stato impiegato *Postman* per il test di *API Rest* e *Heroku* per il deployment dell'app.

Back-end

L'applicazione durante la fase di sviluppo è stata eseguita in locale e il server è stato reso raggiungibile all'indirizzo http://localhost:5000.

Per l'implementazione del lato server è stato utilizzato il linguaggio *Javascript* e quindi Node.js come *runtime Javascript*, un ambiente di esecuzione che permette di eseguire *Javascript* come un qualsiasi altro linguaggio di programmazione.

Setup dell'ambiente

Per poter installare i package, i moduli, necessari allo sviluppo dell'applicazione è stato utilizzato *yarn*, un *package manager* di *Javascript,* rilasciato da Facebook per sopperire ai problemi che presentava il suo rivale *npm*.

Yarn è stato installato mediante il comando:

```
$npm install -global yarn
```

Con il seguente comando è stato creato il file *package.json*:

```
$yarn init
```

Il *package.json* è scritto interamente in JSON e contiene vari metadati rilevanti per il progetto, come la descrizione del progetto, la versione, informazioni sulla licenza. Esso

permette di gestire le dipendenze del progetto e registrare alcuni comandi per facilitare le operazioni dello sviluppatore (è possibile associare degli script con nomi predefiniti come ad esempio start, prestart).


```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^5.0.1",
    "cors": "^2.8.5",
    "dotenv": "^16.0.0",
    "express": "^4.17.3",
    "mongoose": "^6.2.9",
    "nodemon": "^2.0.15",
    "peer": "^0.6.1",
    "require-math": "^3.3.1",
    "simple-peer": "^9.11.1",
    "socket.io": "^4.4.1"
  }
```


I seguenti comandi sono serviti ad installare le dipendenze che verranno aggiunte *al package.json* e messe nella cartella *node_modules*:

```
$yarn install nome_dipendenza
```

```
$yarn add nome_dipendenza
```

È indifferente quale dei due comandi viene utilizzato.

L'output generato, la prima volta che si manda in esecuzione il comando di installazione, è

il file *yarn.lock* (*package-lock.json* nel caso si fosse utilizzato il package manager npm) che elenca tutte le versioni delle dipendenze correnti al momento dell'installazione. Se il file *yarn.lock* è gia esistente quando viene eseguito *yarn install* vengono installate le dipendenze del *yarn.lock*. Se si installa una nuova dipendenza e il file esiste già questa verrà anche aggiunta a quest'ultimo con la sua versione corrente.

La differenza sostanziale tra il *package.json* e *yarn.lock* è che il primo indica la versione minima di installazione di una dipendenza, e se viene aggiornata *il package.json* non ne risentirà, mentre *yarn.lock* indica la versione corrente di una dipendenza installata.

Quindi il *package.json* avrà le dipendenze richieste dall'applicazione mentre *yarn.lock* quelle effettivamente installate. In questo modo con *yarn.lock* se si volesse eseguire il codice in un ambiente diverso da quello dove è stato sviluppato verrebbero installate le stesse dipendenze, in tal modo non si rischia di installare un pacchetto con una versione successiva.

Per mandare in esecuzione lo script, invece, è necessario il comando:

```
$yarn start
```

E' stato possibile utilizzare questo dopo averlo associato nella sezione script nel file *package.json.*

Implementazione server

Dato l'utilizzo di *Node.js*, che supporta il pattern *Model-View-Controller*, il lato *back-end* è stato suddiviso in *Model*, *Controller* e *Route*. Nei seguenti sottoparagrafi sono analizzati in dettaglio.

Index.js

Index.js è il file che viene eseguito per primo: gestisce l'avvio, il routing, le funzioni dell'app e altri moduli che devono essere importati. Infatti per poterlo eseguire è stato

aggiunto nel *package.json* lo script:

```
"start": "nodemon index.js"
```

Dove *nodemon* è uno strumento che permette di riavviare automaticamente l'applicazione quando vengono salvate delle modifiche ed è un wrapper sostitutivo per *node*.

Express

Express è un framework open-source per applicazioni web *Node.js*.

Che semplifica alcuni compiti che risultano fastidiosi da gestire con Node come, ad esempio, la gestione delle rotte o l'accesso al file system. Si basa sul concetto di middleware, che possono essere visti come dei "moduli" ognuno dei quali fornisce una caratteristica specifica.

Dopo aver creato un'istanza di express:

```
const app = express();
```

Si può utilizzare un middleware nel seguente modo:

```
app.use(middleware);
```

Uno dei middleware utilizzati è *CORS*. Nel caso in cui un client proveniente da un dominio, porta o protocollo differente da quello del server faccia richiesta non è detto che questa avrà esito positivo. È un meccanismo di sicurezza del browser e CORS entra in gioco per disabilitare questo meccanismo e consentire l'accesso a queste risorse. L'header di *access-control-allow-origins* determina quali origini sono abilitate al CORS; nel caso ci sia * significa che sono abilitate tutte.

Dopo dunque aver installato la dipendenza di *CORS*, è stato importato il modulo nel progetto e utilizzato come *middleware*.

Per avviare invece il server sulla porta desiderata si ha:

```
const server = app.listen(process.env.PORT, ()=>{
   console.log(`Server started on port ${process.env.PORT}`)
});
```

Il server dopo l'avvio resta in ascolto su tale porta.

Model

Data la necessità di dover effettuare uno storage dei dati degli utenti si è deciso di utilizzare come database *MongoDB* essendo multipiattaforma e open source, inoltre esso è classificato come *noSQL*.

Un server *MongoDB* può contenere più database. Ogni ***database*** raggruppa un certo numero di ***collezioni***, senza schema, che a loro volta contengono un insieme di ***documenti***. Questi ultimi hanno uno schema a sé e non è altro che un oggetto JSON.

È stato quindi creato un database locale chiamato **chat** ed è stata utilizzata **mongodb://localhost:27017/chat** come URI per permettere la connessione tra applicazione e MongoDB.

Il database creato contiene due collezioni:

- *user*, dove vengono archiviate le informazioni dell'utente;
- *message*, dove vengono archiviati i messaggi tra gli utenti.

Per poter integrare MongoDB con NodeJS si è utilizzato *mongoose*, che è una libreria *Object Data Modeling* (ODM) che permette di gestire la relazione tra i dati, fornisce la convalida degli schemi e traduce gli oggetti definiti nel codice in oggetti *MongoDB*.

