JTogether

Applicazioni e Servizi Web

Abu Ismam - 000725342 {ismam.abu@studio.unibo.it}
Gardini Lorenzo - 000725342 {lorenzo.gardini7@studio.unibo.it}

13 luglio 2021

Indice

Capitolo 1

Introduzione

L'applicazione proposta ha lo scopo di essere un social network per persone che vogliono organizzare attività di qualunque tipo, promuovendo gli incontri tra sconosciuti.

Si utilizzano vari servizi, tra i quali vi è la geo-localizzazione, se un utente non dovesse concedere il permesso di accedere alla sua posizione non potrà ricevere consigli sulle attività che si svolgeranno nelle sue vicinanze. Dopo aver partecipato ad un'attività è possibile scrivere nella chat dedicata e poter ricevere notifiche, sia online che offline, di eventuali modifiche e/o cancellazione dell'attività stessa.

Capitolo 2

Requisiti

Ogni utente per poter utilizzare il servizio deve creare un profilo personale (fornendo diverse informazioni: username, email) ed effettuare il login. Dalla home ogni utente può:

- visualizzare e/o cambiare le impostazioni del proprio profilo (anche eliminarlo)
- creare delle attività impostando un luogo, una data, un'ora ed una descrizione
- partecipare ad un'attività esistente reperibile tramite i consigliati e tramite ricerca
- cercare attività tramite nome dell'utente organizzatore, titolo o descrizione
- modificare una propria attività
- cliccare sulle notifiche per visualizzare le informazioni sull'attività
- interagire come le chat delle attività create e partecipate
- effettuare logout

2.1 Notifiche

Il servizio invia due tipi di notifiche diverse (che possono essere disabilitate):

- notifiche per segnalare una modifica (anche cancellazione) di un'attività a cui si stava partecipando
- notifiche per segnalare un'attività che avvengono vicino alla posizione dell'utente e che potrebbero interessargli

2.2 Chat

Il servizio crea una chat per tutte le attività. Appena un'utente entra nella chat riceve tutti i messaggi precedentemente inviati. Ogni messagggio della chat si compone di username del mittente, messaggio e data e orario di invio.

Capitolo 3

Design

L'intero progetto si suddivide in due sotto-progetti, uno per il lato front-end, chiamato *JTogether* ed uno per il back-end, chiamato *jtogether_ server*.

3.1 Front-end

Prima di iniziare a ragionare sullo scheletro del front-end si sono discussi gli usi e la vera e propria struttura dell'applicazione, in particolare si sono realizzati due schemi, un diagramma delle attività (Figura 3.4) e un diagramma dei casi d'uso (Figura 3.5). Il primo rappresenta il flusso delle attività all'interno della pagina, mentre con il secondo le azioni che ogni utente può eseguire. Per la grafica invece è svolto un design partecipativo, effettuato con la partecipazione di circa venti persone, che ha portato alla produzione e all'evoluzione di diversi **mockups** per le diverse schermate (sia in versione mobile che desktop).(Figura 3.1). Inoltre, l'applicazione è stata pensata per essere mobile-first.

