

04/06/2018

# Compte rendu

IHM

« J'atteste que ce travail est original, qu'il indique de façon appropriée tous les emprunts, et qu'il fait référence de façon appropriée à chaque source utilisée »

Léo Guilpain - IoT

## Table des matières

Fonctionnalités.....	2
Serveur et socket.....	2
Composant Login.....	2
Composant Client .....	5
Composant Group .....	11
Composant GroupMessage .....	16
Affichage des différents types de messages .....	18
Organisation de l'application .....	19
Ce qui n'a pas été fait.....	19
Wit.ai .....	19
Transfert de fichier .....	19
Utilisation de source.....	20

## Fonctionnalités

### Serveur et socket

Le serveur écoute sur port 8080.

```
// On lance le serveur en écoutant les connexions arrivant sur le port 3000
http.listen(8080, function(){
  console.log('Server is listening on *:8080');
});
```

La socket du client se connecte au serveur.

```
export var socket = require('socket.io-client')('http://192.168.0.11:8080');
```

### Composant Login

Les fonctionnalités présentes dans cette application de chat sont les mêmes que celles développer lors du premier semestre.

L'application possède un serveur et une partie cliente. Contrairement aux commandes à taper du type "/l", tout se fait avec des boutons ou des champs textes à remplir.

Lors de la connexion de la socket client avec le serveur, un client est créé avec de nombreux paramètre.

```
var client = {socketclient : socket, pseudo : pseudo, mdp : mdp , repertoire : repertoire, listmess : list};
tabclient.push(client);
```

Figure 1 : Création d'un client à la connexion

Comme vous pouvez le voir, le client possède une socket, un pseudo, un mot de passe, un répertoire et une liste de message.

Pour commencer, le client arrive sur la première page :

USERNAME :

PASSWORD :

Sur cette page, il doit rentrer son login et son mot de passe. Ces deux valeurs sont ensuite comparées à une base de données qui contient les informations des utilisateurs.

Voici le code lorsque l'utilisateur clique sur le bouton login ou create :

```
clickedLogin(){
    var ajouter = JSON.stringify({nickname : this.state.username, mdp:this.state.password});
    socket.emit('connexion',ajouter)
}

clickedCreate(){
    var ajouter = JSON.stringify({nickname : this.state.username, mdp:this.state.password})
    socket.emit('addPseudo',ajouter)
}
```

Figure 2 : Code représentant l'action des boutons

On voit bien ici que la socket envoie l'événement 'connexion' ou 'addPseudo'. Cet événement est ensuite récupéré dans la partie serveur.

Dans l'exemple ci-dessous, cela se passe en cas de réception de l'événement 'connexion'.

```
//client déjà existant
socket.on('connexion',function(message){
    var donnees = JSON.parse(message);
    var nom = donnees.nickname;
    var mdpenclair = donnees.mdp;
    ex = 0;
    var ex1 = 1;
    db.serialize(function(){
        tabpseudo.forEach(function(element){
            if(nom == element){
                ex = 1;
            }
        })
    })
    if(ex != 1){
        var mess = 'Erreur ! Le pseudo est incorrect';
        var mes = JSON.stringify({mes : mess})
        socket.emit('erreur',mes);
    }

    tabclient.forEach(function (client){
        if(client.pseudo == nom){
            ex1 = 0;
            var mess = 'Erreur ! Vous êtes déjà en ligne';
            var mes = JSON.stringify({mes : mess})
            socket.emit('erreur', mes);
        };
    });

    if(ex ==1 & ex1 == 1){
        var sql = 'SELECT mdp FROM client WHERE name = ?';
        db.get(sql,[nom], function (err, row){
            if(donnees.mdp == row.mdp) {
                client.pseudo = nom;
                client.mdp = mdpenclair;
                chemin = process.cwd() + '/' + client.pseudo;
                client.repertoire = chemin;

                var message = 'Salut ' + donnees.nickname + ', tu es connecté ! ';
                idclient = idclient + 1;
                client.id = idclient;
                var indic = "vient de se connecter"
                var mes = JSON.stringify({mes : indic, user : nom})
                tabclient.forEach(function (client){
                    if(client.pseudo != nom){
                        client.socketclient.emit('message', mes);
                    };
                });
                var mes = JSON.stringify({chat : 'client', mes : message})
                socket.emit('loginok',mes);
            }
        })
    }
})
```

