

TD5 - WOB Compte-rendu

Avant-propos :

Le but de ce TD est d'implémenter les fonctionnalités qui ont été faites en TD3 et en TD4 mais cette fois-ci avec socket.io.

Nous allons donc voir les détails de ce qui a été implémenté, ainsi que les difficultés rencontrées.

De plus, au TD précédent, il m'a manqué du temps pour implémenter la base de données. J'explique après la partie sur le code des fonctionnalités du chat comment j'ai procédé pour élaborer cette base de données.

Clarté du code :

En vue d'améliorer la lisibilité du code par rapport à ce qui a été fait en TD3 et TD4, j'ai essayé de l'aérer un maximum et de commenter en détails les concepts pour les expliquer. Dans le switch du serveur, voilà le procédé qui a été adopté :

```
break;
case 'broadcast':
    socket.broadcast.emit('message', { sender: 'server' });
    break;
case 'list':
    socket.emit('message', { sender: 'server', data: groups });
    break;
case 'quit':
    socket.disconnect();
    break;
case 'cgroup':
    creerGroupe();
    break;
case 'join':
    joinGroupe();
    break;
```

Chaque « case » du switch ne comporte que deux lignes pour aérer au maximum. A chaque « case », on fait appel à la fonction appropriée. Les fonctions sont toutes définies en dessous du code. Ainsi, la personne qui veut y accéder pourra distinguer les différentes parties et observer seulement ce qui l'intéresse.

```
case 'user.quit':
    console.log(`${data.user} a quitté chat.`);
    break;
case 'user.broadcast':
    console.log(`${data.sender}>${data.msg}`);
    break;
case 'group.create':
    console.log(`${data.sender} a créé le groupe`);
    break;
case 'group.create.already.exists':
    console.log(`Le groupe existe déjà.`);
    break;
```

Côté client, il a été décidé de ne pas utiliser de fonction cette fois-ci, puisque le contenu des « case » est léger. Cela évite de limiter un peu plus la longueur du code et par conséquent de gagner en lisibilité.

```
socket.emit('message', { sender: 'server',
```

La principale différence que l'on va observer ici sera l'utilisation du « socket.emit » plutôt que « socket.write » que l'on utilisait dans les TDs précédents.

Guide de l'utilisateur :

Voici les commandes qu'un utilisateur peut utiliser lorsqu'il est connecté :

Intitulé	Effet de la commande	Commande
Broadcast	Envoyer un message à tous les utilisateurs	b;
List	Afficher la liste de tous les utilisateurs	ls;
Quit	Se déconnecter	q;
Create Group	Créer un groupe	cg;
Join	Joindre un groupe	j;
Broadcast Group	Diffuser un message à tous les utilisateurs d'un groupe	bg;
List members	Afficher les membres d'un groupe	members;
List messages	Afficher l'historique des messages d'un groupe	messages;
List groups	Afficher la liste des groupes	groups;
Leave	Quitter un groupe	leave;
Invite	Inviter un utilisateur dans un groupe	invite;
Kick	Exclure un utilisateur d'un groupe	kick;
Ban	Bannir un utilisateur d'un groupe	ban;
Unban	Débannir un utilisateur d'un groupe	unban;
States	Afficher les opérations effectuées dans un groupe	states;

Ci-dessous, j'expliquerai en détails comment les fonctionnalités voulues ont été codées.

Voici trois fonctions primordiales qui ont été créées. Il s'agit de fonction qui vont vérifier si un utilisateur est présent dans un groupe, si l'utilisateur expéditeur est banni d'un groupe et si l'utilisateur destinataire est banni d'un groupe. Elles parcourent les tableaux des utilisateurs et des utilisateurs bannis et renvoient un Boolean selon si l'utilisateur est présent ou pas. Elles seront très utiles pour coder les fonctionnalités du chat.

```
function testpres () //test de l'appartenance à un groupe
{
  let test = false; //boolean qui change de valeur si présent
  for(i=0;i<groups[0].length;i++)
  {
    if(groups[0][i]==data.sender && groups[2][i]==data.group)
    {
      test=true; //true si déjà présent dans le groupe
    }
  }
  return test;
}

function testbanjoin() //test si le sender est banni
{
  let test = false; //boolean qui change de valeur
  for(i=0;i<bannis[0].length;i++)
  {
    if(bannis[0][i]==data.sender && bannis[2][i]==data.group)
    {
      test=true; //true si déjà présent dans le groupe
    }
  }
  return test;
}

function testbandest() //test si le destinataire est banni
{
  let test = false;
  for(i=0;i<bannis[0].length;i++)
  {
    if(bannis[0][i]==data.dest && bannis[2][i]==data.group)
    {
      test=true;
    }
  }
}
```

