Compte-Rendu : Socket Chat

Réalisé par Alexandre Bentz

Sommaire

I. Contexte professionnel

Il est dans l'intérêt d'une entreprise de favoriser la communication en son sein afin de faciliter la circulation de l'information, primordiale dans la prise de décision de tous les acteurs. L'application de chat en temps réel est destinée à fournir un nouveau moyen d'échanger les informations entre les salariés. Dans le cadre du projet, les fonctionnalités souhaitées étaient : l'inscription et la connexion de chaque collaborateur, un salon de discussion général et la possibilité d'envoyer des messages privés aux différents contacts connectés.

II. Tableau comparatif des solutions de développement mobile

	Application hybride	**Application Native**
Concept	Assure l'interopérabilité de l'application avec des librairies qui utiliseront les composants natifs selon le terminal	En interaction directe avec les composants natifs de l'environnement
Langages	Web (HTML, CSS, Javascript et dérivés…)	Java pour Android, Swift pour iOS
Avantages	- Un seul code pour l'application : temps de développement et de maintenance réduit - Développement facilité par l'accessibilité des différents composants	- Solution performante et fluide (animation 3D…) - Mises à jour régulières, stables et offrant de nombreuses fonctionnalités
Limites	- Performance moindre - Solution souvent open-source, donc mises à jour régulières présentant parfois des bugs non résolus par la communauté	- Deux application, cout et temps de maintenance lourd et interopérabilité restreinte

La solution retenue a donc été Ionic, framework se basant sur Angular et TypeScript pour créer des applications hybrides directement en interaction avec les éléments natives de terminaux. L'application sera mise en relation avec un serveur Socket.Io, se basant sur du JavaScript pour assurer la communication des évènements en temps réel.

III. Déroulement du projet

A. MISE EN PLACE DE LA TRANSMISSION DES MESSAGES AU SERVEUR

Afin de mettre en place l'envoi des messages, la méthode sendMessage() doit émettre au serveur l'évènement « general-message » avec le message en paramètre et doit réinitialiser la variable message à vide.

Voici le code de la méthode, située dans le fichier **home.ts** :

```
sendMessage() {
    this.socket.emit('general-message', this.message);
    this.message = '';
  }
```

B. MISE EN PLACE DE LA RÉCEPTION DES MESSAGES DU SERVEUR

Afin de mettre en place la réception des messages, la méthode getMessage() doit contenir un observable qui va surveiller l'envoi d'évènement « message » par le serveur et émettre un observateur lors que cela arrive.

Voici le code de la méthode, située dans le fichier **home.ts :**

```
getMessages() { // Reception des messages
    // Création de l'observateur du serveur
    let observable = new Observable(observer => {
      // Observation du serveur : réception des messages du socket
      this.socket.on('g-message', (data) => {
        // On place dans l'objet observer l'objet message retourné par
notre serveur Socket
        observer.next(data);
      });
    });
    return observable; // Retour de l'observateur
  }
```

Afin de surveiller les évènements captés par la méthode, il faut « souscrire » à la méthode dans le constructeur. À chaque nouvelle arrivée de données envoyées par le serveur, on va ainsi ajouter le message qui nous a été transmis dans la liste des messages.

Voici le code de la méthode, située dans le constructeur du fichier **home.ts** :

```
this.getMessages().subscribe( message => {
// Surveillance des changements
      this.messages.push(message);
    });
```

C. MISE EN PLACE DE L'AFFICHAGE DES MESSAGES

Afin de mettre l'affichage des messages, la réalisation de la page HTML devait gérer les possibilités ou le message provenait de l'utilisateur ou d'une autre personne. Ainsi, j'ai implémenté une grille dans laquelle on boucle les lignes sur la liste des messages qui affiche deux types de colonnes en fonction de l'expéditeur. Ces deux colonnes se différencient par leur positionnement et leur class Angular, mais elles affichent toutes les deux l'expéditeur du message, le contenu du message et l'horodatation du message.

Voici le code de la page **home.html** qui concerne cet affichage :

```
<ion-content no-bounce padding>
  <ion-grid>
    <!-- Boucle qui affiche l'ensemble des messages recus, last représente la
fin du tableau messages -->
    <ion-row *ngFor="let message of messages; let last = last">
      <!-- Si le message vient d'un autre utilisateur -->
      <ion-col col-3 *ngIf="message.from !== pseudo"
class="message" [ngClass]="'other_message'">
        <span class="user_name"><b>{{ message.from }} </b> : </span><br>
        <span> {{ message.text }}</span>       <div class="time">
{{ message.created | date: 'dd/MM HH:mm' }} </div>
      </ion-col>

      <!-- Si le message provient de l'utilisateur -->
      <ion-col offset-4 *ngIf="message.from === pseudo"
class="message" [ngClass]=« 'my_message'">
        <span class="user_name"><b> {{ message.from }} </b>:</span><br>
        <span> {{ message.text }}</span>
        <div class="time"> {{ message.created | date: 'dd/MM HH:mm' }} </div>
      </ion-col>
      {{ last ? fin() : ''}}
      <!-- Amène le curseur vers le dernier message  -->
    </ion-row>
  </ion-grid>
</ion-content>
```

En ce qui concerne le fichier home.ts, il était donc nécessaire de définir la méthode fin, qui permet de descendre automatiquement le curseur au niveau du dernier message au cas ou ils ne rentreraient plus dans la taille de l'écran.

Voici le code de la fonction fin() située dans le fichier **home.ts :**

```
fin() {
    this.content.scrollToBottom(0);
  }
```

D. MISE EN PLACE DU DESIGN DU SALON

Pour compléter l'affichage des messages, on ajoute au salon une barre contenant une zone d'écriture et un bouton afin de permettre à l'utilisateur d'entrer et d'envoyer son message. Le paramètre keyup.enter de la zone de texte permet d'appeler la fonction au cas ou l'utilisateur appuie sur la touche « Retour » de son mobile afin d'améliorer l'ergonomie.

Voici le code ajouté au fichier **home.html** :

```
<ion-footer>
  <ion-item no-lines class="footer">
    <ion-textarea type="text" placeholder="Entrer votre
message" [(ngModel)]="message" (keyup.enter)="sendMessage()"></ion-textarea>
    <button ion-button round outline item-end icon-only
color="primary" (click)="sendMessage()"
      [disabled]="message === ''">
      <ion-icon class="send" ios="ios-send" md="md-send"></ion-icon>
    </button>
  </ion-item>
</ion-footer>
```

Ensuite, l'ensemble de la page a été mise en forme grâce au fichier home.scss qui permet d'appliquer les règles de style du langage CSS au fichier home.html.