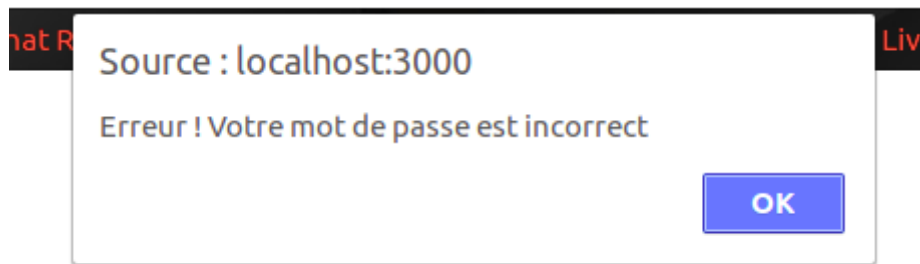
Figure 3 : Code du serveur lors de la réception de l'évènement 'connexion'

Après avoir reçu l'évènement, le serveur traite les données et les compare avec la base de données. On peut voir qu'en fonction des données rentrées, différents cas sont possibles et donc différents messages sont renvoyés à l'utilisateur.

Pour commencer, l'utilisateur peut recevoir 3 types d'erreurs :

- Le mot de passe est incorrect
- Le pseudo est incorrect
- L'utilisateur est déjà en ligne

Il reçoit ces messages sous forme d'alerte comme ci-dessous :



L'utilisateur doit donc retenter sa chance avec un autre mot de passe.

En revanche si toutes les coordonnées sont correctes, alors le serveur envoie l'évènement "loginok". Dans cet événement, la variable chat est mis à jour et prend comme valeur 'client' et un message est envoyé aux autres utilisateurs les prévenant que ce dernier est connecté.

Lors de la réception de cet événement le client change l'état de sa variable chat et cela permet de passer au composant suivant :

```
socket.on('loginok',data=>{
  var donnees = JSON.parse(data);
  var mess = donnees.chat;
  var logok = donnees.mes;
  this.setState({infoconn : logok})
  this.setState({chat : mess})
})
```

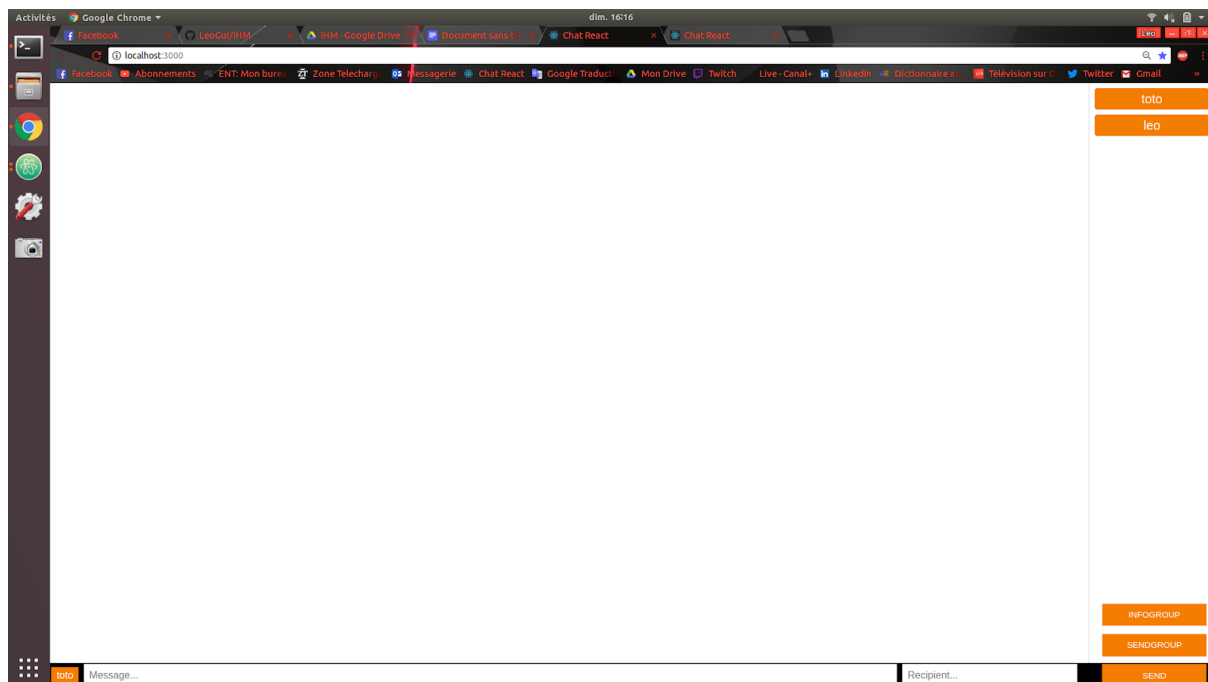
Figure 4 : Mise à jour de la variable chat dans le client

```
if(this.state.chat === 'client'){
  return(
    <div>
      <Client infoconn = {this.state.infoconn} username = {this.state.username} password = {this.state.password} />
    </div>
  )
}
```

Figure 5 : Passage au composant suivant

Ensuite nous arrivons sur le deuxième composant de notre application.

