MENARD Victor & BELLOUK Ayoub
ESIR2 - IoT

IHMW - Rapport de projet
Enseignants : David Bromberg & Johann Bourcier
Encadrant : Johann Bourcier

"Nous attestons que ce travail est original et qu'il fait référence aux sources utilisées"

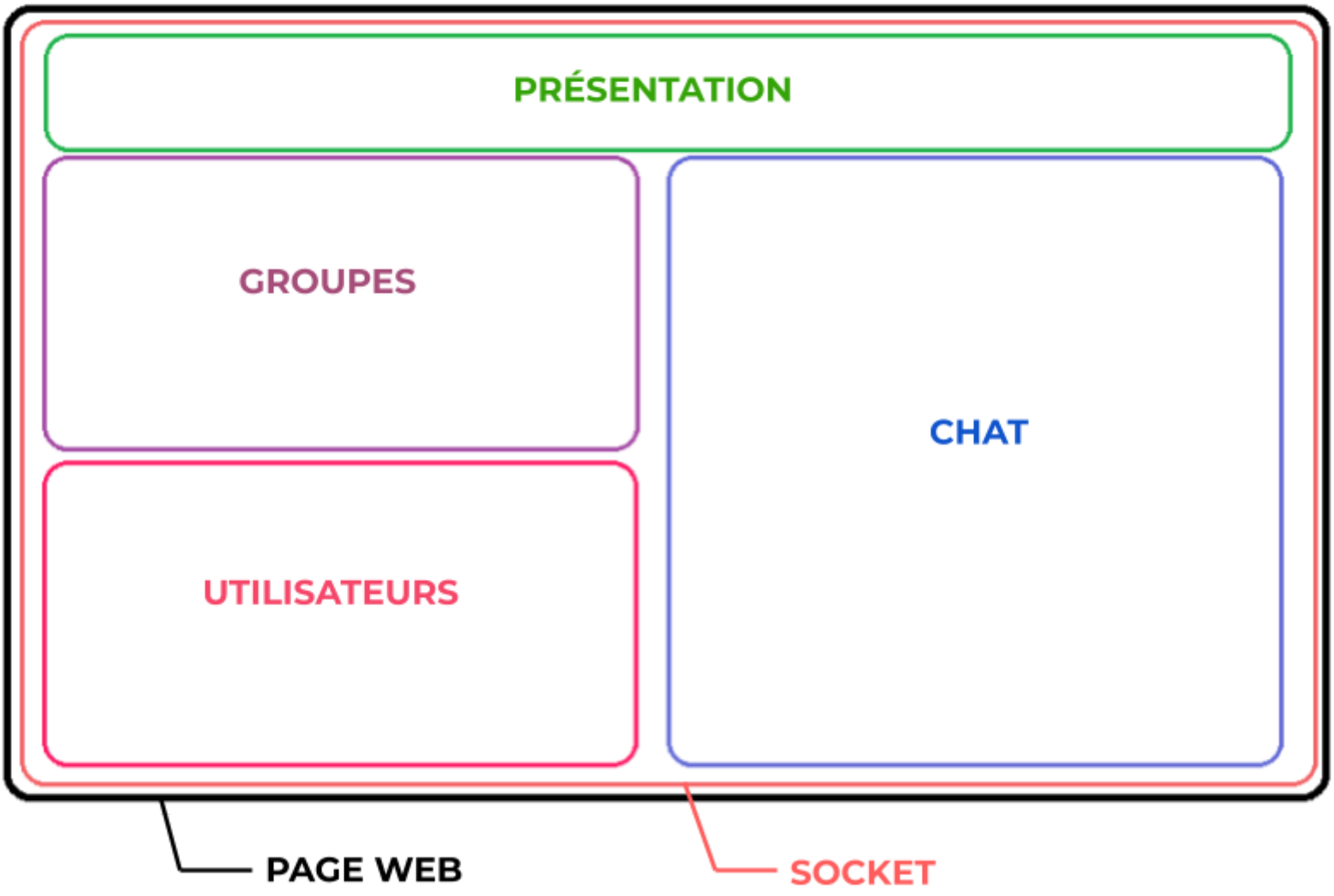
Avant-propos

L'objectif de ce projet est de construire une interface de chat grâce à REACTJS et en utilisant le serveur socket.io que nous avions développé en WOB au premier semestre.
Les fonctionnalités du serveur sont les suivantes :