Voici le code du fichier **home.scss** :

```scss
page-home {

  .user_name {
    color: #636363;
  }

  .message {
      padding: 7px !important;
      border-radius: 10px !important;
      margin-bottom: 4px !important;
  }

  .my_message {
    background: rgb(123, 197, 236) !important;
    max-width: 75%;
  }

  .other_message {
    background: #dcdcdc !important;
    margin-right: 100px;
  }

  .time {
      color: #636363;
      margin-top: 15px;
      float: right;
      font-size: 10px;
  }

  .footer button ion-icon {
    margin: 10px;
  }

  .footer {
    border-radius: 10px;
    background-color: #ddd;
    margin: 3px;
    margin-left: 17px;
    width: 90%;
  }

  ion-textarea {
    height: 35px;
    font-size: 15px;
  }

}
```

E. CRÉATION DE LA BASE DE DONNÉES

Script de création de la base de données sockchat et des tables demandées :

```
CREATE DATABASE sockchat;
USE sockchat;
CREATE TABLE `membres` (
  `mail` varchar(50) NOT NULL,
  `pseudo` varchar(30) DEFAULT NULL,
  `mdp` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`mail`),
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
CREATE TABLE `etre_ami`(
    `mail1` varchar(100) NOT NULL,
    `mail2`varchar(100) NOT NULL,
    PRIMARY KEY (`mail1`,`mail2`),
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

F. CRÉATION DE LA PAGE D'INSCRIPTION

La page d'inscription doit se présenter sous forme de formulaire, lorsqu'il sera validé un signal au serveur sera transmis afin d'enregistrer les informations dans la base de données. Elle doit être accessible à partir d'un bouton dans la page de connexion. Différents champs sont requis : pseudo, adresse mail et mot de passe. La gestion des erreurs est prise en charge grâce aux Validateurs fournis par Angular, qui permettent ainsi de distinguer les différents types d'erreurs et donc d'afficher des messages en conséquence.

Voici le code la page **inscription.html** :

```
<ion-content padding>

  <form [formGroup]="validator" (ngSubmit)="inscription()">
    <ion-list>

      <ion-list-header>Pseudo</ion-list-header>
      <ion-item>
        <ion-input placeholder="Votre Pseudo" [(ngModel)]="pseudo"
formControlName="pseudo"></ion-input>
      </ion-item>

      <ion-item *ngIf="( validator.get('pseudo').hasError('minlength') ||
validator.get('pseudo').hasError('required') ||
validator.get('pseudo').hasError('pattern') ||
validator.get('pseudo').hasError('maxlength') ) &&
validator.get('pseudo').touched">
        <div class="error"
*ngIf="validator.get('pseudo').hasError('required') &&
validator.get('pseudo').touched">
          Veuillez entrer un pseudo
        </div>
        <div class="error"
*ngIf="validator.get('pseudo').hasError('minlength') &&
validator.get('pseudo').touched">
          5 carctères minimum
        </div>
        <div class="error"
*ngIf="validator.get('pseudo').hasError('maxlength') &&
validator.get('pseudo').touched">
          15 caractères maximum
        </div>
        <div class="error" *ngIf="validator.get('pseudo').hasError('pattern')
&& validator.get('pseudo').touched">
          Pseudo non valide
        </div>
      </ion-item>

      <ion-list-header>Adresse Mail</ion-list-header>
      <ion-item>
        <ion-input placeholder="example@domain.com"
type="email" [(ngModel)]="mail" formControlName="mail"></ion-input>
      </ion-item>
```

```
 <!-- Message qui s'affiche en cas de non validité d'un condition de
validation -->
      <ion-item *ngIf="( validator.get('mail').hasError('required') ||
validator.get('mail').hasError('pattern') ) &&
validator.get('mail').touched">
        <div class="error" *ngIf="validator.get('mail').hasError('required')
&& validator.get('mail').touched">
          Veuillez entrer une adresse email
        </div>
<div class="error" *ngIf="validator.get('mail').hasError('pattern') &&
validator.get('mail').touched">
          Adresse email non valide
        </div>
      </ion-item>

<ion-list-header>Mot de passe</ion-list-header>
      <ion-item>
        <ion-input placeholder="••••••••" type="password" [(ngModel)]="mdp"
formControlName="mdp"></ion-input>
      </ion-item>
<ion-item *ngIf="( validator.get('mdp').hasError('minlength') ||
validator.get('mdp').hasError('required') ) && validator.get('mdp').touched">
        <div class="error" *ngIf="validator.get('mdp').hasError('required')
&& validator.get('mdp').touched">
          Veuillez entrer un mot de passe
        </div>
<div class="error" *ngIf="validator.get('mdp').hasError('minlength') &&
validator.get('mdp').touched">
          8 carctères minimum
        </div>
<div class="error" *ngIf="validator.get('mdp').hasError('pattern') &&
validator.get('mdp').touched">
            Requis : majuscules, minuscules et chiffres
          </div>
      </ion-item>
<button ion-button full type='submit' [disabled]="!
validator.valid">S'inscrire</button>

    </ion-list>
  </form>
</ion-content>
```

En ce qui concerne le fichier inscription.ts, les différentes variables sont instanciées et les Validateurs sont définis dans le constructeur. Chaque champ possède des conditions de validité différentes, voici ceux sélectionnés :

- Pseudo : requis, 5 caractères minimum, 15 caractères maximum, composés de lettre majuscule et minuscule uniquement
- Adresse mail : requis, format valide
- Mot de passe : requis, 8 caractères minimum, composé de minuscules, majuscules et chiffres

Voici le code des parties concernées dans le fichier **inscription.ts** :