## Composant Client



Sur ce composant, on peut différencier 4 parties :

- Le champ de réception des message
- La liste des clients connectés
- La barre pour envoyer un message
- Les boutons effectuant des actions

On peut voir, d'après la liste se trouvant sur la droite, que les utilisateurs "toto" et "leo" sont connectés.

L'utilisateur qui souhaite envoyer un message doit remplir la case "message" avec le message qu'il souhaite envoyer et la case "Recipient" avec le nom de l'utilisateur à qui il veut l'envoyer.

```
clickedSend(){  
  // message privé  
  if (this.state.destinataire != ''){  
    var mes = JSON.stringify({source : this.state.ownuser, destination : this.state.destinataire, message :this.state.writemessage});  
    socket.emit('private', mes)  
  };  
  
  //message broadcast  
  if (this.state.destinataire == ''){  
    var mes = JSON.stringify({source : 'Broadcast : ' + this.state.ownuser, message :this.state.writemessage});  
    socket.emit('broadcast', mes)  
  };  
}
```

Figure 6 : Code montrant l'action faite lors de l'appui sur le bouton send

Si la case "Recipient" n'est pas remplie lors du clique sur le bouton 'send' le message est envoyé à tous les utilisateurs (message broadcast)

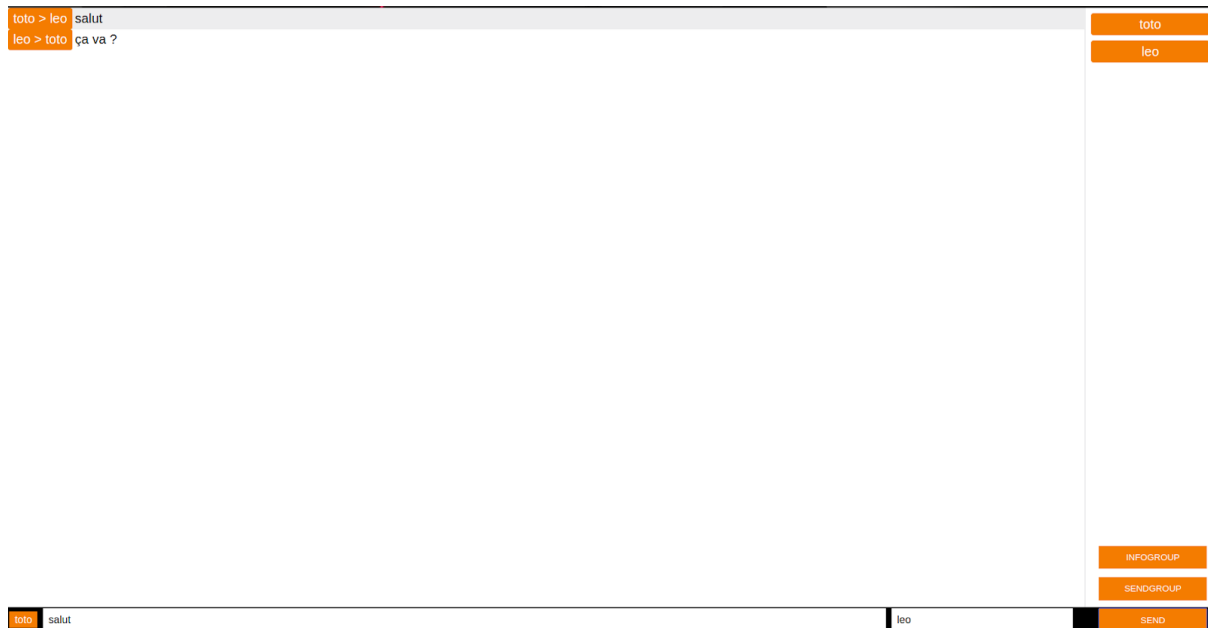
```
// broadcast
socket.on('broadcast',function(message){
  var donnees = JSON.parse(message);
  var envoyeur = donnees.source;
  var message = donnees.message;
  var mes = JSON.stringify({mes : message, user : envoyeur})
  tabclient.forEach(function (client){
    if(client.pseudo != envoyeur){
      client.socketclient.emit('message', mes);
    };
  });
  socket.emit('message', mes);
});

// message privé
socket.on('private',function(message){
  var donnees = JSON.parse(message);
  var ex;
  var envoyeur = donnees.source;
  var receivemess = donnees.message
  var utilisateur = envoyeur + ' > ' + donnees.destination;
  var mes = JSON.stringify({mes: receivemess, user : utilisateur})
  tabclient.forEach(function (client){
    if (client.pseudo == donnees.destination){
      ex = 1;
      client.socketclient.emit('message', mes);
      socket.emit('message', mes);}
  });
  if (ex != 1){
    var indic = " n'est pas connecté !";
    var mes = JSON.stringify({mes: indic, user : donnees.destination})
    socket.emit('message',mes);
  };
});
```