Create Group

```
function creerGroupe()
{
  if(!(groups[2].includes(data.group))) //on vérifie que le groupe n'existe pas
  {
    groups[0].push(data.sender); //on push le sender dans le tableau
    groups[1].push(socket); //on push la socket du sender dans le tableau
    groups[2].push(data.group); //on push son groupe dans le tableau

    socket.emit('message', {sender: data.sender, group: data.group, event: 'group.create'});
    socket.broadcast.emit('message', {sender: data.sender, group: data.group, event: 'group.create'});
  }

  else //le groupe existe déjà
  { socket.emit('message', {sender: data.sender, group: data.group, event: 'group.create.already.exists'});}
}
```

On distingue ici deux cas : soit le groupe existe déjà et on renvoie à l'utilisateur qu'il ne peut pas créer un groupe qui existe déjà, soit le groupe n'existe pas et il peut le créer. Pour cela, on parcourt le tableau qui contient les groupes, et on ajoute le nom du nouveau groupe s'il n'existe pas déjà.

Join

```
function joinGroupe(){
  //Gestion des événements
  operations[0].push(data.sender + " a joint le groupe " + data.group);
  operations[1].push(data.group);

  //si le groupe existe et l'utilisateur y appartient
  if(testpres()==true)
  { socket.emit('message', { sender: data.sender, group: data.group, event: 'group.already.joined'}); }

  //si l'utilisateur n'appartient pas au groupe, que le groupe existe et qu'il n'est pas banni
  else if(testpres()==false && groups[2].includes(data.group) && testbanjoin()==false)
  {
    groups[0].push(data.sender); //on push le sender dans le tableau
    groups[1].push(socket); //on push la socket
    groups[2].push(data.group); //on push le groupe
    socket.emit('message', { sender: data.sender, group: data.group, event: 'group.join'});
  }

  //si le groupe n'existe pas
  else if(!groups[2].includes(data.group)){ //le groupe n'existe pas -> on refuse
    socket.emit('message', { sender: data.sender, group: data.group, event: 'group.does.not.exist'});
  }
}
```

La fonctionnalité join doit gérer plusieurs cas de figure. Avec la fonction testpres() décrite précédemment, on teste si l'utilisateur appartient déjà au groupe qu'il souhaite rejoindre, et on lui renvoie le message adéquat. Avec cette même fonction couplée à testbanjoin() pour voir si l'utilisateur est banni du groupe qu'il souhaite rejoindre ainsi qu'avec un test pour savoir si le groupe existe on distingue un autre cas. On ajoute alors l'utilisateur dans le tableau associé à son nom de

groupe. Le dernier cas que l'on doit gérer est de voir si le groupe n'existe pas. On renvoie là aussi le message approprié.

Broadcast Group

```
function broadcast(){
//Gestion des messages avec un tableau contenant les messages
liste_messages[0].push(data.sender + ' : ' + data.msg);
liste_messages[1].push(socket);
liste_messages[2].push(data.group);

//Gestion des évènements avec un tableau contenant
operations[0].push(data.sender + " : " + data.msg);
operations[1].push(data.group);

if(testpres()==true) //si le groupe existe et qu'il fait partie du groupe
{
  for(i=0;i<groups[0].length;i++)
  {
    if(groups[2][i]==data.group)
    {groups[1][i].emit('message', { sender: data.sender, group: data.group, msg: data.msg, event: 'group.broadcast'});}
  }
}
else
{ socket.emit('message', { sender: data.sender, group: data.group, msg: data.msg, event: 'group.broadcast.not.in.group'});}
}
```

Pour cette fonctionnalité, on teste si le groupe dans lequel le message doit être envoyé existe et si l'utilisateur fait partie du groupe. Si c'est le cas, on envoie à tous les utilisateurs du tableau qui ont pour nom de groupe celui qui a été mentionné dans la commande. Si on ne se trouve pas dans ce cas, on refuse l'envoi d'un message.

List members

```
function membres(){
//Gestion des évènements
operations[0].push(data.sender + " a affiché la liste des membres du groupe " + data.group);
operations[1].push(data.group);

//on stocke les membres du groupe demandé dans un tableau
var listemembres = [];
function memberlist()
{
  for(i=0;i<groups[0].length;i++)
  {
    if(groups[2][i]==data.group)
    {listemembres.push(groups[0][i]);}
  }
  return listemembres;
}

socket.emit('message', { sender: data.sender, tab : memberlist(), group: data.group, event: 'group.members'});
}
```

Pour afficher la liste des membres appartenant à un groupe, j'ai décidé d'adopter la stratégie suivante : créer un nouveau tableau listemembres et stocker dans ce tableau tous les utilisateurs

associés à endroit dans le tableau groups. Après le parcours de ce tableau terminé, on peut afficher listemembres.