```
mail: string = "";
 pseudo: string = "";
  mdp: string = "";
  validator: FormGroup;

  constructor(
   public navCtrl: NavController,
    public navParams: NavParams,
    public socket: Socket,
    public formBuilder: FormBuilder,
    public alertCtrl: AlertController
   ) {
    this.validator = this.formBuilder.group({
      pseudo: ['', Validators.compose([
        Validators.required,
        Validators.minLength(5),
        Validators.maxLength(15),
        Validators.pattern('[a-zA-Z]*')
      ])],
      mail: ['', Validators.compose([
        Validators.required,
        Validators.pattern('^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+.[a-zA-Z0-9-.]+
$')
      ])],
      mdp: ['', Validators.compose([
        Validators.minLength(8),
        Validators.pattern('^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])[a-zA-Z0-9]+
$'),
        Validators.required
     ])]
    });
  }
```

G. MISE EN PLACE DE L'INSCRIPTION AVEC LA BASE DE DONNÉES

Du côté du serveur, il faut réceptionner les différentes informations envoyées par l'utilisateur, puis vérifier que l'utilisateur n'est pas déjà présent dans la base de données. Après cette vérification, on ajoute l'utilisateur à la base de données puis on envoie le signal comme quoi la requête à bien été effectuée, sinon envoie un signal pour prévenir que l'utilisateur qu'il a déjà un compte, l'inscription n'est alors pas effectuée.

Voici le code de la méthode concernée située dans le fichier **index.js** :

```
socket.on('inscription', (infos) => { // Inscription d'un utilisateur
    db.query({ // On vérifie que le mail n'est pas déjà présent dans la BDD
      sql: 'SELECT * FROM membres WHERE mail=?', // Requete SQL
      values: [infos.mail] // Paramètres
    }, function (error, results, fields) {
      if (error) {
        throw error;
      }
      if (results.length > 0) { // Si un ligne est retourné, erreur car
utilisateur déjà existant
        io.sockets.connected[socket.id].emit('status-inscription',
{ status: false });
      } else { // Sinon, inscription de l'utilsiateur
           db.query({
           sql: 'INSERT INTO membres SET ?',
           values: [{
              mail: infos.mail,
            pseudo: infos.pseudo,
              mdp: infos.mdp
              }]
            }, (error, results, fields) => {
              if (error) {
                throw error;
              }
              if (results.affectedRows > 0) {
                io.sockets.connected[socket.id].emit('status-inscription',
{ status: true });
              }
            });
          });
        });

      }
    });
  });
```

Du côté de l'application, il faut transmettre les informations en envoyant l'évènement d'inscription au serveur, et ce seulement si tous les champs sont valides. Il faut ensuite écouter le retour du serveur et en fonction du statut de l'inscription (true, false), il faudra signaler le résultat à l'utilisateur.

Voici le code de la méthode inscription() située dans le fichier **inscription.ts :**

```
inscription() {
    if (this.validator.valid) { // Inscription possible suelement si le
formulaire est valide
      this.socket.connect();
      this.socket.emit('inscription', { mail: this.mail, pseudo:
this.pseudo, mdp: this.mdp });

      this.socket.once('status-inscription', (verif) => {
        let affichage;
        if (verif['status'] == true) { // Si l'inscription est réussi,
redirection vers la page de connexion
          affichage = {
            title: "Succès",
            message: "Création du compte réussie !",
            buttons: [{ text: 'OK', handler: () => {
              this.navCtrl.goToRoot({});
            }}]
          };
        } else { // Si l'inscription n'a pas pu être effectuée, affichage
de l'erreur
          affichage = {
            title: "Echec",
            subTitle: "Le compte n'a pu être crée, l'adresse mail est déjà
utilisée",
            buttons: ['OK']
          };
        }
        this.alertCtrl.create(affichage).present();
      });
    } else {

      this.socket.disconnect();
      this.alertCtrl.create({
        title: "Echec",
        subTitle: "Inscription impossible, champs non valides",
        buttons: ["OK"]
      }).present();
    }
  }
```

H. MISE EN PLACE DE LA CONNEXION AVEC LA BASE DE DONNÉES

La vue de la page de connexion devait contenir deux champs, afin de permettre à l'utilisateur d'entrer son adresse mail et son mot de passe afin de s'identifier. Ces deux zones de textes sont liées avec des variables respectives qui seront transmises au serveur lorsque le bouton de validation sera activé. On y retrouve par ailleurs le bouton permettant d'accéder à la page d'inscription.

Voici le code de la page **connexion.html** :

```
<ion-content padding no-bounce>
  <ion-grid style="height: 100%;">
      <ion-row justify-content-center align-items-center class="" style="height:
100%;">
        <ion-col col-6>
          <ion-img width="250px" height="250px" class="mx-auto d-block" src="../../
assets/imgs/logo.png"></ion-img>
          <form (ngSubmit)="connexion()">
            <ion-card style="padding-bottom: 25px;">
              <ion-card-title>
                <ion-title>Connexion</ion-title>
              </ion-card-title>
              <ion-item>
                <ion-label floating color="primary" for="email">Email</ion-label>
                <ion-input type="email" [(ngModel)]="mail" name="mail"></ion-input>
              </ion-item>
              <ion-item>
                <ion-label floating color="primary" for="password">Password</ion-
label>
                <ion-input type="{{ passType }}" [(ngModel)]="password"
name="password"></ion-input>
             </ion-item>
            </ion-card>
            <div class="row pt-5">
              <div class="col-5 mx-auto">
                <button ion-button full outline type="submit" [disabled]="mail ==
''">Connexion</button>
              </div>
            <div class="col-5 mx-auto">
              <button ion-button full outline
type="button" (click)="inscription()">Inscription</button>
            </div>
          </div>
        </form>
      </ion-col>
    </ion-row>
  </ion-grid>
</ion-content>
```

La méthode de connexion présente sur le serveur doit-elle effectuer une requête dans la base de données afin de vérifier la présence de l'adresse mail transmise puis la conformité du mot de passe qui lui est associé. En cas de succès des deux vérifications, les informations de l'utilisateur sont enregistrées dans un tableau, variable propre au serveur et non au client, puis un signal est envoyé à l'application pour validé la connexion et transmettre le pseudo. En cas d' échec, un signal est aussi envoyé avec un message d'erreur pour expliquer à l'utilisateur le problème rencontré (utilisateur inexistant, mot de passe incorrect).

Voici le code de la méthode concernée située dans le fichier **index.js** :