Intitulé	Effet de la commande
Broadcast	Envoyer un message à tous les utilisateurs
List	Afficher la liste de tous les utilisateurs
Quit	Se déconnecter
Create Group	Créer un groupe
Join	Joindre un groupe
Broadcast Group	Diffuser un message à tous les utilisateurs d'un groupe
List members	Afficher les membres d'un groupe
List messages	Afficher l'historique des messages d'un groupe
List groups	Afficher la liste des groupes
Leave	Quitter un groupe
Kick	Exclure un utilisateur d'un groupe
Ban	Bannir un utilisateur d'un groupe
Unban	Débannir un utilisateur d'un groupe

On veut pouvoir utiliser ces fonctionnalités de manières graphiques, alors que l'on pouvait y avoir accès par commande sur le terminal précédemment.

Structure de l’interface



Notre interface web est divisée en plusieurs *Components*, comme l’utilisation de React nous permet de le faire. En voici la liste :

- Presentation : Ce component se trouve tout en haut de la page. Elle contient 3 éléments. Le premier élément est l’encart qui permet à pour se connecter ainsi qu’un logo correspondant aux couleurs de l’utilisateur de saisir son pseudonyme. Il y a également un bouton l’ESIR.

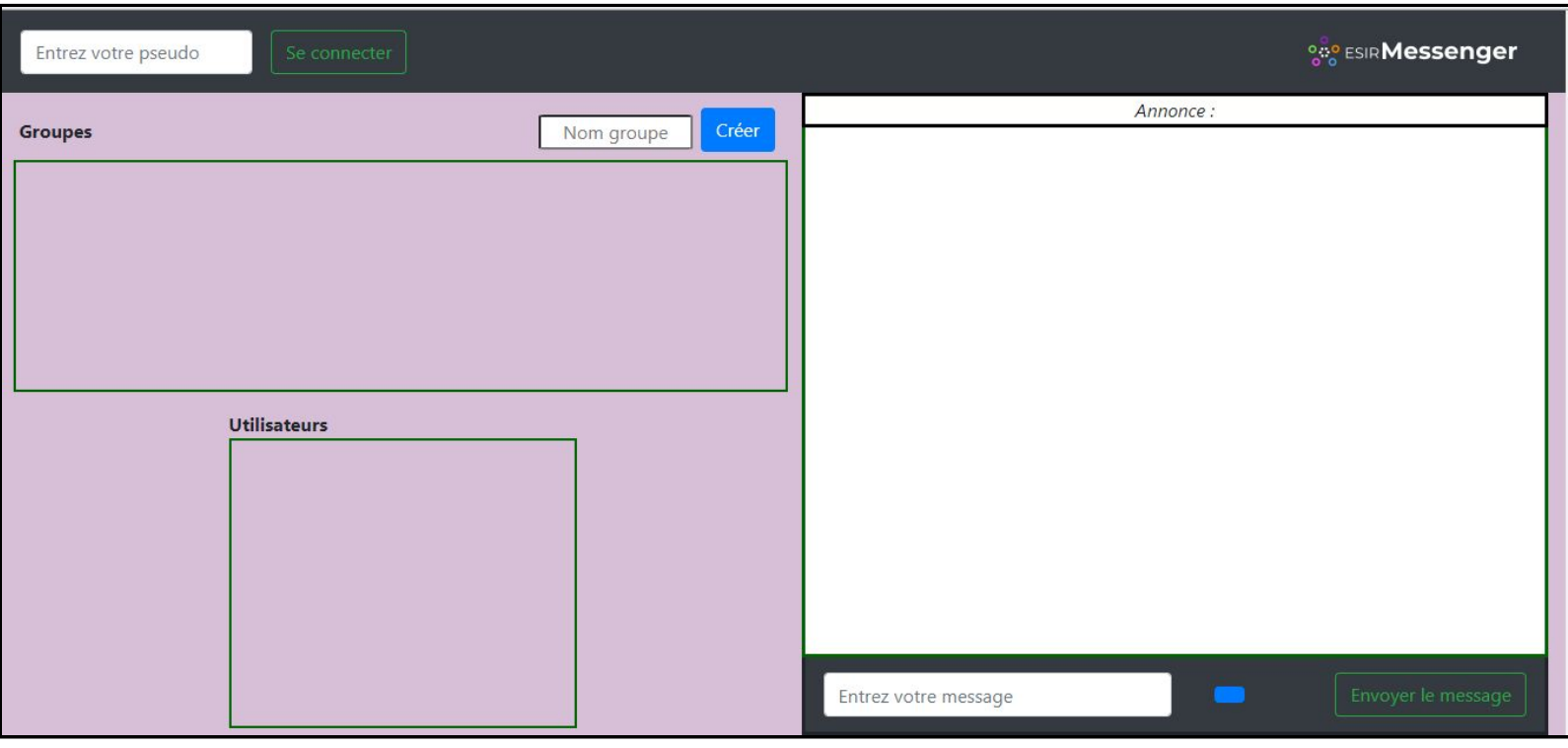
- Chat : Ce component se trouve dans la partie droite de la page. Il contient une zone pour les messages reçus et et les messages envoyés par l’utilisateur. Il y a également une zone d’écriture des messages, un emplacement prévu pour l’affichage du pseudonyme