Figure 7 : Code du serveur pour la réception d'un message

Comme on peut le voir, lorsque le serveur reçoit l'événement 'broadcast' ou 'private', il envoie l'évènement 'message'.

Lors de la réception du message, ce dernier apparaît comme ceci :



Pour l'afficher comme ceci, il a fallu créer une liste :

```
class Client extends Component {
  constructor(props){
    super(props);
    this.state = ({
      destinataire:'',
      username : this.props.username,
      writemessage: '',
      group:'',
      password : this.props.password,
      connection : this.props.infoconn,
      ownuser : this.props.username,
      listmessage : [],
      listuser : [],
      chat : 'client',
    });
  }
}
```

Figure 8 : état initial du client

```
socket.on('message',data=>{
  var donnees = JSON.parse(data);

  var mess = donnees.mes;
  this.setState({writemessage : mess});
  var user = donnees.user;
  this.setState({username : user})
  var utilisateur = this.state.username;
  var message = this.state.writemessage;
  var objet = {utilisateur,message};

  var listmessage = this.state.listmessage;

  listmessage.push(objet)
  this.setState({listmessage : listmessage})

  var test = JSON.stringify({source : this.state.ownuser,list : listmessage})
  socket.emit('listmessage',test)
})
```

Figure 9 : Action suite à la réception de l'évènement 'message'



Lors de la réception de l'évènement 'message', le message et le nom d'utilisateur sont remplacés dans le 'state'.

Ensuite, il faut créer un objet contenant le nom de l'utilisateur et le message. Après avoir créé une liste vide que l'on remplace par la liste présente dans le 'state', il suffit d'ajouter l'objet contenant le message et la liste utilisateur.

Enfin, il faut mettre à jour la liste présente dans le 'state' en la remplaçant par la liste à laquelle on a ajouté l'objet.

L'évènement 'listmessage' est ensuite envoyé. Le serveur reçoit ceci :

```
socket.on('listmessage',function(message){
  var donnees = JSON.parse(message);

  tabclient.forEach(function (client){
    if (client.pseudo == donnees.source){
      client.listmess = donnees.list;
      var listmes = JSON.stringify({mes : client.listmess})
      socket.emit('history', listmes)
    }
  });
})

socket.on('update',function(message){
  var donnees = JSON.parse(message);
  tabclient.forEach(function (client){
    if (client.pseudo == donnees.source){
      var listmes = JSON.stringify({mes : client.listmess})
      socket.emit('history', listmes)
    }
  });
})
})
```

Figure 10 : Code du serveur suite à la réception de l'évènement 'listmessage'

```
socket.emit('update',pseudo);

socket.on('history',data=>{
  var donnees = JSON.parse(data);
  var listmess = donnees.mes;
  this.setState({listmessage : listmess})
})
```

Figure 11 : Réception coté client

Ces fonctions ont pour but de stocker la liste de message sur le serveur. Cela permet ainsi à l'utilisateur d'avoir accès à ses messages en continu.

Pour l'affichage, il suffit de faire :

```
const messagepc = this.state.listmessage.map((x,i) => <li className = "text" key ={i}> <span class = "Username"> {x.utilisateur} </span>{x.message}</li>)
```

Ici, on affiche le nom d'utilisateur de façon surligné et le message ensuite.