```
socket.on('connexion', (infos) => {
    db.query({
      sql: 'SELECT * FROM membres WHERE mail=?',
      values: [infos.mail]
    }, function (error, results) {
      if (error) {
        throw error;
      }
      if (results.length == 0) {
        io.sockets.connected[socket.id].emit('status-connexion', { status:
false, error: "Utilisateur inexistant" });
      } else {
        if (infos.password == results[0].mdp) {
           socket.nickname = results[0].pseudo;
           membres.push({ id: socket.id, pseudo: socket.nickname }); // Ajout
du membres dans la liste des utilisateurs connectés
                console.log(socket.nickname + " connecté à " + new Date());
                io.sockets.connected[socket.id].emit('status-connexion',
{ status: true, pseudo: socket.nickname }); // Emission du résultat
uniquement au client concerné
                socket.broadcast.emit('user-connected', socket.nickname + '
s\'est connecté');
              });
            });
          } else {
            io.sockets.connected[socket.id].emit('status-connexion', {status:
false, error: "L'utilisateur esr déjà connecté"});
          }
        } else {
          io.sockets.connected[socket.id].emit('status-connexion', { status:
false, error: "Mot de passe incorrect" });
        }
      }
    });
  });
```

La méthode connexion() s'occupe d'envoyer l'évènement de connexion au serveur avec l'adresse mail et le mot de passe en paramètres. On crée l'observateur pour récupérer la réponse du serveur, puis on gère le retour : si la connexion est réussie, on redirige l'utilisateur vers le salon général et passant le pseudo en paramètre, que l'on a récupéré dans la réponse du serveur.

Voici le code de la méthode connexion() située dans le fichier connexion.ts :

```
connexion() {
    this.socket.connect();
    this.socket.emit('connexion', { mail: this.mail, password:
this.password });
    this.socket.once('status-connexion', verif => {
      if (verif.status == true) {
        this.navCtrl.push(TabsPage, { pseudo: verif.pseudo });
      } else {
        this.alertCtrl.create({
          title: "Echec de la connexion",
          subTitle: verif.error,
          buttons: ['OK']
        }).present();
      }
    });
  }
```

I. GESTION DE L'AFFICHAGE DES CONTACTS

En ce qui concerne l'affichage des contacts du côté serveur, l'évènement de récupération de la liste doit être et doit envoyer une réponse accompagnée de la liste des membres, générale au serveur.

Voici le code de la méthode concernée située dans le fichier **index.js** :

```
socket.on('get-contacts', () => {
        io.emit('contacts', membres);
    });
  });
```

L'affichage de la liste au niveau de l'application se fait dans le fichier HTML, ainsi chaque contact récupéré dans la liste transmise par le serveur correspondra à un bouton qui redirigera vers un salon de discussion privé. La condition ajoutée dans le bouton permet de ne pas afficher celui correspondant à l'utilisateur même, pour éviter tout conflit dans l'application.

Voici le code de la page **contact.html** :

```
<ion-content>
  <ion-list>
    <ion-list-header>Mes Contacts</ion-list-header>
    <div *ngFor="let contact of contacts">
    <button ion-item *ngIf="contact.pseudo !== pseudo"
id="contact.id" (click)="goToProfile(contact.id, contact.pseudo)">
     {{ contact.pseudo }}
    </button></div>
  </ion-list>
</ion-content>
```

Le fichier contact.ts instancie les variables (pseudo et liste de contacts) et appelle le serveur afin de récupérer la liste des contacts lors de son ouverture. On observe le retour du serveur dans le constructeur, afin de valoriser et de mettre à jour la liste des contacts en temps réel. Ceci est permis par les instructions situées dans la méthode ionViewWillLeave(), qui recontacte le serveur pour mettre à jour la liste auprès des autres utilisateurs.

```
export class ContactPage {
  pseudo: any = "";
  contacts: any;
  constructor(public navCtrl: NavController, public alertCtrl: AlertController, public
navParams: NavParams, public socket: Socket) {
    this.pseudo = this.navParams.get('pseudo');
    this.getContacts().subscribe(data => {
        this.contacts = data;
   });
  }
  ionViewWillEnter() {
    this.socket.emit("get-contacts", {});
    this.getContacts().subscribe(data => {
      this.contacts = data;
    });
  }
  ionViewWillLeave() {
    this.socket.emit("get-contacts", {});
  }
  goToRoom(id: any, pseudo: any) {
    this.navCtrl.push(RoomPage, { pseudo: this.pseudo, otherId: id, otherPseudo:
pseudo});
  }
  getContacts() {
    let observable = new Observable(observer => {
      this.socket.on('contacts', (data) => {
        observer.next(data);
      });
    });
    return observable;
  }
}
```

J. CRÉATION DE LA PAGE ET GESTION DE L'ENVOI DES MESSAGES PRIVÉS

La vue du salon privé est fortement semblable à celle du salon général, les messages sont positionnés en fonction de l'émetteur, et une zone de texte et un bouton sont prévus pour envoyer les messages.

Voici le code de la page **room.html** :

```
<ion-content no-bounce padding>
    <ion-grid>
      <!-- Boucle qui affiche l'ensemble des messages recus, last représente la
fin du tableau messages -->
      <ion-row *ngFor="let message of messages; let last = last">

        <!-- Si le message vient d'un autre utilisateur -->
        <ion-col col-3 *ngIf="message.from === pseudoDest"
class="message" [ngClass]="'other_message'">
           <span class="user_name"><b>{{ message.from }} </b> : </span><br>
            <span> {{ message.text }}</span>
           <div class="time"> {{ message.created | date: 'dd/MM HH:mm' }} </div>
        </ion-col>
        <!-- Si le message provient de l'utilisateur -->
        <ion-col offset-4 *ngIf="message.from === pseudo"
class="message" [ngClass]="'my_message'">
           <span class="user_name"><b> {{ message.from }} </b>:</span><br>
            <span> {{ message.text }}</span>
           <div class="time"> {{ message.created | date: 'dd/MM HH:mm' }} </div>
        </ion-col>
        {{ last ? fin() : ''}} <!-- Amène le curseur vers le dernier message  -->
      </ion-row>
    </ion-grid>
 </ion-content>
<ion-footer>
    <ion-item no-lines class="footer">
      <ion-textarea type="text" placeholder="Entrer votre
message" [(ngModel)]="message" (keypress)="IsTyping()" (keyup)="noActivity()" (ke
yup.enter)="sendMessage()"></ion-textarea>
     <button ion-button round outline item-end icon-only
color="primary" (click)="sendMessage()" [disabled]="message === ''">
        <ion-icon class="send" ios="ios-send" md="md-send"></ion-icon>
      </button>
    </ion-item>
  </ion-footer>
```

Dans le même sens, room.ts est fortement semblable au fichier home.ts, il décrit les méthodes permettant d'envoyer le message, d'écouter la réponse du serveur et de placer le curseur sur le dernier message. Le constructeur récupère les différents paramètres fournis par la page précédente et souscrit aux réponses du serveur et ajoute les différents messages reçus dans la liste.

Voici le code de la page **room.ts** :