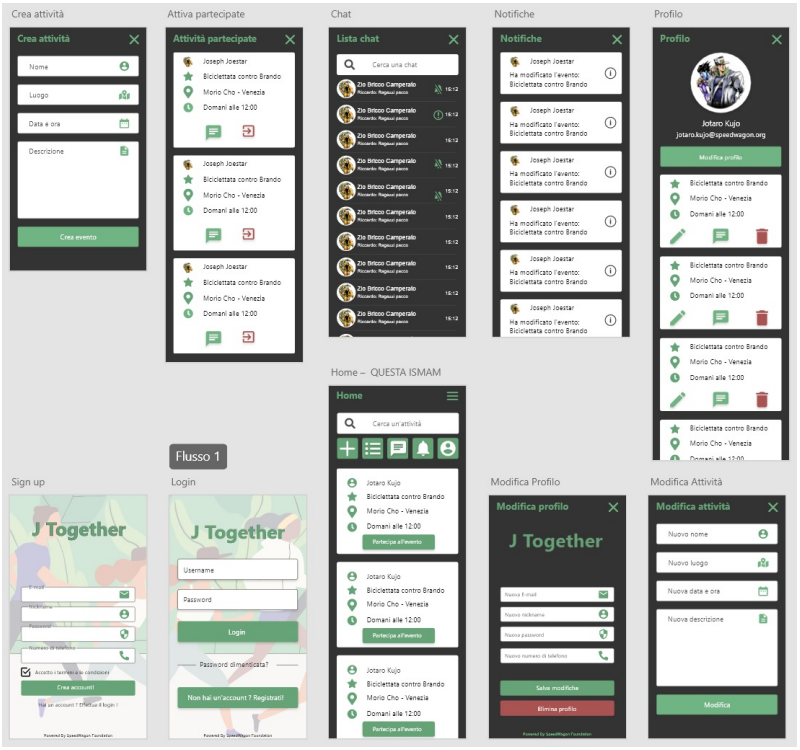

Figura 3.1: Parte di uno dei mockup

L'implementazione vera e propria è stata realizzata utilizzando Angular cercando di seguire il più possibile le linee guida e le best-practice offerte dalla documentazione ufficiale. L'argomento verrà approfondito maggiormente nel capitolo delle tecnologie utilizzate.

3.1.1 Componenti

La nostra applicazione prevede che ogni schermata sia composta da una gerarchia di componenti in modo da redere tutto più mantenibile, controllabile ed atomico. Inoltre, esistono alcuni componenti (come il navbar e il footer) che sono presenti nella maggioranza delle pagine.

Nelle pagine di *home, profilo, attività partecipate e notifiche* le informazioni sono rappresentate mediante delle *Cards*, queste sono personalizzate in base alle informazioni che si vogliono mostrare. Queste cards hanno in comune alcune parti di stile, ed è stato generalizzato in un file scss contenente dei *@mixins* che vengono richiamati dai propri file di stile.

La disposizione delle card all'interno della pagina è gestita dal componente *card-list* che, date un certo numero di card, le organizza in tutta la pagina,

in modo che siano equamente spaziate e distribuite in base alla grandezza del dispositivo utilizzato.

In figura 3.2 sono elencati tutti i componenti che sono stati creati.

3.1.2 Servizi

All'interno dell'applicazione sono presenti vari servizi, ognuno con un certo ruolo:

- **chat.service:** serve per gestire le chat, offre due metodi, uno per collegarsi alla stanza remota ed il secondo per uscire dalla stanza.
- **data.service:** contiene tutti i metodi per inviare e ricevere informazioni dal server, questa classe contiene:
  - **Metodi privati:** sono il *doGet* e il *doPost*, restituiscono entrambi una Promise del tipo di argomento specificato ed effettuano il relativo tipo di richiesta.
  - **Metodi pubblici:** sono tutti i vari tipi di richiesta, come *login*, *signup* ecc., questi metodi, al loro interno, chiamano il *doGet* o il *doPost*, e restituiscono una Promise dell'argomento specificato.

  Tutte le richieste e risposte vengono gestite utilizzando le Promise, ottenendo un codice pulito ed evitando la *Pyramid of doom*.
- **geolocation.service** gestice la geolocalizzazione dell'utente (dai permessi alla sua posizione attuale). Anche in questo. Contiene anche un metodo per calcolare la distanza tra due coordinate geografiche che viene utilizzata nelle attività mostrate nella Home.
- **jrouter.service:** contiene tutti i metodi per navigare la pagina, ogni metodo effettua il passaggio da un componente a quello richiesto.
- **local-storage.service:** questo servizio si occupa del *localStorage*, contiene metodi per salvare in memoria alcune informazioni utili in tutto il sito (come l'username dell'utente o il refresh token)
- **notification.service:** si occupa di instaurare un collegamento websocket con il server in modo da ricevere eventuali notifiche push.
- **snackbar.service:** questo servizio gestisce le *snackbar*. All'interno dell'applicazione esistono due tipi di snackbar: una che avvisa quando un'azione è stata completata con successo mentre l'altra per avvisare di eventuali errori.
- **tokens-manager.service:** questo servizio serve per gestire i due tipi di token che vengono utilizzati all'interno del sistema: **AccessToken** e **RefreshToken**. Sull'utilizzo dei token verrà approfondito di più nel capitolo delle tecnologie.