Ensuite il suffit d'afficher messagepc.

toto > leo yo

En ce qui concerne la liste d'utilisateur connecté, lors de sa connexion, l'utilisateur émet l'événement 'updateList'.

```
socket.emit('updateListe',pseudo);
socket.on('updateliste',data=>{
  var donnees = JSON.parse(data);
  var listclient = donnees.mes;
  this.setState({listuser : listclient})
})
```

Figure 12 : Code pour mettre à jour la liste

Le serveur exécute ceci ensuite :

```
// liste des clients
socket.on('updateListe',function(message){
  Update(message);
});

function Update(message){
  var listclient = []
  var donnees = JSON.parse(message);
  var envoyeur = donnees.source;
  tabclient.forEach(function (client){
    listclient.push(client.pseudo);
  });
  var message = listclient;
  var mes = JSON.stringify({mes: message})
  tabclient.forEach(function (client){
    if(client.pseudo != envoyeur){
      client.socketclient.emit('updateliste', mes);
    }
  });
  socket.emit('updateliste',mes);
}
```

Figure 13 : Code du serveur suite à la réception de l'évènement 'updateList'

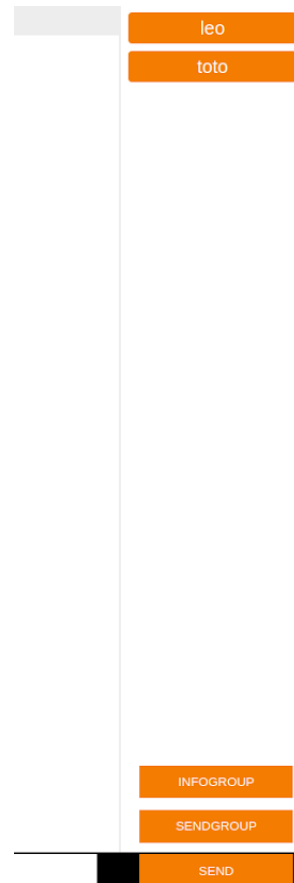
Lorsque l'événement 'updateListe' est envoyé, une liste de client est créée dans laquelle on ajoute tous les pseudos des clients, ensuite on envoie cette liste à travers les socket via l'événement 'updateliste'.

Lorsque le client le reçoit, il met à jour sa liste d'utilisateur en la remplaçant par cette liste.

Pour l'afficher, il suffit de faire ceci :

```
const listclient = this.state.listuser.map((x,i) => <li className = "listclient" key ={i}> {x}</li>)
```

Ensuite, on affiche 'listclient'.



Pour avoir accès au groupe, il suffit de cliquer sur le bouton 'infogroup'.  
On arrive sur le composant suivant.

## Composant Group

JOIN

CREATE

LEAVE



BACK

Plusieurs solutions s'offrent au client :

- Il rejoint un groupe
- Il le crée
- Il le quitte
- Il obtient des informations sur ce groupe
- Il peut également revenir au composant précédent en appuyant sur le bouton back

Dans un premier temps, l'utilisateur doit rentrer le nom du groupe avec lequel il souhaite interagir.

Lorsque l'utilisateur clique sur 'create' :

```
clickedCreate(){  
  var message = JSON.stringify({nomGroupe : this.state.nameGroup});  
  socket.emit('createGroup',message)  
  socket.emit('listeGroupe');  
}
```

Figure 14 : Action suite à l'appui sur le bouton create

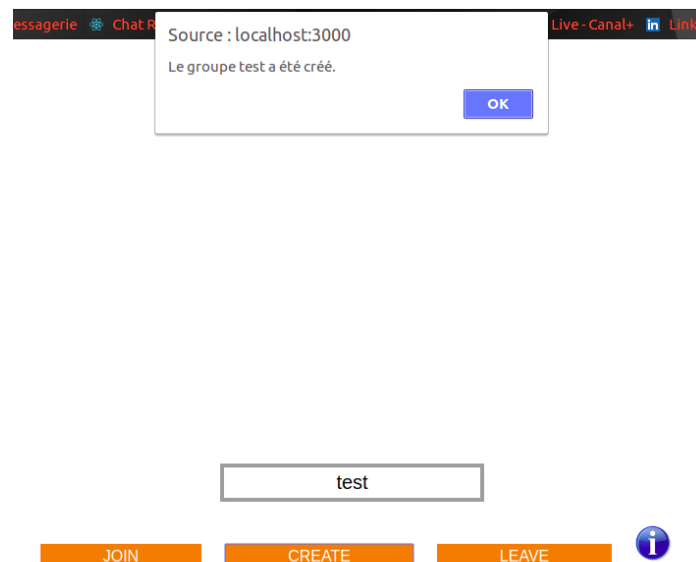
L'événement 'createGroup' est envoyé ainsi que l'événement 'listeGroupe'.

```
// créer un groupe
socket.on('createGroup',function(message){
  var donnees = JSON.parse(message)
  var nomGroupe = donnees.nomGroupe
  var ex = 0 ;
  tabgroup.forEach(function(groupe){
    if (nomGroupe == groupe.name){
      var message = 'Le groupe ' + nomGroupe + ' existe déjà.'
      var mes = JSON.stringify({mes: message})
      socket.emit('erreur',mes);
      ex = 1 ;}
  });
  if (ex!=1){
    num = num + 1 ;
    var tableauclient = [];
    var groupe = {numgroupe : num ,name : nomGroupe, tableauclient : tableauclient};
    tabgroup.push(groupe);
    var message = 'Le groupe ' + nomGroupe + ' a été créé.'
    var mes = JSON.stringify({mes: message})
    socket.emit('createok', mes);}
});
```