```
export class RoomPage {

  @ViewChild(Content) content: Content; // Gestion des composants de la vue

  pseudo: any; // Pseudo utilisateur
  idDest: any;
  pseudoDest: any;
  messages = []; // Tableau des messages
  message = ''; // Message en cours de saisie par l'utilisateur
  constructor(
    public navCtrl: NavController,
    public navParams: NavParams,
    private socket: Socket,
 ) {
    this.pseudo = this.navParams.get('pseudo');
    this.idDest = this.navParams.get('destId');
    this.pseudoDest = this.navParams.get('destPseudo')

    this.getMessages().subscribe( message => { // Surveillance des changements
      this.messages.push(message);
    });
}
  sendMessage() {
   this.socket.emit('private-message', { id: this.idDest, text: this.message });
    this.message = '';
  }
  getMessages() { // Reception des messages
    // Création de l'observateur du serveur
    let observable = new Observable(observer => {
      // Observation du serveur : réception des messages du socket
      this.socket.on('p-message', (data) => {
        // On place dans l'objet observer l'objet message retourné par notre
serveur Socket
        observer.next(data);
      });
    });
    return observable; // Retour de l'observateur
  }
  fin() {
    this.content.scrollToBottom(0);
  }
}
```

Enfin le serveur lui s'occupe de récupérer les messages transmis par les clients et les renvoie uniquement aux personnes concernées, identifiées par un identifiant.

Voici le code de la méthode concernée dans le fichier **index.js** :