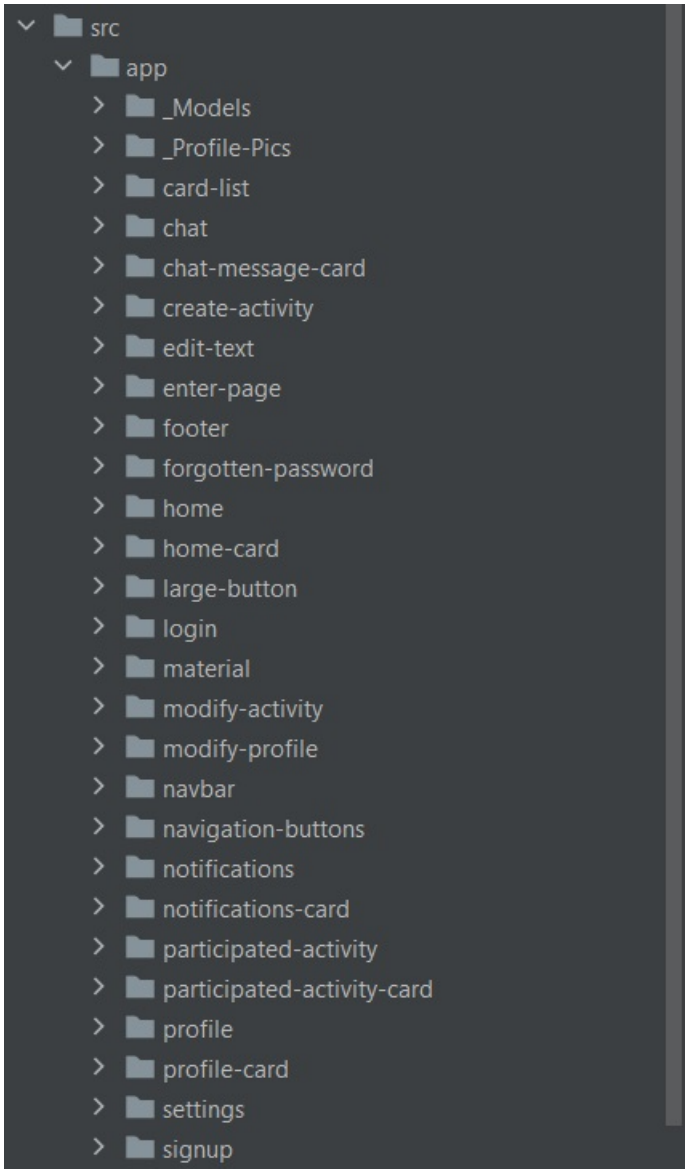

Figura 3.2: Lista delle componenti

- **utility.service:** Contiene dei metodi di utilità che vengono chiamati da più componenti, come per esempio il format della data.