Il codice seguente illustra la connessione a Mongo ed è stato inserito nel file *index.js*:

```
mongoose.connect(process.env.MONGO_URL,{
  useNewUrlParser: true,
  useUnifiedTopology: true,
}).then(()=>{
   console.log("DB Connection successfull");
}).catch((err)=>{
```

```
    console.log(err.message);
  });
```

Lo schema della collezione *user* è il seguente:

```
const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    min: 3,
    max: 20,
    unique: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
    max: 50,
  },
  password: {
    type: String,
    required: true,
    min: 8,
  },
  isAvatarImageSet: {
    type: Boolean,
    default: false,
  },
  avatarImage: {
    type: String,
    default: "",
  },
  friends:[{
    type: String,
    default:"",
  }]
});
```

```
module.exports = mongoose.model("Users", userSchema);
```

Mentre quello della collezione *message* è:

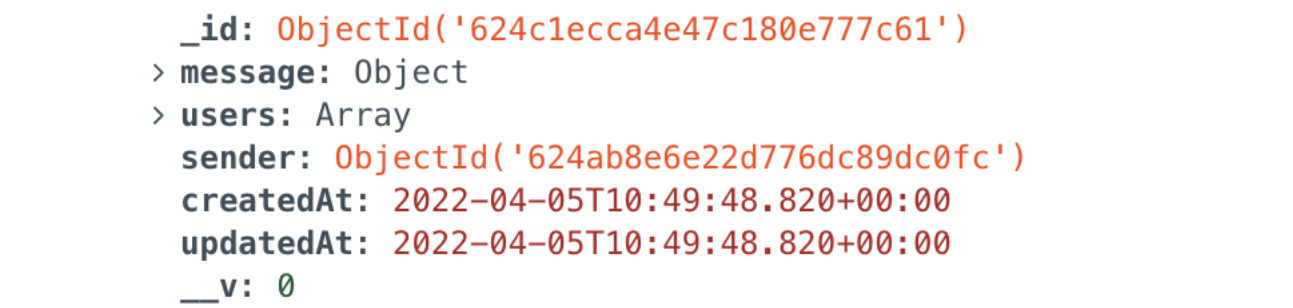
```
const mongoose = require("mongoose");

const MessageSchema = mongoose.Schema(
 {
  message: {
   text: { type: String, required: true },
  },
  users: Array,
  sender: {
   type: mongoose.Schema.Types.ObjectId,
   ref: "User",
   required: true,
  },
 },
 {
  timestamps: true,
 }
);

module.exports = mongoose.model("Messages", MessageSchema);
```

La gestione del database è avvenuta utilizzando *MongoDB Compass*, che non è altro che una GUI per *MongoDB* dove è potuto avere una panoramica di come i dati sono stati archiviati e poterli gestire.

Nelle immagini seguenti vi è un esempio di documento per entrambi gli schemi.



Controller

Nel *Controller* sono state definite tutte le funzioni, che ricevuto un comando dal front-end, attraverso la *View*, vanno a modificare il *Model* e che possono portare a cambiamenti della *View* stessa.

Le funzioni che permettono di fare questo sono asincrone (viene utilizzata la keyword *async)* dato che abbiamo un'iterazione con il database. Esse non restituiscono un valore ma una *Promise.* Quest'ultima è un oggetto che rappresenta il risultato di un'operazione asincrona che controlla il flusso di tale chiamata, e ci dice quando è essa termina o se fallisce.

La keyword *await*, valida solo all'interno di funzioni asincrone, e ad essa è associata una promise. Si attende dunque che tale attività asincrona venga risolta e venga restituita una *Promise*. Utile nel caso in cui si vuole far in modo che il codice sincrono attenda quello asincrono.

I parametri delle funzioni asincrone utilizzati sono:

- *req:* è la richiesta inviata dal client;

- *res:* è la risposta. *res.json()* ritorna al client una risposta di tipo JSON.

- *next:* argomento che indica la funzione di *callback*, tale funzione viene chiamata se la promessa è rigettata/rifiutata o si verifica un errore e viene chiamata sull'errore stesso

Nel seguente esempio è riportata la funzione asincrona per la registrazione:

```
module.exports.register = async (req, res, next) => {
  try {
    const { username, email, password } = req.body;
    const usernameCheck = await User.findOne({ username });
    if (usernameCheck)
      return res.json({ msg: "Username already used", status: false });
    const emailCheck = await User.findOne({ email });
    if (emailCheck)
      return res.json({ msg: "Email already used", status: false });
    const hashedPassword = await bcrypt.hash(password, 10);
    const user = await User.create({
      email,
      username,
      password: hashedPassword,
    });
    delete user.password;
    return res.json({ status: true, user });
  } catch (ex) {
    next(ex);
  }
};
```

Route

A questo punto definite le API si deve esporre all'esterno in modo che il front-end possa fare la richiesta. Devono, dunque, essere definiti dei percorsi, delle rotte (*Route*).

Express mette a disposizione il metodo *express.route()* che crea un nuovo oggetto *Router*,

necessario per la gestione delle richieste da parte di un client.

Tale meccanismo di router permette di definire le rotte in un file separato da quello *index.js,* in modo da non riempire quest'ultimo con molto codice. Altrimenti poteva essere utilizzato sull'oggetto app istanziato nell'*index.js.*

Dopo aver definito l'oggetto router vengono chiamati su di esso i metodi che corrispondono a metodi HTTP; ad esempio, *router.get()* per gestire le richieste GET e *router.post()* per quelle POST. A tali metodi sono passati come parametro il *path*, che è un percorso sul server, e l'*handler* è la funzione che si esegue quando il percorso è abbinato e che serve a gestire la richiesta.

```
//UserRoutes
const {
    register,
    login,
    setAvatar,
    getAllUsers,
    addFriends,
    getAllUsersDB
} = require("../controllers/userController");

const router = require("express").Router();

router.post("/register",register);
router.post("/login",login);
router.post("/setAvatar/:id",setAvatar);
router.get("/allusers/:id", getAllUsers);
router.post("/addFriends/:id", addFriends);
router.get("/allusersdb/:id", getAllUsersDB);
module.exports = router;


//MessageRoutes
const { addMessage, getMessages } = require("../controllers/messageController");
const router = require("express").Router();

router.post("/addmsg/", addMessage);
```

```
router.post("/getmsg/", getMessages);

module.exports = router;
```

A questo punto sempre servendosi di *express*, nel file *index.js* è stata utilizzata:

```
app.use(base_path, router);
```

Per poter montare *router* come middleware nel percorso *base_path*. La funzione viene eseguita quando la base del percorso richiesto corrisponde al *base_path*.