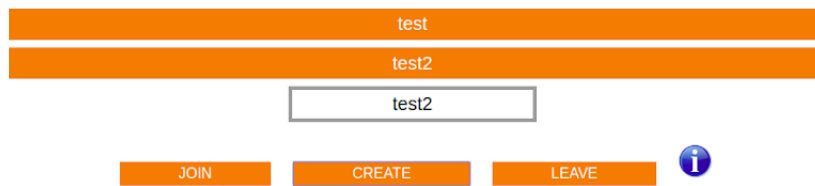
Figure 15 : Code serveur lors de la réception de l'évènement create

Comme on peut le voir, le serveur vérifie si le groupe n'a pas déjà été créé auparavant, si ce n'est pas le cas, il envoie l'évènement 'createok'.

Lors de sa réception par le client, une alerte est créée comme ceci :



Comme on a pu le voir, lors de l'appui sur le bouton 'create', le client envoie l'évènement 'listeGroupe', cela permet de mettre à jour la liste des groupes et de l'afficher par la suite. Pour l'affichage, cela fonctionne exactement comme pour l'affichage de la liste des clients. On obtient ainsi ce résultat.



Ici, les groupes 'test' et 'test2' ont été créé. Désormais l'utilisateur peut rejoindre le ou les groupes qu'il souhaite.

Nous allons rejoindre le groupe 'test'.

```
clickedJoin(){  
  var message = JSON.stringify({source : this.state.username, nomGroupe : this.state.nameGroup});  
  socket.emit('join',message)  
}
```

Figure 16 : Action suite à l'appui sur le bouton join

Le client envoie à travers la socket l'événement 'join' et le serveur le reçoit.

```
// rejoindre un groupe
socket.on('join',function(message){
  var donnees = JSON.parse(message)
  var ex = 0;
  var envoyeur = donnees.source;
  var groupeinit = donnees.nomGroupe;
  var message = 'Vous avez rejoins le groupe : ' + donnees.nomGroupe;
  var mes = JSON.stringify({mes: message, chat:'client'})
  var indic1 = ' a rejoint le groupe ' + donnees.nomGroupe;
  var indic = JSON.stringify({user : envoyeur, mes: indic1})
  tabgroup.forEach(function (groupe){
    if(groupeinit == groupe.name){
      ex = 1;
      groupe.tableauclient.push(client);
      socket.emit('joinok',mes);
    }
    groupe.tableauclient.forEach(function(client){
      if(groupeinit == groupe.name){
        if(envoyeur != client.pseudo ) {
          client.socketclient.emit('message',indic);
        }
      };
    });
  });
  if (ex!=1){
    var message = 'Aucun groupe du nom ' + donnees.nomGroupe + ' existe.';
    var mes = JSON.stringify({mes: message})
    socket.emit('erreur',mes);
  }
});
```

Figure 17 : Code du serveur lorsqu'il reçoit l'événement 'join'

Le serveur envoie une alerte au client pour le prévenir de sa bonne connexion au groupe ou non. Il émet également un message à tous les utilisateurs qui sont déjà présents pour les informer de la connexions d'un autre membre. Voici ce que l'on obtient en cliquant sur le bouton 'join'.

Vous avez rejoins le groupe : test

OK

En ce qui concerne le bouton 'leave', idem que pour les autres boutons les échanges entre le client et le serveur sont identiques. Lorsqu'il quitte, l'utilisateur est supprimé du groupe et ne peut plus envoyer ni recevoir de message via ce groupe.