3.1.3 Material, CSS ed SCSS

Google ha sviluppato per Angular una serie di componenti, pronti all'uso e supportati da un'esaustiva documentazione, chiamati Angular Material (`https://material.angular.io/`). Tra le componenti che abbiamo utilizzato maggiormente ci sono: **MatFormFieldModule, MatButtonModule, MatIconModule, MatInputModule e MatSnackBarModule**.
Angular offre tre scelte per realizzare il lato grafico di un'applicazione, SASS, CSS ed SCSS, noi abbiamo scelto SCSS. Questo ci ha permesso di utilizzare all'interno del codice i vari *@content*, *@mixin*, *@media* per semplificare e riutilizzare il codice. Ecco alcuni esempi:

- **mixins.scss:** questo file di stile contiene la maggior parte dei *@mixin* che utilizziamo, tra i più importanti ci sono *FlexRow* e *FlexCol* che si occupano di disporre o in orizzontale o in verticale il contenuto di un tag (come div o form), vi sono inoltre i mixins per la grandezza dello schermo attraverso l'uso di *@media screen*.

- **content-centerer-mixins.scss:** questo file di stile viene utilizzato per tutte le componenti come Login, SignUp, Create Activity ecc., hanno in comune il fatto di avere al proprio interno un form, questa classe ha alcuni mixin che si occupano di centrare e sistemare tutto il contenuti dei vari form, chiaramente non è necessario che i form contengano le stesse informazioni. All'interno di questo file è possibile richiamare mixin del file mixin.scss.

- **card-mixins.scss:** questo file di stile contiene alcuni mixin per lo stile delle card, come per i form non è necessario che il contenuto delle card sia uguale perché funzioni.

- **variables.scss:** questo file di stile contiene le variabili dei colori che sono utilizzati in tutti i file di stile.

3.2 Back-end

Il back-end è stato realizzando con un'architettura a strati in modo da rendere completamente trasparente i livelli sottostanti. Ogni richiesta che il server riceve attraversa quattro strati:

- **istanza express**: express filtra in base al path tutte le richieste che raggiungono il server smistandoli nei vari controller

- **controller**: il controller valida la richiesta (per esempio controlla che i parametri siano presenti e appartenenti al dominio corretto) e chiama il

model con i parametri della richiesta. Successivamente, risponde al client utilizzando il risultato prodotto dal model o un suo eventuale errore

- **model**: contiene la logica delle varie operazioni, effettua operazioni sui dati (e.g. applica l'hash alla password durante il signup)
- **Api**: contiene le chiamate all'istanza di MongoDB

Le richieste da parte dell'utente vengono divise in macro gruppi:

- gestione delle attività
- gestione dell'autenticazione
- gestione dell'utente
- gestione della chat

Anche express utilizza a sua volta un'architettura a strati, in cui ogni strato viene definito **middleware**. Ogni richiesta attraversa un numero variabile di questi livelli dipendentemente dalla possibilità di gestione della richiesta o della generazione di qualche errore. I primi due livelli permettono rispettivamente di convertire automaticamente il tipo del dato contenuto nella richiesta del client in un json e di permettere chiamate **CORS** (**Cross-Origin Resource Sharing**). Seguono poi i diversi middleware corrispondenti ai vari controller per poi terminare con due controller speciali, uno per la gestione degli errori che possono essersi generati nei passi precedenti e uno per gestire il *404 resource not found*.

3.2.1 Json Web Token

Sulla maggior parte delle richieste, per la validazione dell'autenticazione dell'utente, viene utilizzata la teconologia Json Web Token. Ogni volta che l'utente effettua il login vengono generati dal server due token: uno che ha una scadenza di circa sei mesi e che si chiama **refresh token**, mentre il secondo ha una durata di qualche decina di minuti e si chiama **access token**.

L'access token server per effettuare delle richieste al server, mentre il refresh token, come dice il nome, server per ottenere un nuovo access token una volta che il precedente è scaduto. La poca durata dell'access token deriva dal fatto che se per qualche motivo viene rubato può essere utilizzato solamente per qualche minuto.

Per generarli il server utilizza l'username dell'utente e una stringa segreta presente in un file del server. Ogni richiesta verso il server che necessita di aver effettuato precedentemente login o che utilizza il refresh token deve contenere nei campi dell'header della richiesta http in campo *Authorization* che contiene il corrispondente token.