```
socket.on("private-message", (data) => {
      io.to(data.id).emit("p-message", { text: data.text, from:
socket.nickname, type: 'text', created: new Date() });
      io.to(socket.id).emit("p-message", { text: data.text, from:
socket.nickname,type: 'text', created: new Date() })    });
  });
```

IV. Fonctionnalités additionnelles implémentées

En plus de ce qui était demander au cours du projet, j'ai ajouté de nouvelles fonctionnalités, envoie la liste exhaustive et quelques explications de la mise en place :

- Photo de profil : Chaque utilisateur dispose en plus d'une photo de profil, stockée en base64 dans la base de données; il a la possibilité lors de son inscription, d'importer une photo de sa galerie, à défaut de prendre l'avatar par défaut; chaque message est accompagné de l'avatar de l'émetteur; cette photo est aussi affichée dans la liste des contacts et dans le profil de l'utilisateur.

- Envoi d'image : Comme pour l'inscription, il est possible d'accéder à la galerie à partir du salon général et du salon privé, et ainsi envoyer l'image aux autres utilisateurs; le conteneur du message s'adapte automatiquement au format (texte ou image).

- Onglet profil : Le troisième onglet de l'application correspond au profil de l'utilisateur, il peut y consulter ses différentes informations (résumé, statut de connexion…)

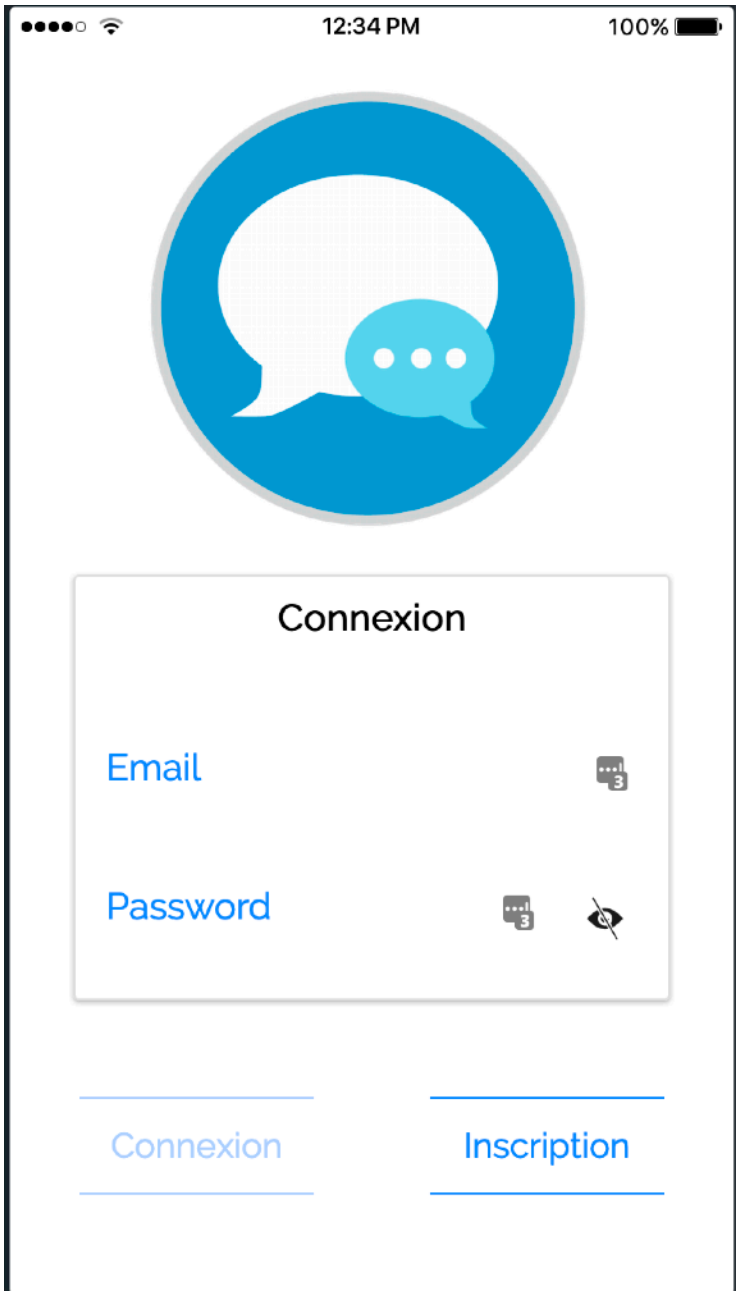
- Profil des utilisateurs : Au lieu d'accéder directement au salon privé avec l'utilisateur sélectionne, l'application redirige vers une page de contact, avec les différentes informations et un bouton permettant d'accéder au salon privé avec ce dernier

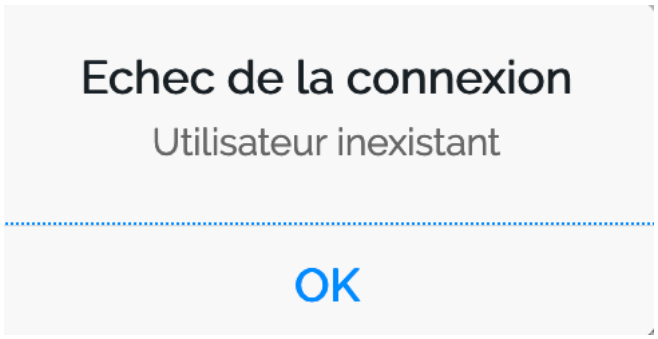
- Nombre d'utilisateurs connectés : En haut à droite du salon général s'affiche le nombre d'utilisateurs connectés au serveur, si l'utilisateur appuie dessus, il sera redirigé vers une liste de ces différents utilisateurs

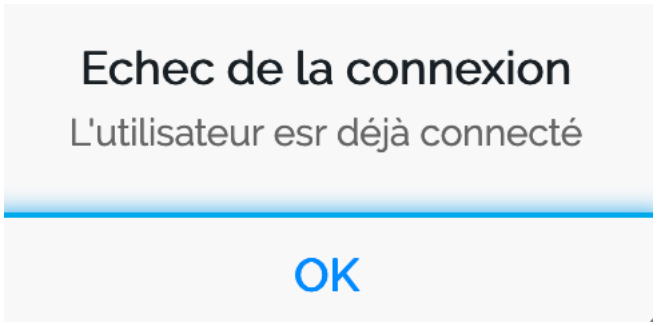
- Déconnexion : Dans un menu accessible après la connexion, il est possible à l'utilisateur de se déconnecter afin de pouvoir changer de compte si nécessaire.

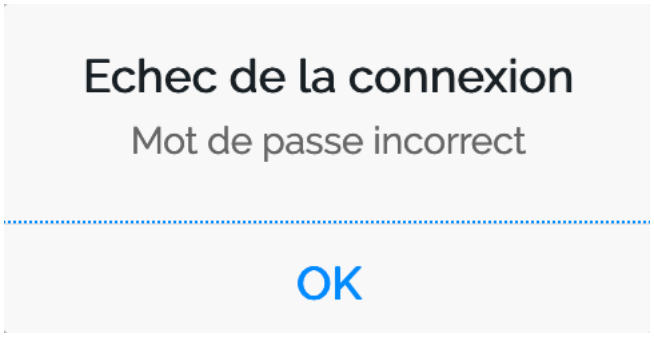
IV. Extrait de la production

A. INTERFACE DE CONNEXION

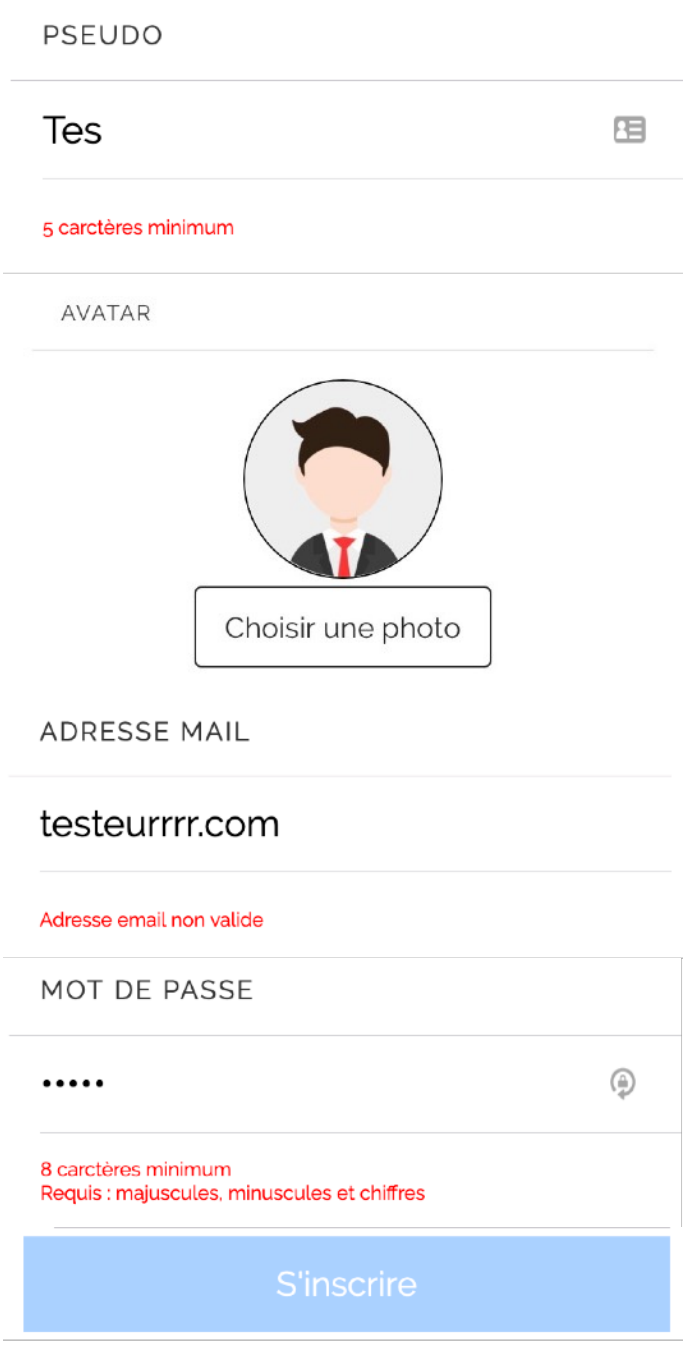




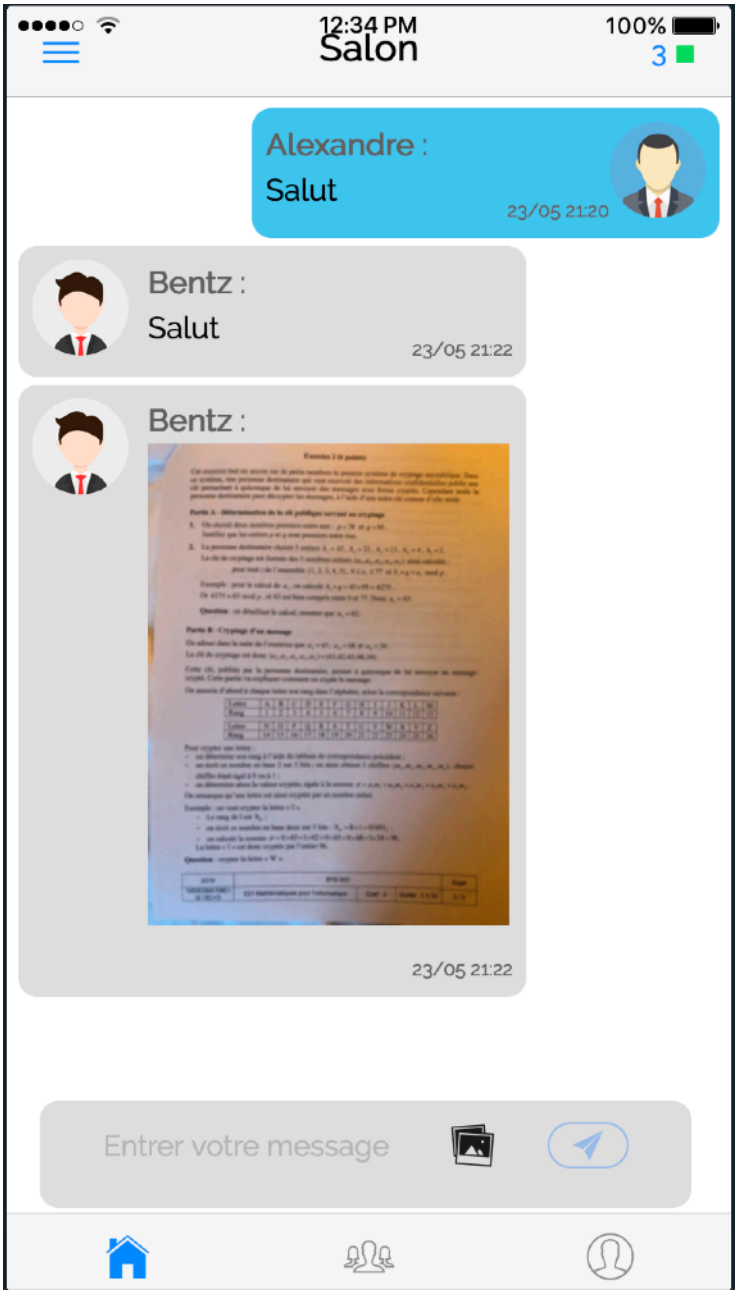




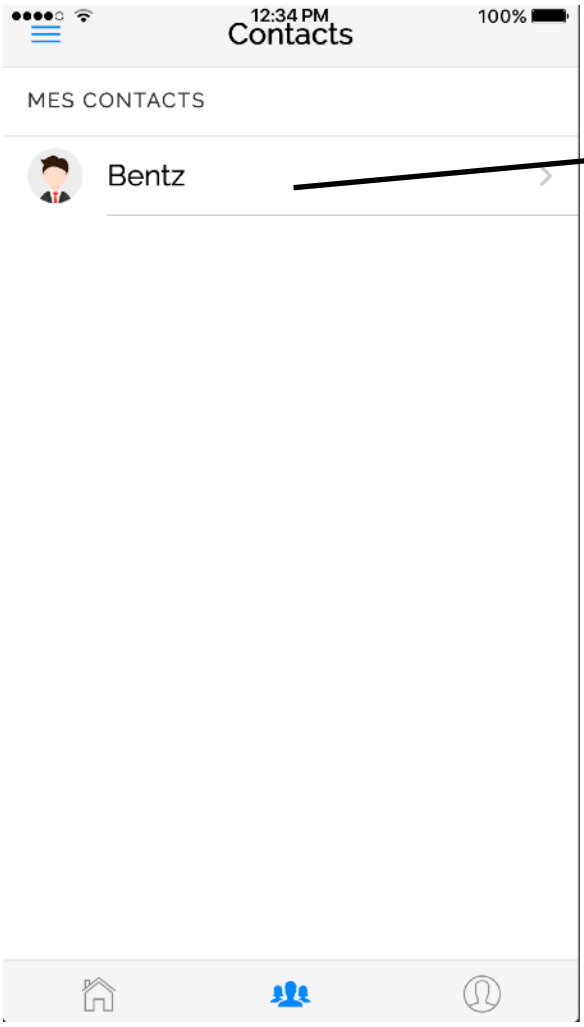
B. INTERFACE D'INSCRIPTION

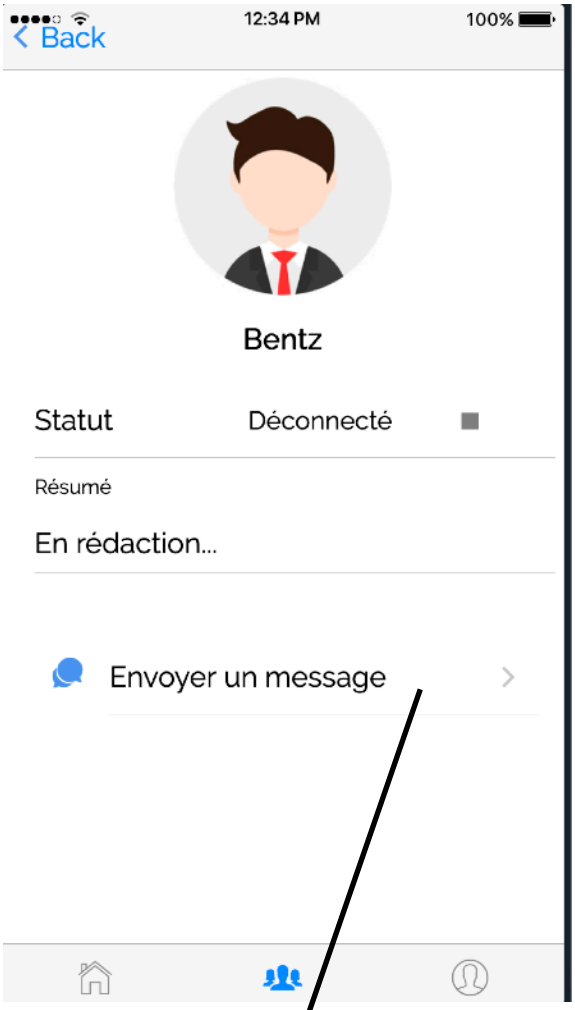