Nel caso specifico a questo progetto si avrà:

```
app.use("/api/auth", userRoutes);
app.use("/api/messages",messageRoutes);
```

Mentre se si considera:

```
app.use(sub_path, handler);
```

imposta il *sub_path* e definisce un *handler* che viene eseguito quando il percorso è abbinato.

Dunque se arriva una richiesta del tipo http://localhost:5000/api/auth/register verrà eseguita la funzione che permette la registrazione dell'utente.

Test delle REST API

Postman è stato impiegato per testare le API del back-end. Esso non è altro che una piattaforma per lo sviluppo di API, che consente agli utenti di progettare, simulare, eseguire il debug, testare, documentare, monitorare e pubblicare API.

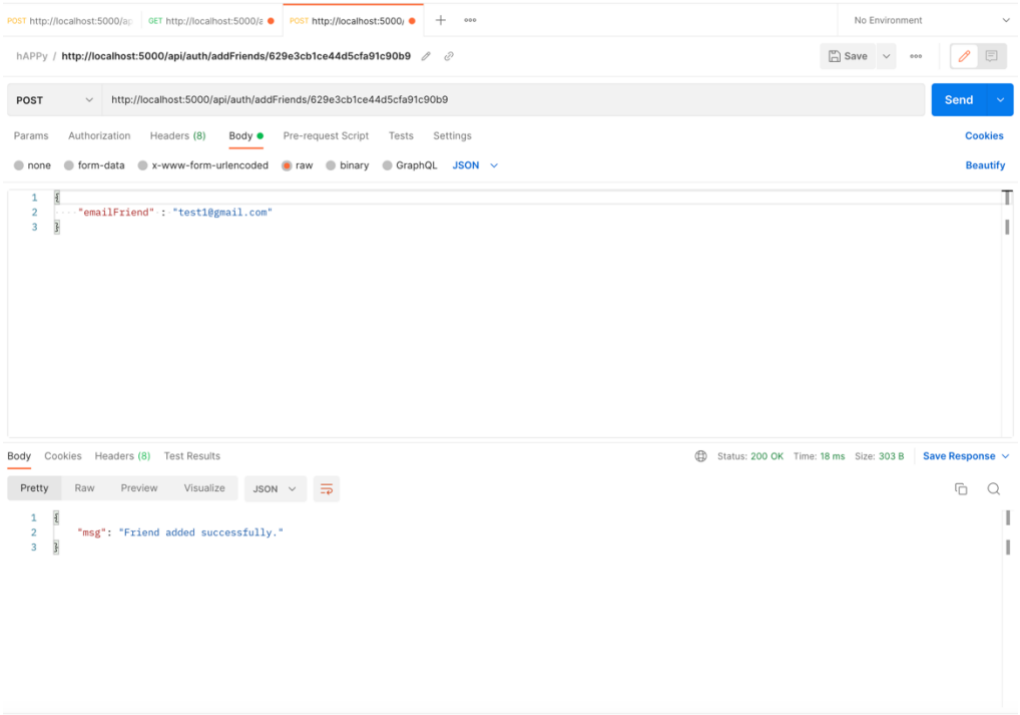
Si parla di REST API, in quanto con REST si richiede che ogni risorsa sul web è associato un URL e su queste risorse si possono fare le operazioni CRUD (Create, Retrieve, Update e Delate). In HTTP queste si traducono in post, *get, put e delate* che sono proprio quelle che permette di testare *Postman*,

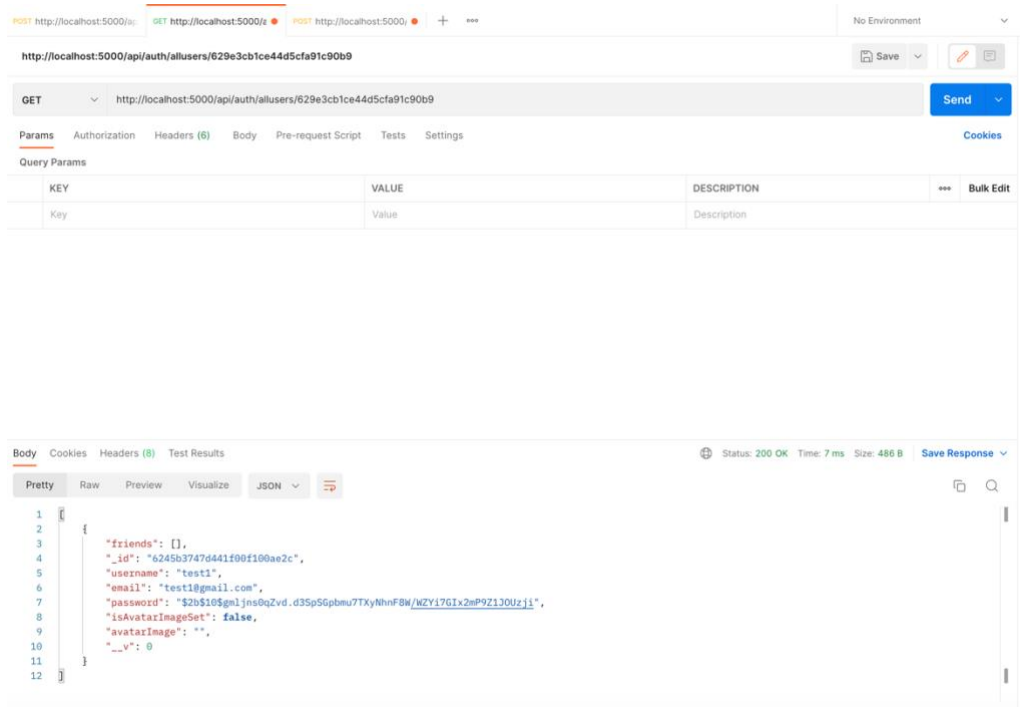
Le API REST comunicano tramite richieste HTTP per eseguire funzioni di database

standard come la creazione, la lettura, l'aggiornamento e l'eliminazione di record.

Tramite l'interfaccia grafica è possibile selezionare il tipo di chiamata da fare (*get, post* ecc..) e impostare l'URL a cui indirizzare la chiamata. Permette dunque di eseguire richieste HTTP ad un server di back-end. Per il body delle richieste è stato utilizzato JSON.

Di seguito è riportato l'esempio in cui prima si fa una *post* per aggiungere un amico e poi una *get* per vedere la lista degli amici.





Front-end

Il front-end è la parte dell'applicazione che l'utente finale e con il quale interagisce. Quindi esso implementa l'interfaccia grafica e in base agli eventi su di essa invia richieste al server; è raggiungibile all'indirizzo http://localhost:3000.

React

Si è scelto di utilizzare *React* per l'implementazione della parte front-end.