sur lequel on peut cliquer pour se déconnecter, un bouton pour envoyer le message ainsi qu'une zone d'annonce qui affiche les actions importantes qui ont été faites par les utilisateurs du chat.

- Groupes : Il s'agit d'une tableau avec une zone permettant de créer un groupe (input + bouton). On peut y avoir la liste des groupes qui s'actualise en temps réel. Plusieurs boutons sont associés aux groupes pour effectuer différentes actions. Ils permettent de rejoindre un groupe, de quitter un groupe, d'envoyer un message aux membres d'un groupe, d'afficher les membres appartenant à un groupe. Il est également possible de bannir un utilisateur d'un groupe.

- Utilisateurs : Il s'agit d'une tableau contenant un bouton d'actualisation qui permet d'afficher les utilisateurs connectés. A côté de chaque pseudo d'utilisateur, on retrouve un bouton "Envoyer" associé pour envoyer un message privé.

Aperçu de la page web :



Organisation

Tableau de répartition des tâches :

	Ayoub	Victor
Component Presentation	Gestion du design Création du logo	Connexion des utilisateurs
Component Groupe	Programmation des fonctionnalités Create Groupe, Join, Quit Actualisation en direct des groupes, Design	Programmation des fonctionnalités Broadcast Group, Members
Component Chat	Gestion des annonces pour les fonctionnalités Rejoindre et Quitter	Gestion des messages envoyés et reçus, broadcast, affichage du pseudo, Design
Component Utilisateurs	Programmation de la fonctionnalité Send privé, design	Gestion de l'actualisation des utilisateurs
Autres	Débug, Veille sur les technologies de ReactJS, Structure du code, Rédaction du rapport, Communication entre les components, Gestion du CSS, Commits Github	

Fonctionnement du code

- La classe socket.js permet d'assurer la communication avec le serveur : on écoute sur le port 4000.

```
import socketIOClient from "socket.io-client";
import React from "react";

var url = "http://localhost:4000";

export const socketclient = socketIOClient(url);
```

- La classe App.js contient les components et on peut gérer l'agencement entre eux avec le code HTML et le code CSS. On importe au début du code tous les components en précisant leur chemin.

```
<div>
<Presentation functionCallFromParent={this.parentFunction.bind(this)}/>
<Socket valuefromparent={this.state.user}/>
<div id='gauche'>
<Groupe valuefromparent={this.state.user}/>
<Utilisateurs valuefromparent={this.state.user}></Utilisateurs>
</div>

<div id='droite'>
<Container fluid> <Chat valuefromparent={this.state.user}/> </Container>
</div>
</div>
```

- Chaque component dispose de son code CSS pour gérer leur apparence

- On stocke les informations nécessaires dans des states ou props qu'on communique au serveur avec socket.emit et auxquelles les components pertinents peuvent accéder. Exemple :

```
constructor(props){
  super(props);
  this.state={
      user:"",
      Chat:[]
  }
}
  parentFunction=(data_from_child)=>{
    this.setState({user:data_from_child});
  }
```

Connexion :
Au clic de "Se connecter", pour se connecter au chat :

```
  childFunction=(e)=>{

    e.preventDefault();
    this.props.functionCallFromParent(this.state.user);
    socketclient.emit('send', {'action':'client-connexion', 'sender':this.state.user});
    document.getElementById('text-input').value = '';

}
```

Groupe :
Pour gérer les groupes, un tableau auquel on ajoute des lignes et des cellules au fur et à mesure a été créé.