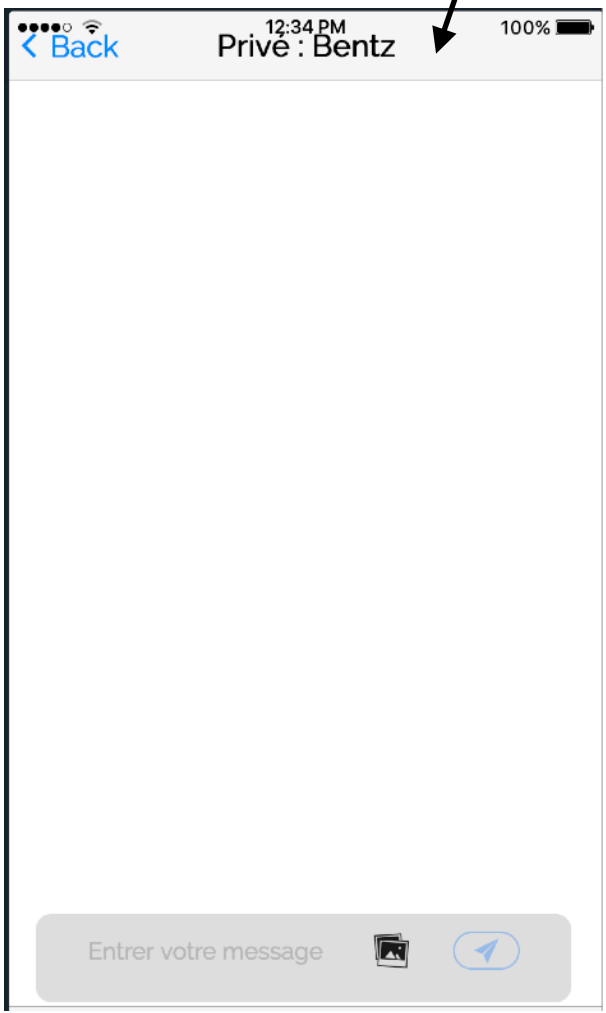
C. SALON GÉNÉRAL



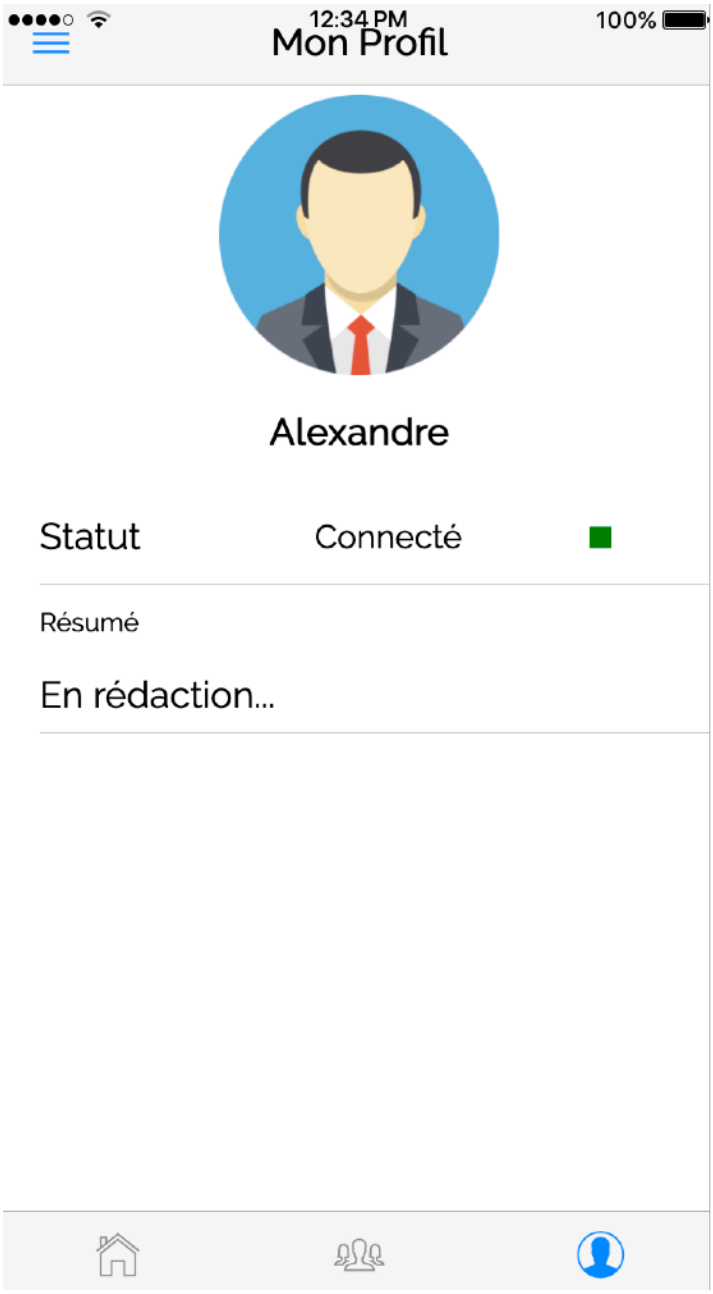
D. INTERFACE DES CONTACTS







E. PROFIL DE L'UTILISATEUR



V. Objectifs d'améliorations

Les pistes d'évolution sont nombreuses, parmi elles, voici les fonctionnalités les plus intéressantes :

- Firebase est un ensemble de service proposé par Google pour tout type d'application. Il met à disposition différents services d'authentification, de bases de données, de mesure de performance, d'hébergement de fichier… Il permettrait donc de proposer différentes façons de s'authentifier pour l'utilisateur (Facebook, Google, mail…). De plus l'hébergement de fichier permettrait de proposer des transferts de fichiers, d'image à travers les discussions.

- La possibilité de créer différents salons avec des droits, des rôles et tout un lot d'information pourrait enrichir les fonctionnalités du cette application en permettant de regrouper les utilisateurs par centre d'intérêt, par service, par secteur géographique…

- L'intégration des messages vocaux, des appels et des Visio-chat pourrait ajouter une nouvelle dimension dans l'interaction entre les utilisateurs et ainsi lui apporterait plus de valeur.