React è una libreria Javascript per la creazione di interfacce utente (UI) e consente di sviluppare applicazioni dinamiche che non necessitano di ricaricare la pagina per visualizzare i dati modificati. Inoltre, permette che le modifiche effettuare sul codice si possono visualizzare in tempo reale. Tutto ciò, dunque, permette uno sviluppo rapido, efficiente e flessibile di applicazioni web.

Per l'installazione di *React* nel progetto è stato utilizzato il seguente comando:

```
$npx create-react-app chat-app
```

Per quanto riguarda invece la sintassi si è scelto di utilizzare l'estensione di *Javascript*, .**jsx**, che produce elementi React e permette di aggiungere elementi al DOM senza usare particolari funzioni (come createElement() o appendChild()).

Componenti ed elementi

Tale libreria dà la possibilità di costruire interfacce complesse a partire da componenti più piccoli. I componenti rappresentano come la UI viene suddivisa logicamente e contengono

sia la logica che markup, possono essere sia funzioni che classi. Nel progetto si è scelto di utilizzare l'approccio con le funzioni. Per mettere insieme i componenti è stato utilizzato l'approccio della composizione; quindi, nel loro output i componenti fanno riferimento ad altri componenti. Esempi di componenti in *React* sono un pulsante, un form, una finestra di dialogo, uno schermo.

Tutto ciò che deve essere visualizzato invece è descritto da un elemento. Quest'ultimo viene restituito dai componenti ed è un oggetto che descrive virtualmente i nodi del DOM rappresentati da un componente. Se abbiamo dunque un componente definito mediante funzione, l'elemento è l'oggetto restituito dalla funzione. Con il componente classe, invece, l'elemento è l'oggetto restituito dalla funzione di rendering del componente.

Gli elementi *React* non sono ciò che vediamo nel browser, ma oggetti di memoria e non si può cambiare nulla di loro.

React DOM e Virtual DOM

Utilizzando *React* non è necessario ricorrere alle funzionalità del DOM per modificare elementi della pagina: è sufficiente cambiare le informazioni contenute all'interno dell'oggetto che rappresenta il modello dei dati, rappresentato dallo stato (*state*) per ottenere l'aggiornamento automatico dell'interfaccia utente. Per lo stato esiste un Hook chiamato *useState* che consente di tenere traccia dello stato di un componente mediante l'utilizzo di variabili di stato. Lo stato si riferisce generalmente a dati o proprietà che devono essere monitorati in un'applicazione.

React utilizza un sistema differente dagli altri per tracciare le modifiche e, quindi, determinare su quali elementi della pagina agire e quali cambiamenti apportare, ricorrendo al *Virtual DOM*, ossia una rappresentazione virtuale della struttura della pagina immagazzinata in memoria e del tutto simile al DOM originale, dal quale è vista come una astrazione.

Nel momento in cui si verifica un evento in cui è necessario reagire ad esso modificando gli elementi della pagina, *React* applica prima tali interventi al *Virtual DOM*. Analizzando

le differenze tra lo stato del Virtual DOM precedente al verificarsi dell'evento e quello nuovo ottenuto dall'applicazione delle modifiche, *React* determina i cambiamenti effettivi da apportare al DOM vero e proprio.

Calcolare le differenze tra il precedente stato e il corrente stato del Virtual DOM è estremamente veloce e grazie a esso si limitano al minimo gli interventi sul DOM reale, la cui manipolazione è molto più lenta, ottenendo così un incremento delle performance

Dal punto di vista del codice si consideri *index.js*, dove viene istanziata la radice del DOM:

```
import React from 'react';

import ReactDOM from "react-dom/client";
import './index.css';
import App from './App';


const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Dopo la creazione di un elemento *root* (chiamato così perché tutto quello che è al suo interno verrà gestito con *ReactDOM*) il quale è unico, viene effettuato il rendering dell'elemento App in modo tale che questa pagina essere visualizzata e aggiunta al DOM.

Infatti, l'elemento *root* è l'elemento radice dell'albero *React DOM* ed è l'elemento passato alla funzione di rendering.

Il rendering nel gergo *React* ha a che fare con la creazione di elementi all'interno del client: è l'atto in cui la libreria crea un nuovo elemento e lo inserisce nel DOM per raffigurarlo nella pagina.

Interessante è anche l'hook *useEffect* che React mette a disposizione e che permette di

eseguire la funzione *effect* dopo aver effettuato le modifiche del DOM. *React* esegue gli *effect* dopo ogni rendering, incluso il primo.

React Router

React Router è una libreria che permette di creare applicazioni *React* con più pagine in cui la transizione fra una pagina e l'altra avviene in maniera dinamica tramite *Javascript*, senza dover ogni volta ricaricare la pagina.

React Router mette a disposizione diversi tipi di *Router* a seconda del genere di applicazione che vogliamo realizzare o delle funzionalità di cui abbiamo bisogno.

Per il progetto è stato utilizzato il componente *BrowserRouter* che utilizza la History API di HTML5, quest'ultima risolve il problema riguardante gli URL dopo aver eseguito una richiesta AJAX. Con AJAX è consentito effettuare l'aggiornamento dinamico di una pagina senza doverla ricaricare. Quindi a seguito di una richiesta AJAX, dopo aver ricevuto i nuovi contenuti l'URL non veniva modificato se non si ricaricava la pagina e quindi se condiviso con altri non veniva visualizzato il nuovo contenuto in quanto l'URL era lo stesso. History API di HTML5 ha risolto questo problema.

Nel file APP.js sono stati quindi definiti i vari percorsi, le rotte dell'applicazione. Si è andato a creare un nuovo componente APP, nel quale è stato utilizzato un elemento *BrowserRouter* che ha come elemento discendente *Routes* che a sua volta ha come discendenti una serie di elementi *Route*.

Ogni elemento *Route* ha un attributo **path** che indica per quale URL è mostrato un certo **elemento**.