Inoltre, il server mantiene internamente una lista di refresh token validi (che ha effettivamente generato) e che viene periodicamente pulito dai token scaduti. Questo ha lo scopo di rendere ancora più sicuro il sistema.

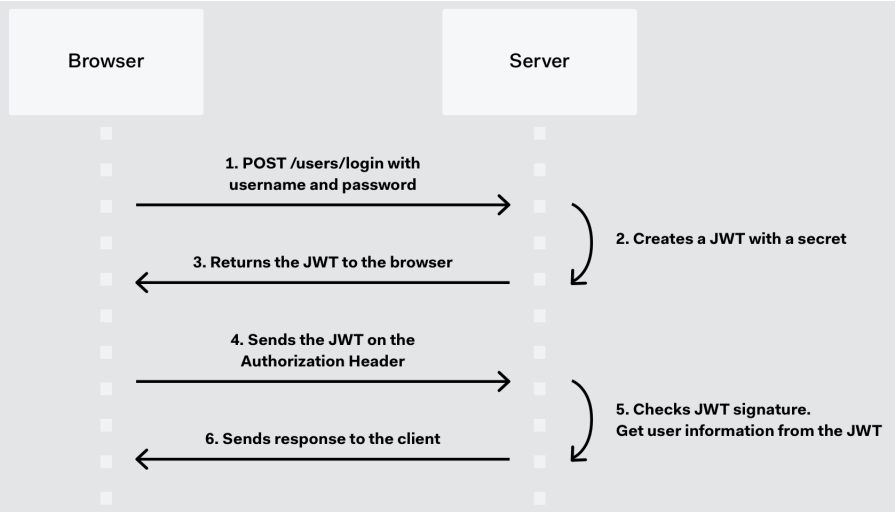

Figura 3.3: Diagramma di sequenza di JWT

3.2.2 MongoDB

Gli schemi utilizzati per il database rappresentano le varie informazioni che necessitano di essere salvate permanentemente:

- attività
- users
- messaggi della chat
- notifiche

Lo schema dei messaggi viene utilizzato per creare una lista di documenti innestati nell'attività che formano la chat dell'attività stessa. Stesso ragionamento vale per lo schema delle notifiche con gli utenti.

Abbiamo deciso di ridondare l'informazione delle partecipazioni/creazioni delle attività, presente nello schema delle attività, all'interno dello schema dell'utente. Questo serve per semplificare enormemente la complessità delle query che permettono di ottenere le attività dall'utente.

3.2.3 Geolocation

Le attività della home sono quelle vicine alla posizione dell'utente nel raggio di 20km circa. La posizione in coordinate geografiche di un'attività viene calcolata nel controlle di queste ultime grazie ad un servizio esterno gratuito di *geocoding* chiamato **LocationIQ** per trasformare automaticamente il luogo impostato dall'utente in coordinate geografiche. Le coordinate vengono salvate come campo dell'attività di tipo *2dsphere* per poter poi calcolare la distanza.