List messages

```
function messages(){
  //Gestion des évènements
  operations[0].push(data.sender + " a affiché l'historique des messages du groupe " + data.group);
  operations[1].push(data.group);

  //on stocke les messages du groupe demandé dans un tableau
  var liste_messages_groupe = [];
  for(i=0;i<liste_messages[0].length;i++)
  {
    if(liste_messages[2][i]==data.group)
    {liste_messages_groupe.push(liste_messages[0][i]);}
  }
  socket.emit('message', { sender: data.sender, tab: liste_messages_groupe, group: data.group, event: 'group.messages'});
}
```

Pour cette fonctionnalité, on crée un tableau liste_messages_groupe. Dans la fonctionnalité broadcast, on push dans un tableau liste_messages chaque message qui a été envoyé, quel que soit l'utilisateur, quel que soit son groupe. Ainsi, on peut parcourir ce tableau et on push dans list_messages_groupe uniquement les messages correspond au nom de groupe mentionné dans la commande. Enfin, on peut le renvoyer pour pouvoir l'afficher.

List groups

```
var listegroupes = []; //tableau qui va contenir la liste des groupes existants
function grouplist()
{
  for(i=0;i<groups[0].length;i++) //on parcourt les groupes et on les ajoute dans un tableau en évitant les doublons
  {
    if(listegroupes.includes(groups[2][i])==false)
    {listegroupes.push(groups[2][i]);}
  }
  return listegroupes;
}

socket.emit('message', { sender: data.sender, tab : grouplist(), event: 'group.list'});
}
```

On crée un tableau listegroupes qui contiendra la liste des groupes. On parcourt le tableau groups et on ajoute le nom du groupe lu dans listegroupes. On vérifie à chaque itération que le tableau listegroupes ne contient pas déjà le groupe pour ne pas avoir de doublons et on affiche la liste.

Leave

```
function quitter() {  
  //Gestion des évènements  
  operations[0].push(data.sender + " a quitté le groupe " + data.group);  
  operations[1].push(data.group);  
  
  for(i=0;i<groups[0].length;i++)  
  {  
    if(groups[0][i]==data.sender && groups[2][i]==data.group)  
    {  
      groups[0].splice(i,1); //On supprime le nom  
      groups[1].splice(i,1); //On supprime la socket  
      groups[2].splice(i,1); //On supprime le groupe  
    }  
  }  
  
  socket.emit('message', { sender: data.sender, group: data.group, event: 'group.leave'});  
}
```

Pour la fonctionnalité « leave », il faut ici parcourir le tableau groups. On regarde la correspondance avec l'utilisateur qui demande à partir et son groupe, et on le supprime du tableau groups avec la commande splice().

Invite

```
function inviter(){  
  //Gestion des évènements  
  operations[0].push(data.sender + " a invité " + data.dest + " dans le groupe " + data.group);  
  operations[1].push(data.group);  
  //console.log(bannis);  
  console.log(testbandest());  
  if(testbandest()==false)  
  {  
    groups[0].push(data.dest);  
    groups[2].push(data.group);  
    socket.emit('message', { sender: data.sender, group: data.group, dest: data.dest, event: 'group.invite'});  
  }  
}
```

Pour inviter un utilisateur, on vérifie que l'utilisateur destinataire n'est pas banni du groupe demandé. Si tel est le cas, alors on peut l'ajouter dans le tableau groups avec le nom du groupe associé.

Kick

```
function exclure() {
  //Gestion des évènements
  operations[0].push(data.sender + " a exclu " + data.dest + " du groupe " + data.group);
  operations[1].push(data.group);

  for(i=0;i<groups[0].length;i++)
  {
    if(groups[0][i]==data.dest) //si le dest correspond
    {
      groups[0].splice(i,1); //on supprime nom
      groups[1].splice(i,1); //on supprime socket
      groups[2].splice(i,1); //on supprime groupe
    }
  }
  socket.emit('message', { sender: data.sender, group: data.group, dest: data.dest, reason: data.reason, event: 'group.kick' });
}
```

Il y a une grande ressemblance ici avec la fonctionnalité « leave ». La seule différence est qu'au lieu de regarder la correspondance avec l'utilisateur qui envoie la commande, on regarde la correspondance avec l'utilisateur destinataire. On le supprime du tableau groups avec splice().

Ban

```
function bannir() {
  //Gestion des évènements
  operations[0].push(data.sender + " a banni " + data.dest + " du groupe " + data.group);
  operations[1].push(data.group);

  for(i=0;i<groups[0].length;i++)
  {
    if(groups[0][i]==data.dest && groups[2][i]==data.group) //si le nom du dest et le groupe correspondent
    {
      bannis[0].push(data.dest); //on push dans le tableau des bannis
      bannis[2].push(data.group);
      groups[0].splice(i,1); //on supprime dans le tableau des utilisateurs
      groups[2].splice(i,1);
    }
  }
  socket.emit('message', { sender: data.sender, group: data.group, dest: data.dest, reason: data.reason, event: 'group.ban' });
}
```

Ici, en plus d'exclure l'utilisateur destinataire en le supprimant du tableau groups, on l'ajoute dans un nouveau tableau nommé bannis. Quand l'utilisateur voudra rejoindre un groupe ou quand il sera invité dans un groupe, on vérifiera d'abord qu'il n'est pas présent dans le tableau des bannis.