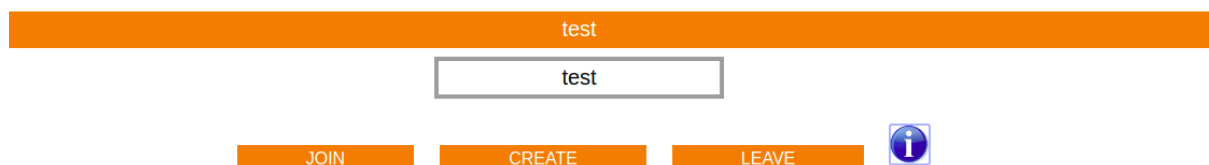
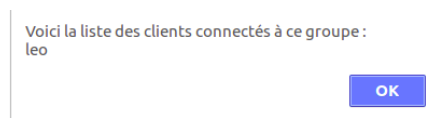
Comme vous pouvez le voir, il y a un bouton information. Ce bouton permet à l'utilisateur de savoir qui est présent dans ce groupe.

Lorsque le client clique sur le bouton, l'événement 'listClientGroupe'.

```
// liste des clients dans le groupe
socket.on('listClientGroupe',function(message){
  var donnees = JSON.parse(message)
  var listeClient = [];
  tabgroup.forEach(function(groupe){
    if(donnees.nomGroupe == groupe.name){
      groupe.tableauclient.forEach(function(client){
        listeClient.push(client.pseudo)});
    };
  });
  var message = 'Voici la liste des clients connectés à ce groupe : \n'+ listeClient;
  var mes = JSON.stringify({mes : message});
  socket.emit('erreur',mes);
});
```

Figure 18 : Code de serveur lors de la réception de l'événement 'listClientGroupe'

Le serveur envoie donc la liste des clients qui sont connectés au groupe. Ces informations sont affichées sous forme d'alerte.



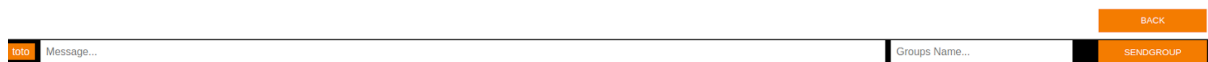
On peut voir ici que le seul client connecté au groupe test est léo.

Après avoir cliqué sur les boutons qu'ils souhaitent, l'utilisateur retourne sur la page principale. Il va ensuite cliquer sur le bouton 'sendGroup'.



## Composant GroupMessage

test



L'utilisateur peut voir en haut de la page les groupes créés précédemment. Pour envoyer un message, il lui suffit de remplir la case 'message' mais également la case avec le nom du groupe.

```
clickedSendGroup(){  
  var message = JSON.stringify({nomGroupe : this.state.group,message : this.state.writemessage,source : this.state.ownuser});  
  socket.emit('broadcastGroup',message);  
  this.setState({chat : 'client'})  
}
```

Figure 19 : Action suite à l'appui sur le bouton Sendgroup

Le client émet l'événement 'broadcastGroup'.

```
// broadcastGroup
socket.on('broadcastGroup',function(message){
    var donnees = JSON.parse(message);

    //Fonction permettant de savoir si le client appartient au groupe ou non
    var existe = function(){
        var etat = false;
        var envoyeur = donnees.source;
        tabgroup.forEach(function(groupe){
            if(groupe.name === donnees.nomGroupe){
                groupe.tableauclient.forEach(function(client){
                    if (envoyeur === client.pseudo){
                        etat = true;
                    }
                })
            }
        });
        return etat;
    }

    var ex;
    envoyeur = client.pseudo;
    var user = envoyeur + " | " + donnees.nomGroupe;
    var mes = JSON.stringify({mes: donnees.message, user : user})
    tabgroup.forEach(function(groupe){
        if(donnees.nomGroupe == groupe.name){
            if (existe() == true){
                groupe.tableauclient.forEach(function(client){
                    if(client.pseudo != envoyeur){
                        ex = 1;
                        client.socketclient.emit('message', mes));
                    }
                });
                socket.emit('message',mes);
            };
        });
        if(ex != 1){
            var message = 'Vous ne faites pas partis de ce groupe';
            var mes = JSON.stringify({mes: message})
            socket.emit('erreur',mes));
        }
    });
});
```

Figure 20 : Code serveur suite à la réception de l'évènement broadcastgroup

Le serveur analyse dans un premier temps si l'utilisateur fait partie du groupe auquel il souhaite envoyer un message. Si c'est le cas, le message est envoyé à tous les utilisateurs composant ce groupe.