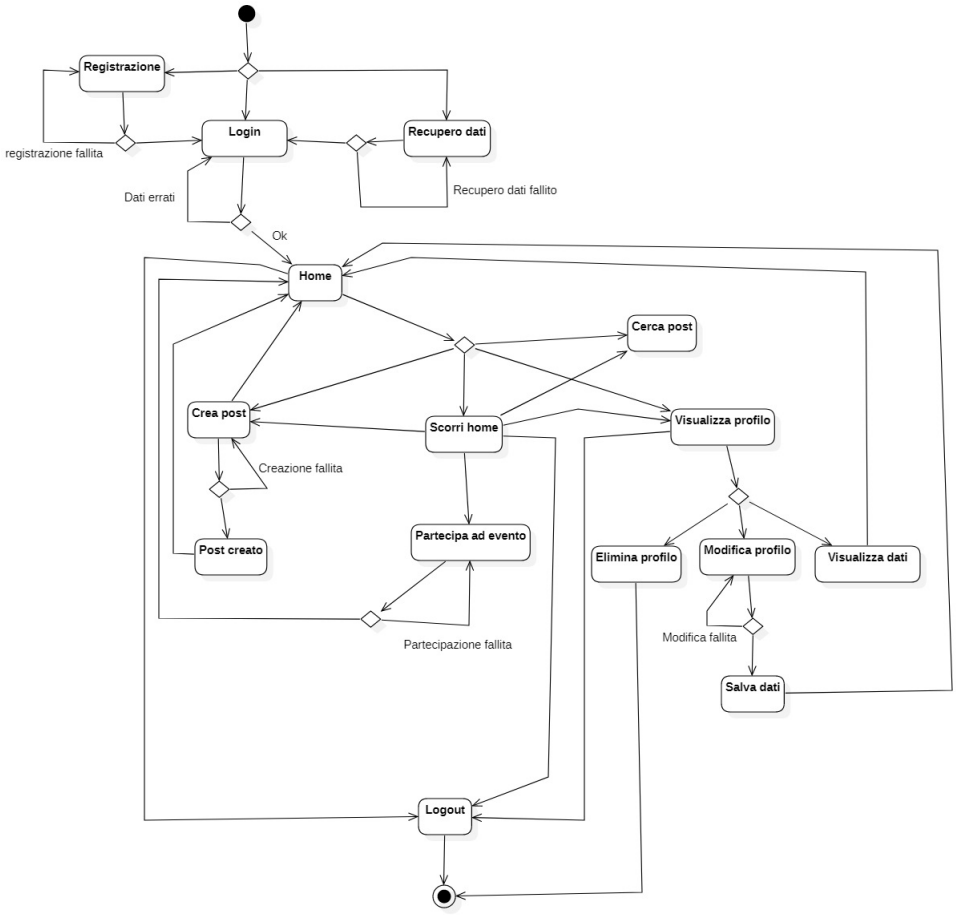

Figura 3.4: Diagramma delle attività

3.2.4 Socket.IO

Per gestire le notifiche è stato utilizzato utilizzato il framework **Socket.IO** per gestire gli eventi push. Viene utilizzato sia lato client, in cui si instaura una connessione e si notifica al server che si è online, e lato server, in cui si gestiscono i messaggi delle chat e le notifiche riguardanti la modifica/cancellazione delle attività. Ogni volta che un'attività subisce delle modifiche il controller prende la lista delle persone da notificare; poi successivamente controlla, per ciascuna di queste, se l'utente è attualmente online oppure no. Se è online il server genera una notifica che viene ricevuta in diretta dall'utente come notifica del browser, altrimenti viene creata una nuova istanza di notifica nel documento di quello specifico utente. Questa casella di notifiche si svuoterà in automatico una volta che l'utente vi avrà fatto l'accesso.

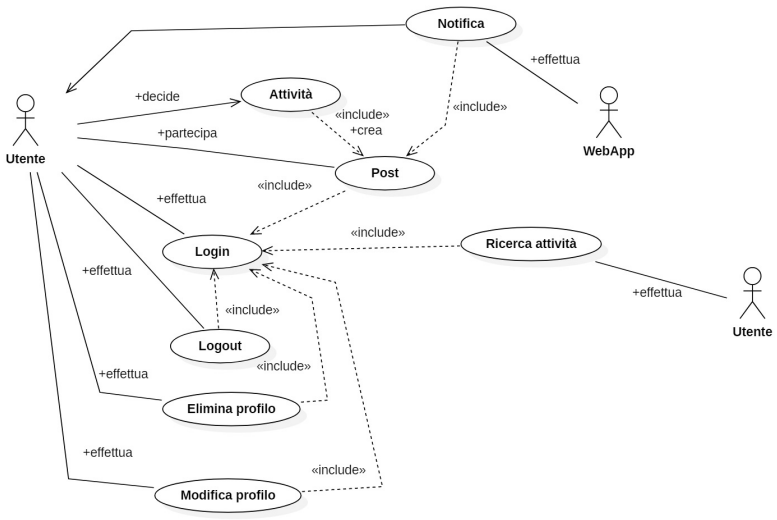

Figura 3.5: Diagramma dei casi d'uso

Capitolo 4

Tecnologie

L'applicazione utilizza le seguenti tecnologie:

- **Angular**: Framework per lo sviluppo dell'applicazione
  - TypeScript: Linguaggio di programmazione ad oggetti, Super-set di JavaScript
  - Material Angular: Libreria per Material Design in Angular
  - SCSS: Estensione di CSS che ne facilita il riuso
- **Express**: Framework per applicazioni web
  - NodeJS: Runtime JavaScript guidato da eventi asincroni
  - JavaScript: linguaggio orientato agli oggetti
- **MongoDB**: Database non relazionale, basato su documenti
  - Mongoose: Libreria di object modelling per MongoDB
- **Socket.IO**: framework per gestire eventi push tra client e server
- **LocationIQ**: servizio che permette di manipolare localizzazioni e coordinate spaziali (https://locationiq.com/)

Capitolo 5

Codice

Tutte le chiamate asincrone sono gestite sia lato client che lato server con le promise fornite da JavaScript

5.1 Front-End

5.1.1 Modelli delle varie entità

Activity

```
export interface Activity {
    profile_pic: string;
    id: string;
    participants: string[];
    name: string;
    creator_username: string;
    location: string;
    date_time: string;
    description: string;
    geolocation: Geolocation;
    chat: Message[];
  }
```

User

```
  export interface User{
    profile_pic: string;
    created_activities: string[];
    participated_activities: string[];
    username: string;
    email: string;
```

```
    access_token: string;
    refresh_token: string;
    notifications: Notification[];
  }
```

Message

```
  export interface Message {
    message: string;
    time_stamp: Date;
    username: string;
  }
```

Notification

```
  export interface Notification {
    message: string;
    date_time: string;
    activity_name: string;
    activity_owner: string;
  }
```

Geolocation

```
  export interface Geolocation{
    latitude: number;
    longitude: number;
  }
```

5.1.2 Token Manager Service

Metodo per ottenere l'Access Token

```
public async getAccessToken(): Promise<string> {
    const jwt = new JwtHelperService();
    const refreshToken = this.localStorage.getRefreshToken();

    if (!refreshToken) {
      return Promise.reject({message: 'refresh token non presente'});
    }

    if (jwt.isTokenExpired(refreshToken, DELTA_EXPIRATION_TIME_SECONDS)){
      return Promise.reject({message: 'refresh token scaduto'});
```

```
    }

    if (this.accessToken && !jwt.isTokenExpired(this.accessToken,
                                    DELTA_EXPIRATION_TIME_SECONDS)){
      return Promise.resolve(this.accessToken);
    }
    return this.dataService.accessToken(refreshToken)
      .then(t => this.accessToken = t.access_token);
  }
```

Metodo per verificare di avere effettuato il login

```
isLoggedIn(isOk: () => void): void{
    const refreshToken = this.getRefreshToken();
    const username = this.localStorage.getUsername();
    if (!(!refreshToken || !username || this.jwt.isTokenExpired(refreshToken)
    || this.jwt.decodeToken<{username: string}>
    (refreshToken).username !== username)){
      isOk();
    } else {
      this.router.goLogin();
    }
}
```

5.1.3 Data Service

Metodo generale da cui effettuare una richiesta POST

```
private doPost<X>(url: string, body?: object, token?: string): Promise<X> {
   return this.httpClient.post<X>(url, body,
        {headers : { Authorization : 'bearer ' + token},
        responseType : 'json'})
        .toPromise();
}
```

Metodo generale da cui effettuare una richiesta GET

```
private doGet<X>(url: string, token: string | null, params?: HttpParams):
Promise<X> {
    const options = token ? {params,
    headers: {Authorization : 'bearer ' + token}} : {params};
    return this.httpClient.get<X>(url, options).toPromise();
}
```