```
import React from "react";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Chat from "./pages/Chat";
import Login from "./pages/Login";
import Register from "./pages/Register";
import SetAvatar from "./pages/SetAvatar";
import ChatVideo from "./pages/ChatVideo";
```

```
import * as process from 'process';

window['process'] = process;


export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/register" element={<Register />} />
        <Route path="/login" element={<Login />} />
        <Route path="/setAvatar" element={<SetAvatar />} />
        <Route path="/" element={<Chat />} />
        <Route path="/chatVideo" element={<ChatVideo />} >
      </Routes>
    </BrowserRouter>
  );
}
```

React Router permette di navigare anche a livello di codice utilizzando il componente *Navigate*. Il routing a livello di codice avviene su una sorta di evento che non è ad esempio un click su un link, ma ad esempio, come accade nel codice del progetto, l'oggetto fa logout e viene reindirizzato alla pagina di login.

Per fare questo è stato importato l'hook *usenavigate.*

```
import React from "react";
import { useNavigate } from "react-router-dom";
import { BiPowerOff } from "react-icons/bi";
import styled from "styled-components";

export default function Logout() {
  const navigate = useNavigate();
  const handleClick = async () => {

     localStorage.clear()
     navigate("/login");

  };
```

```
  return (
    <Button onClick={handleClick}>
      <BiPowerOff />
    </Button>
  );
}
```

Axios

Axios è una libreria *Javascript* che permette di connettersi con le API di back-end e di effettuare e gestire le richieste effettuate tramite il protocollo HTTP.

Il vantaggio di *Axios* sta nel fatto che è *promise-based*, permettendo quindi l'implementazione del codice asincrono. Il promise, su cui si basa *Axios*, è un oggetto JavaScript che consente di completare delle richieste in maniera asincrona, facendole passare per tre stati (in sospeso, soddisfatta, rifiutata).

In *APIRoutes .js* sono state riportare tutte i percorsi per effettuare le richieste al server:

```
export const host = "http://localhost:5000";
export const registerRoute = `${host}/api/auth/register`;
export const loginRoute = `${host}/api/auth/login`;
export const setAvatarRoute = `${host}/api/auth/setAvatar`;
export const allUsersRoute = `${host}/api/auth/allusers`;
export const sendMessageRoute = `${host}/api/messages/addmsg`;
export const getAllMessageRoute = `${host}/api/messages/getmsg`;
export const addFriendRoute = `${host}/api/auth/addFriends`;
export const allUsersDBRoute = `${host}/api/auth/allusersdb`;
```

Per effettuare una richiesta *get* al server, viene passato l'URL dell'API esposta dal server, come riportato nell'esempio:

```
const data = await axios.get(`${allUsersRoute}/${currentUser._id}`);
setContacts(data.data);
```

Nel caso in una richiesta *post* al server oltre all'URL vengono passati anche i parametri che costituiranno il body della richiesta:

```
await axios.post(sendMessageRoute, {
    from: data._id,
    to: currentChat._id,
    message: msg,
  });
```

Implementazione della chat e videocall

Chat

WebSocket

Per l'implementazione della chat si è scelto di utilizzare *WebSocket* che supporta in modo nativo lo scambio in tempo reale di dati generici tramite l'astrazione del canale dati.

Una *WebSocket* non è altro che una socket nel web, come quella tradizionale, ma a livello applicativo. Essa è mappata su una socket del Sistema Operativo.

WebSocket è progettato per essere implementato sia lato browser che server ma può essere utilizzato anche da qualsiasi applicazione client-server. Il protocollo è un'implementazione basata sul protocollo TCP. La sua unica correlazione con l'HTTP è nel modo in cui fa *l'handshake*, che ricorda un'implementazione HTTP così il server può gestirla come una normale richiesta di connessione sulla stessa porta. All'interno della richiesta vengono specificati degli opportuni campi che identificano una richiesta *WebSocket*. Quindi si negozia con HTTP ma si specifica che poi si vuole parlare con un altro protocollo.

Grazie alla libreria *Javascript* ***socket.io*** sono fornite le API che consentono di utilizzare le WebSocket a livello applicativo e quindi necessarie per una comunicazione real-time bidirezionale basata sugli eventi su ogni piattaforma, browser o devices.

Al lato server per poter utilizzare la libreria è stato installato il modulo *socket.io* mediante il comando:

```
$yarn add socket.io-server
```

Al lato client, invece, è stato importato il modulo *socket.io-client*:

```
import {io} from "socket.io-client";
```

Per inviare gli eventi è stato utilizzato il metodo *emit(type,data)*, con *type* che indica il tipo di evento e *data* è il payload dell'evento.

Per ricevere eventi, invece, si è usato il metodo on(type,callback), dove si attende un evento di tipo *type* e alla ricezione si esegue la *callback* registrata.

Lato server, dopo aver instanziato un server *Socket.io*, collegato a un server HTTP:

```
const io = socket(server, {
  cors: {
    origin: "http://localhost:3000",
    methods: ["GET", "POST"],
    credentials: true,
  }
});
```

Si attende l'evento *connection* per la prima connessione:

```
io.on("connection", (socket) => {
//Altri eventi attesi
});
```

Si noti che tale evento ha come payload l'oggetto *socket* sul quale possono essere registrati gli altri eventi attesi. Infatti, socket è specifico per quel determinato client. Permette di comunicare direttamente con esso. Quindi gli eventi *emit* e *on* consentiranno di ascoltare eventi da quel client specifico o emettere eventi a quel client specifico.

Lato client invece si ha la registrazione al backend, avviando dunque una connessione verso l'indirizzo del server:

```
socket.current = io(host)
```

Implementazione chat

Lato client nel momento in cui il client clicca su una chat devono essere prelevati dal

database tutti i messaggi scambiati con il destinatario di quella chat, quindi, verrà effettuata una chiamata post vero l'API del server rinominata lato client con *getAllMessageRoute*, passando come body della richiesta l'id dell'utente corrente e l'id della chat selezionata.

```
useEffect( () => {
  const fetch = async()=>{
  const data = await JSON.parse(localStorage.getItem("chat-app-user"));
  const response = await axios.post(getAllMessageRoute, { //Prelevo tutti i messaggi che
l'utente ha scambiato con il contatto selezionato
    from: data._id, //Id dell'utente loggato
    to: currentChat._id, //Chat selezionata
  });
  setMessages(response.data);
 }
 fetch();
 }, [currentChat]);
```

Per la gestione invece dei messaggi in real-time entrano in gioco le *socket*.

Per quanto riguarda l'invio del messaggio, si ha al lato mittente(client) la funzione asincrona *handSendMsg* che è richiamata nel momento in cui l'utente clicca sul bottone di invio del messaggio. Viene azionato l'invio dell'evento *send-msg*, il cui payload è composto dall'id della chat corrente, dall'id del mittente e il messaggio che vuole inviare.