L'utilisateur 'leo' étant présent dans le même groupe 'test' que 'toto', lorsque ce dernier envoie un message, il le reçoit à la même place que lorsqu'il reçoit un message privé.

toto | test salut

## Affichage des différents types de messages

```
toto | test salut  
Broadcast : toto comment ça va ?  
toto > leo yo
```

Ici, l'utilisateur leo a reçu un message de la part de 'toto'

- via le groupe 'test'
- via un message broadcast
- via un message privé

Enfin la dernière fonctionnalité de cette application est le fait que lorsqu'un utilisateur se déconnecte, il est supprimé de la liste de clients connectés mais également de tous les groupes auxquels il faisait partie.

```
function Updatedis(message){  
  var listclient = []  
  var envoyeur = message;  
  tabclient.forEach(function (client){  
    listclient.push(client.pseudo);  
  });  
  var message = listclient;  
  var mes = JSON.stringify({mes: message})  
  tabclient.forEach(function (client){  
    if(client.pseudo != envoyeur){  
  
      client.socketclient.emit('updateliste', mes);  
    });  
    socket.emit('updateliste',mes);  
  }  
}  
  
// quitter proprement  
socket.on('disconnect',function(){  
  var i = 0;  
  var j = 0;  
  var envoyeur = client.pseudo;  
  tabgroup.forEach(function(groupe){  
    groupe.tableauclient.forEach(function(client){  
      if(envoyeur == client.pseudo){  
        groupe.tableauclient.splice(i,1);  
        i = i + 1;  
      });  
    });  
  });  
  tabclient.forEach(function(client){  
    if(envoyeur == client.pseudo){  
      tabclient.splice(j,1);  
    }  
    if(envoyeur != client.pseudo){  
      var indic = "s'est déconnecté !";  
      var mes = JSON.stringify({user : envoyeur, mes : indic})  
      client.socketclient.emit('message',mes);  
      j = j + 1;  
    }  
  });  
  
  Updatedis(envoyeur)  
  idclient = idclient - 1;  
  
});
```

Figure 21 : Code serveur suite à la réception de l'évènement disconnect

## Organisation de l'application

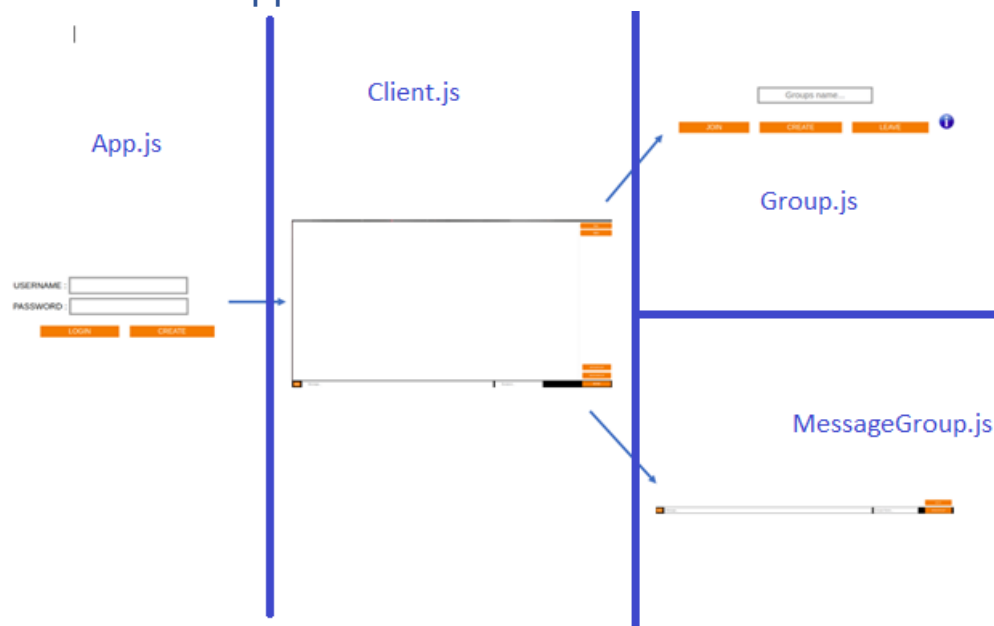


Figure 22 : Organisation de l'application en composants

Dans cette application, il y a 4 composants :

- Login
- Client
- Group
- Sendgroup

## Ce qui n'a pas été fait

### Wit.ai

Malheureusement je n'ai pas eu le temps d'entraîner et d'intégrer le robot dans mon application.

### Transfert de fichier

Au premier semestre, nous avons intégré le transfert de fichier. Malheureusement je n'ai pas réussi à l'intégrer ici.

## Utilisation de source

En ce qui concerne les source utilisés, je n'ai utilisé que ce site :  
<https://blog.bini.io/developper-une-application-avec-socket-io/>

Ce dernier m'a permis de réaliser le CSS de mon application.