Esempio di utilizzo del metodo doPost

```
getNearActivities(position: Geolocation, accessToken: string):
Promise<Activity[]> {
    return this.doGet<Activity[]>(
      this.serverUrl + this.userPath + 'get-near-activities',
          accessToken,
          new HttpParams()
            .append('longitude', position.longitude.toString())
            .append('latitude', position.latitude.toString()));
}
```

5.1.4 Notifications

Metodo per creare una socket

```
createSocket(username: string): void{
    this.socket.disconnect();
    this.socket.connect();
    this.socket.on('connect', () =>
        this.socket.emit('registration', username));
    this.socket.fromEvent<string>(username)
        .subscribe(m => new Notification(m));
}
```

5.2 Back-End

5.2.1 db-model

Activity Schema

```
const activitySchema = mongoose.Schema({
    creator_username : { type: String, required : true, index: 'text'},
    name: { type: String, required: true, index: 'text'},
    description: { type: String, required: true, index: 'text'},
    date_time: { type: Date, required: true},
    participants : {type : [String], default: [], required:true },
    location: {type: String, required: true, index: 'text'},
    geolocation: {type: [Number], index: '2dsphere', required: true},
    profile_pic : {type: String, unique: false, required: true},
    chat: {type: [messageSchema], default: [],required: true}
})
```

User Schema

```
const userSchema = mongoose.Schema({
    username : {type: String, unique: true, required: true},
    email : {type: String, unique : true, required : true},
    created_activities : {type: [String], default: [], required : true},
    participated_activities : {type: [String], default: [], required : true},
    hash : {type: String, required: true},
    notifications: {type: [notificationSchema], default: [], required: true},
    profile_pic : {type: String, unique: false, required: true}
})
```

Message

```
const messageSchema = mongoose.Schema({
    _id: false,
    message: {type: String, required: true},
    time_stamp: {type: Date, required: true, default: new Date()},
    username: {type: String, required: true}
})
```

Notification Schema

```
const notificationSchema = mongoose.Schema({
    _id: false,
    activity_name: {type: String, required: true},
    activity_owner: {type: String, required: true},
    message: {type: String, required: true},
    date_time: {type: Date, required: true, default: new Date()}
})
```

5.2.2 socket.io-controller

Socket.IO

```
this.io.on('connection',(socket) =>{
            socket.on('registration', username => {
                this._socketToUser.set(socket,username)
                this._userToSocket.set(username,socket)
            })
            socket.on('disconnect',() => {
                this._userToSocket.delete(this._socketToUser.get(socket))
                this._socketToUser.delete(socket)
            })
        })
```

5.2.3 jwt

Json Web Token

```
async function authenticateJWT (req, res, next) {
    const authHeader = req.headers.authorization;
    if (authHeader) {
        const token = authHeader.split(' ')[1];

        jwt.verify(token, config.secret, (err, user) => {
            if (err) {
                return res.sendStatus(403);
            }
            req.user = user;
            req.token = token
            next();
        });
    } else {
        res.sendStatus(401);
    }
}
```

5.2.4 error-handler

Gestione degli errori

```
function errorHandler(err, req, res, notUsed) {
    if (typeof (err) === 'string') {
        return res.status(400).json({ message: err })
    }

    ...

    return res.status(500).json({ message: err.message })
}

function resourceNotFoundHandler(req, res){
    return res.status(404).json({ message: 'Resource not found' })
}
```

Capitolo 6

Test

I test back-end sono stati effettuati provando le varie funzionalità grazie a **Postman** che permette di automatizzare le richieste.

Capitolo 7

Conclusioni

In conclusione, conveniamo sul fatto che questo progetto ci abbia aiutato a comprendere e incrementare notevolmente le nostre conoscenze in ambio web su paradigmi di buona progettazione ed implementazione, nello specifico nelle tecnologie indicate sopra.