Unban

```
function debannir() {
  //Gestion des évènements
  operations[0].push(data.sender + " a débanni " + data.dest + " du groupe " + data.group);
  operations[1].push(data.group);

  for(i=0;i<bannis[0].length;i++)
  {
    if(bannis[0][i]==data.dest && bannis[2][i]==data.group) //si le nom du dest et le groupe correspondent
    {
      bannis[0].splice(i,1); //on supprime dans le tableau des bannis
      bannis[2].splice(i,1);
    }
  }
  socket.emit('message', { sender: data.sender, group: data.group, dest: data.dest, event: 'group.unban' });
}
```

Pour débannir un utilisateur, il suffit simplement de le supprimer du tableau bannis avec la commande splice().

States

```
function evenements(){
  //Gestion des évènements
  operations[0].push(data.sender + " a affiché l'historique des opérations du groupe " + data.group);
  operations[1].push(data.group);

  var operations_groupe = [];
  for(i=0;i<operations[0].length;i++)
  {
    if(operations[1][i]==data.group) //si le nom correspond au groupe que l'on veut
    {
      operations_groupe.push(operations[0][i]); //on ajoute les opérations d'un groupe dans un tableau
    }
  }
  socket.emit('message', { sender: data.sender, group: data.group, event: 'group.states' });
}

});
```

Pour la fonctionnalité « states », on crée un tableau nommé operations. Dans chaque fonctionnalité codée jusqu'ici, on remarquera à chaque fois, on push la description du résultat de la commande ainsi que le groupe dans lequel elle a été faite. On pourra alors parcourir operations en regardant le groupe qui nous intéresse et afficher toutes les événements qui ont eu lieu dans ce groupe.

Base de données :

J'ai pu implémenter la fonctionnalité de base de données mentionnée dans le TD4.

```
var sq = require('sqlite3');
```

Je commence par utiliser require pour pouvoir utiliser les base de données.





```
sq.verbose();

var db = new sq.Database('bdd.db', err => {
  if (err)
  {
    throw err
  }
  else { console.log('Database started on bdd.db')}
});
```

Le code ci-contre me permet de créer une base de données et de vérifier qu'elle a bien été créée. J'affiche que la base de données a bien été créée si c'est le cas et sinon j'affiche une erreur.

```
ayoub@DESKTOP-B2JEDII MINGW64 ~/td5-ayoub1k/src/server (master)
$ node server.js
Database started on bdd.db
```

En exécutant le programme, on remarque que la base de données a bien été créée.

Nom	Mo
 bdd	01/
 package	24/
 package-lock	24/
 server	01/

On peut facilement le vérifier en vérifiant dans le dossier de création. Ici, le fichier bdd.db a bien été créé après l'exécution de la commande.

```
db.run('CREATE TABLE message (content TEXT, nom_user TEXT, nom_groupe TEXT)');
```

En utilisant CREATE TABLE, il est possible de créer une table nommée message qui va contenir les messages envoyés, les noms des utilisateurs qui ont envoyé le message ainsi que le groupe dans lequel il a été envoyé.

```
}
db.run("INSERT INTO message VALUES(?,?,?)", [data.msg, data.sender, data.group]);
}
```

Dans la fonctionnalité de broadcast, on peut alors mettre la ligne de code ci-dessous avec un INSERT INTO. Dans la table, on insère le message, l'expéditeur du message et le groupe dans lequel le message a été envoyé. A chaque fois qu'un utilisateur enverra un message, la base de données sera alors actualisée.

```
case 'bdd':
  db.get('SELECT * FROM message', (err, data) => {
    if(err)
    {throw err;}
    else { console.log(data);}
  });
  break;
```

On crée un nouveau case dans le switch qui va permettre à l'utilisateur de demander l'affichage de la base de données. On fait ici un SELECT pour récupérer les informations que l'on souhaite.

```
> cg;lol  
> ayoub a créé le groupe lol  
> bg;lol;hey  
> ayoub>hey  
> bdd;
```

Ici, on crée un groupe dans lequel l'utilisateur envoie le message « hey ». Ensuite, il utilise la commande bdd;

```
ayoub@DESKTOP-B2JEDII MINGW64 ~/td5-ayoub1k/src/server (master)  
$ node server.js  
Database started on bdd.db  
{ content: 'hey', nom_user: 'ayoub', nom_groupe: 'lol' }  
|
```

La base de données avec les informations demandées est alors affichée.