```
//Invio messaggio
 const handleSendMsg = async (msg) => {
  const data = await JSON.parse(localStorage.getItem("chat-app-user")); //Prelevo i dati dal
local storage dell'utente loggato
  socket.current.emit("send-msg", { //Invio l'evento send-msg
    to: currentChat._id, //Payload dell'evento: id della chat corrente, id dell'utente loggato e il
messaggio
    from: data._id,
    msg,
  });
```

```
    await axios.post(sendMessageRoute, { //Faccio una richiesta Post per archiviare sul
database il messaggio
      from: data._id,
      to: currentChat._id,
      message: msg,
    });
          ... //Azioni per gestire il messaggio inviato
  };
```

Il server, che è in ascolto di questo evento sulla socket, appena lo riceve lo deve gestire: il server appena riceve l'evento *send-msg*, cerca tra gli utenti online la *socket* del destinatario e se la trova gli invia l'evento *msg-recieve*, nel cui payload vi è il messaggio del mittente. Se l'utente non è online, la comunicazione in tempo reale non è necessaria, dunque il messaggio viene archiviato solo sul database.

```
socket.on("send-msg", (data) => {
    const sendUserSocket = onlineUsers.get(data.to); //Verifico che l'utente sia online, se è
online ottengo il valore della socket associato al suo id
    if (sendUserSocket) {
      socket.to(sendUserSocket).emit("msg-recieve", data.msg); //Invio un emit al
destinatario sulla sua socket
    }
});
```

Infine, il destinatario (client), che era in attesa della ricezione dell'evento msg-receive, il cui payload contiene il messaggio del mittente.

```
//Ricezione messaggio
  useEffect(() => {
    if (socket.current) {
      socket.current.on("msg-recieve", (msg) => { //Ricezione evento msg_receive, nel
payload c'è il messaggio del mittente
        setArrivalMessage({ fromSelf: false, message: msg }); //Setto il messaggio ricevuto
nella variabile arrivalMessage
      });
                    ... //Azioni per gestire il messaggio ricevuto
    }
  }, [socket]);
```

Utilizzando il meccanismo delle *socket* è stato implementato anche lo stato dell'utente con cui si vuole iniziare una conversarione (Online/Offline/Sta digitando).

Videocall

PeerJS

La funzionalità di videochiamata deve permettere di avviare una videochiamata tra due utenti. Prevede dunque l'utilizzo del protocollo *WebRTC* che permette una comunicazione real-time multimediale all'interno del browser, con l'approccio *peer-to-peer*, di tipo paritetico.

A tal scopo è stata utilizzata la libreria Javascript *simple-peer*, installata mediante il comando:

```
$yarn add simple-peer
```

Tale libreria mette a disposizioni le API necessarie per la creazione di un canale video/audio e altri dati *one-to-one*, la cui trasmissione sia in tempo reale.
Anche se crea connessioni peer-to-one, non include un server di segnalazione. In altre parole, viene creata la connessione tra i due peer dove viaggeranno i dati, quindi un canale diretto, ma non viene gestita la segnalazione.
Un server di segnalazione, solitamente viene implementato con le *WebSocket*, e serve per scambiare dati di segnalazione tra i due browser fino a quando non viene stabilita una connessione peer-to-peer.

Le seguenti righe di codice permettono di creare una nuova *peer connection*:

```
const peer = new Peer({
    initiator: true,
    trickle: false,
    stream: stream
  })
```

- *Initiator*: indica se il peer è colui che vuole iniziare la trasmissione;
- *Trickle*: per abilitare o disabilitare Trickle Ice. Se disabilitato si ha un unico evento 'signal' (è più lento);
- *Stream*: viene associato lo stream ottenuto dal *getUserMedia*(), con cui accede in locale ai propri dispositivi multimediali; si predispone così ad accedere a webcam e microfono

Il seguente evento viene attivato quando un peer desidera inviare i dati di segnalazione al peer remoto. Per *l'Initiator* si attiva immediatamente. Altrimenti si arriva quando viene ricevuta l'offerta remota. I dati incapsulano un'offerta *WebRTC*, una risposta o un candidato ICE. Ed è compito del server di segnalazione, quindi della *WebSocket* inoltrare i dati all'altro peer.

```
peer.on("signal", (data) => {
//...
}
```

Mentre quando il peer riceve uno stream video remoto, che può ad esempio essere visualizzato in un tag video viene utilizzato il seguente evento:

```
peer.on("stream", (stream) => {
//...
}
```

È stato utilizzato il seguente metodo, che viene chiamato ogni volta che il peer remoto emette un evento peer.on("signal"). Tale metodo serve per connettersi con l'altro peer:

```
peer.signal(signal);
```

Infine, il metodo riportato nel codice permette di distruggere e ripulire la connection peer:

```
connectionRef.destroy()
```

Implementazione videocall

Segnalazione

Quando l'utente che vuole avviare la video chiamata clicca sul pulsante viene chiamata la funzione getUserMedia() e quando lo stream locale non è più indefinito è chiamata *funzioneChiamata()* che crea inizialmente la nuova *connection peer* e emette l'evento *signa*l per inviare i dati di segnalazione:

```
        //Creazione di una nuova connection peer
        const peer = new Peer({
      initiator: true, //E' l'iniziatore
      trickle: false, //Tricke ICE disabilitato
      stream: stream, //Associo lo stream locale al peer
    })


                //Evento per inviare i dati di segnalazione al peer remoto
    peer.on("signal", (data) => {
      socket.current.emit("callUser", { //Emetto l'evento che sto chiamando l'utente
        userToCall: socketClient, //Passo come parametri: la socket del chiamato, i dati, e l'id di
chi chiama
        signalData: data,
        from: socket.current.id,

      })
    })
```

Di seguito è riportato un esempio di signal inviato dal chiamante al chiamato; il signal non è altro che il messaggio SDP che descrive la sessione multimediale:

```
                                                    ChatContainer.jsx:247
▶ {type: 'offer', sdp: 'v=0\r\no=- 1132291094184670558 2 IN IP4 127.0.0.1\r
  \ns…:2\r\na=sctp-port:5000\r\na=max-message-size:262144\r\n'}
```

Il server che è in ascolto dell'evento *callUser* sulla *socket*, lo riceve e lo gestisce,

emettendo a sua volta un evento callUser al peer chiamato, dove come payload passa il *signal*:

```
//Il server è in ascolto dell'evento callUser da parte di un client che vuole chiamare un altro
   socket.on("callUser", (data) => {
    socket.to(data.userToCall).emit("callUser", { signal: data.signalData, from: data.from })
//Invio al client da chiamare il segnale e il socket.id di chi ha chiamato
   })
```

A questo punto il chiamato riceve l'evento *callUser* sulla *socket* e viene chiamata la *funzioneRisposta()* dove viene creata anche qui la connection peer e generato l'evento *signal*, all'interno del quale viene emesso l'evento *answerCall* sulla *socket* e il peer il cui payload è costituito dal signal che deve essere inviato al peer chiamante. Inoltre, chiama il metodo signal al quale passa il segnale che ha ricevuto dal peer remoto così può connettersi:

```
//Socket in ascolto di callUser necessario per la videochiamata
   socket.current.on("callUser", ({signal,from}) => { //Nel payload ricevo il segnale, la socket
di chi chiama
    setReceivingCall(true)
    setCaller(from)
    setCallerSignal(signal)
   })


//Creazione di una nuova connection peer
 const peer = new Peer({
    initiator: false, //Non è l'iniziatore
    trickle: false, //Disabilito trickle ICE
    stream: stream, //Associo lo stream locale al peer
   })


 //Evento per inviare i dati di segnalazione al peer remoto
  peer.on("signal", (data) => {
            socket.current.emit("answerCall", { signal: data, to: caller
        }) //Emetto l'evento answerCall che ha come payload il segnale da inviare e la socket
        di chi ha chiamato
   })
```

```
//Si connette al peer remoto
peer.signal(callerSignal)
```

Di seguito è riportato un esempio di signal inviato dal chiamante al chiamato; anche qui è un messaggio SDP:

```
                                                    ChatContainer.jsx:237
▶ {type: 'answer', sdp: 'v=0\r\no=- 6935454125728879113 2 IN IP4 127.0.0.1\r
  \ns…:2\r\na=sctp-port:5000\r\na=max-message-size:262144\r\n'}
```

Il server in ascolto sulla *socket* dell'evento *answerCall*, emette un ulteriore evento di *callAccepted* per inoltrare al chiamato il segnale che gli ha inviato il chiamante:

```
//Il server è in ascolto dell'evento callUser
   socket.on("answerCall", (data) => {
     socket.to(data.to).emit("callAccepted", data.signal) //Invia a al chiamante tale evento il
cui payload è il segnale del chiamato
   })
```

Il chiamante in ascolto ancora sulla *socket* riceve l'evento di *callAccepted*, dunque, può iniziare anche lui la connessione chiamando il metodo *signal*:

```
socket.current.on("callAccepted", (signal) => { //Resto in ascolto dell'evento callAccepted,
dove nel payload vi è il signal del peer remoto
     peer.signal(signal)
   })
```

Comunicazione tra i peer

Terminata dunque la segnalazione, si è creato un canale di comunicazione tra i due peer che ora possono iniziare a scambiarsi gli stream di dati, entrambi i peer saranno sull'evento stream nel cui payload vi è lo stream del peer remoto:

```
  peer.on("stream", (stream) => { //Attendo l'evento stream, nel cui payload vi è lo stream del
peer remoto
```

```
      userVideo.current.srcObject = stream
      console.log(stream)
    })
```

Conclusione videochiamata

Per concludere la chiamata, dopo aver premuto sul bottone di chiusura, viene chiamata la funzione *leaveCall(),* dove viene emesso l'evento *chiudichiamata* e viene distrutta la connection peer:

```
//Chiusura chiamata
const leaveCall = () => {
  connectionRef.current.destroy()
  if(callAccepted){
  socket.current.emit("chiudichiamata", socketClient);
  }
  window.location.reload();
}
```

Il server riceve l'evento e ne emette un altro per l'altro peer comunicandogli che l'altro ha abbandonato la chiamata:

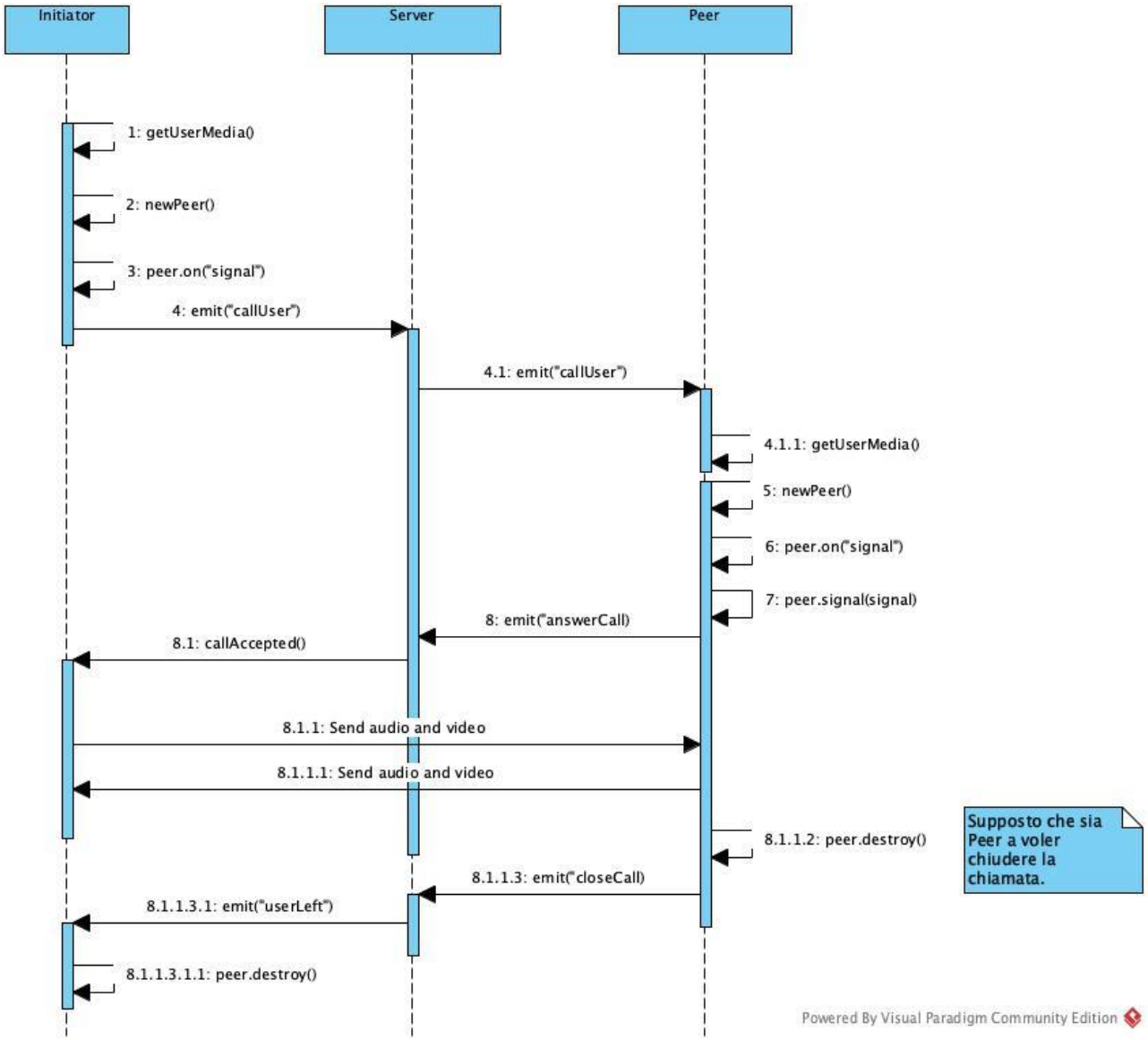
```
//Socket che comunica a uno dei peer che l'altro ha deciso di concludere la chiamata
  socket.on("chiudichiamata", (socketClient) => {
    socket.to(socketClient).emit("user left");
  })
```

L'altro peer riceve l'evento e distrugge la peer connection:

```
//Evento che comunica che l'altro peer ha lasciato la chiamata
    socket.current.on("user left",()=>{
      delete connectionRef.current;
      window.location.reload();
    });
```

Sequence Diagram

Si riporta di seguito il sequence diagram che descrive i passi di una videocall per una maggiore chiarezza:



Deployment

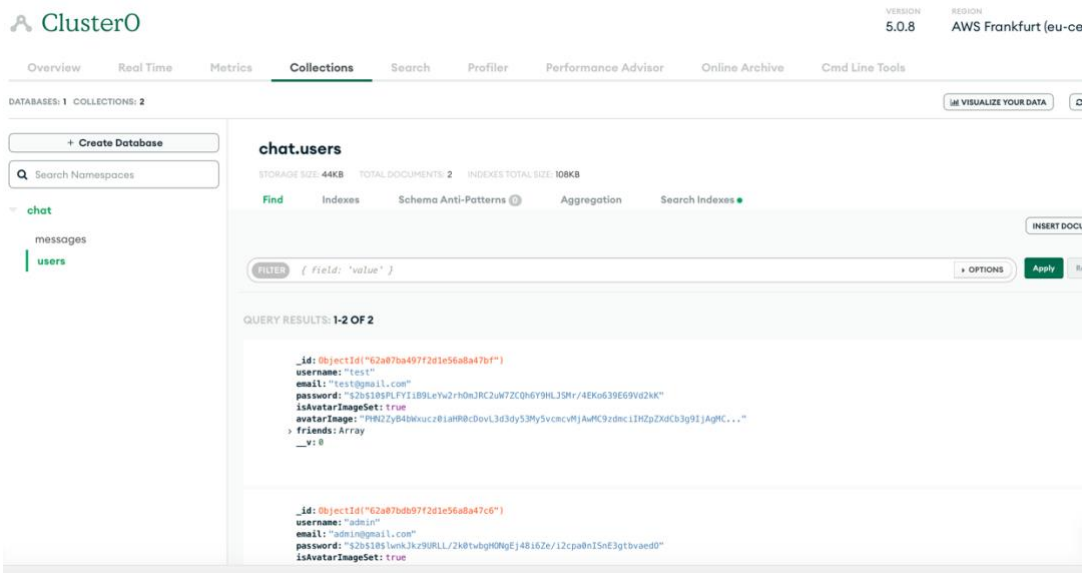
Per il deployment dell'applicazione si è utilizzato *Heroku,* ma è stato necessario apportare alcune modifiche che sono analizzate di seguito.

MongoDB Atlas

Innanzitutto, è stato necessario utilizzare il database non più in locale servendosi di MongoDB Atlas. Quest'ultimo è un database cloud che permette di gestire tutta la complessità del deployment su cloud service provider (AWS, Azure e GCP).

È stato dunque creato un nuovo database, **chat**, ricevendo una nuova URI che permette di connettersi a MongoDB. Si noti che tale URI è stata inserita nella sezione variabili su Heroku, in quanto il file *.env*, contenente le variabili d'ambiente, non viene caricato sulla piattaforma.

mongodb+srv://happyDB:happydb@cluster0.oo4z1.mongodb.net/chat?retryWrites=true&w=majority



Modifiche lato client

Per la creazione della build del client sono state effettuate due operazioni fondamentali.

1) È stata modificata la base delle API che è diventata:

```
export const host = "";
```

2) Aggiunto ai peer i parametri host, secure e port:

```
const peer = new Peer({
    initiator: true,
    trickle: false,
    stream: stream,
    secure: true, //Versione sicura
    host: "happyappesame.herokuapp.com", //Server host
    port: 443 //Server port (https)
})
```

A questo punto è stata realizzata la build del client utilizzando il seguente comando:

```
$yarn run build
```

Modifiche lato server

Le modifiche al lato server sono due:

1) Sostituzione di *nodemon* con *node* in quanto serve solo per lo sviluppo dell'app;
2) È stato inoltre modificato l'origine delle richieste, che adesso vengono accettate tutte ed è stato abilitato il parametro secure, quindi si accettano origini sicure:

```
const io = socket(server, {
   cors: {
    origin: ["*"],
         secure: true,
    methods: ["GET", "POST"],
    credentials: true,
   }
});
```

3) Importazione della cartella build del client nella directory del server.

Aggiungendo il seguente codice si può accedere all'applicazione all'indirizzo http://localhost/5000:

```
app.use(express.static(path.join(__dirname + "/public")))
```

Quindi l'app *React* e *Node.js* sono in esecuzione sulla stessa porta perché il comando che crea la build, una cartella contenente tutto il codice dell'applicazione *React*. Quindi le righe di codice (che devono eseguire qualcosa ed eseguono la build dentro il server) sopra faranno si che si prenderà tutto il contenuto della cartella build e lo visualizzerà quando si accede ad http://localhost/5000.

Dato dunque le API Node.js sono già disponibili sulla porta 5000, entrambe le applicazioni sono in esecuzione sulla stessa porta e non è necessario eseguire due comandi separati in terminali separati per avviare l'app *React* e quella *Node.js*.

Deployment

Heroku gestisce la distribuzione delle app mediante *Git*.

Dopo aver raggiunto il repository del server, è stata creata una nuova applicazione su *Heroku,* con il comando

```
$heroku create -a happyesame
```

Mediante il commando *add* sono stati aggiunti i file modifcati alla coda per il commit, ma non sono stati ancora salvati. Poi è stato effettuato il commit.

```
$git add.
```

```
$git commit “nome commit”
```

Per effettuare il deploy, quindi l’invio alla repository remota del codice, è stato utilizzato:

```
$git push Heroku master
```

Come risultato è stato ottenuto l’URL per accedere all’applicazione:

https://happyappesame.herokuapp.com