Au clic de "Créer", pour créer un groupe :

```
socketclient.emit('send', {'action' : 'cgroup', 'sender': this.props.valuefromparent, 'groupe':this.state.groupe })
```

Au clic de "Rejoindre", pour rejoindre un groupe :

```
socketclient.emit('send', {'action':'join', 'sender': this.props.valuefromparent, 'groupe': groupe})
```

Au clic de “Quitter”, pour quitter un groupe :

```
socketclient.emit('send', {'action':'leave', 'sender': this.props.valuefromparent, 'groupe': groupe})
```

Au clic de “Envoyer”, pour envoyer un message à un groupe :

```
socketclient.emit('send', {'action':'gbroadcast', 'sender': this.props.valuefromparent, 'groupe':groupe, "message":
```

Au clic de “Membres”, pour afficher les membres d’un groupe :

```
socketclient.emit('send', {'action':'members', 'sender': this.props.valuefromparent, 'groupe':groupe});
```

Après inscription du pseudonyme pour bannir un utilisateur :

```
socketclient.emit('send', {'action':'ban', 'sender': this.props.valuefromparent, 'groupe': ban.className, 'dest': ban.value})
```

Après inscription du pseudonyme pour débannir un utilisateur :

```
socketclient.emit('send', {'action':'deban', 'sender': data.sender, 'groupe':data.groupe,'dest':data.dest}
```

Utilisateurs :

Pour gérer les utilisateurs, un tableau qu’on actualise à l’aide d’un bouton a été créé.

Au clic de “Se connecter”, pour afficher les utilisateurs connectés :

```
socketclient.emit('send', {'action':'client-list-clients'});
```

Chat :

On utilise un tableau pour gérer les messages envoyés et reçus par les utilisateurs du chat. A l’envoi du message, on les dispose dans le compartiment adapté.

Au clic de “Envoyer le message”, pour envoyer un message à tout le monde :

```
socketclient.emit('send', { 'action':'client-broadcast', 'msg': this.state.msg, 'sender': this.props.valuefromparent});
```

Au clic du pseudonyme, pour se déconnecter :

```
onClick={a => socketclient.emit('send', { 'action': 'client-quit' , 'sender': this.props.valuefromparent})}
```

Une fois les requêtes émises, on écoute les requêtes avec `socket.on` dans lequel on met un `switch` pour gérer les différents cas. Ensuite, on affiche le résultat sur la page web sur les bons components et pour les utilisateurs pertinents pour la fonctionnalité choisie.

```
socketclient.off('message').on('message', function (data) {

switch(data.action){
```

Difficultés rencontrées

Messages dupliqués

Au commencement du projet, après avoir connectés le client et le serveur. Nous nous sommes rendus compte d'un problème. A chaque fois que l'on lançait une commande, elle était exécutée plusieurs fois. Par conséquent, tous les messages envoyés, les groupes créés, les utilisateurs connectés étaient dupliqués. Il a été très compliqué de comprendre d'où venait le problème. Après de multiples recherches, la solution proposée sur ce site web : https://dev.to/bravemaster619/how-to-prevent-multiple-socket-connections-and-events-in-react-531d qui était de limiter le bind du listener à une fois a été payante.

Gestion des binds

C'est un aspect primordial et n'est pas propre à ReactJS. Cela n'a pas été aisé de comprendre. Cela permet de gérer les évènements des éléments. On peut *"associer des données d'un élément à un autre élément"* comme cela est expliqué sur openclassroom : https://openclassrooms.com/fr/courses/4902061-developpez-une-application-mobile-react-native/5095876-apprivoisez-le-data-binding.

Cycle de vie des components

C'est un concept qui a également été compliqué à comprendre et que nous n'avons pas encore totalement assimilé. React nous permet de gérer les étapes de vie des components. En pratique, les méthodes les plus utilisées pour gérer ce cycle de vie sont constructor() ainsi que render(). Il en existe d'autres comme componentWillMount() et componentDidMount() comme expliqué sur openclassroom : https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664881-apprivoisez-le-cycle-de-vie-des-composants.

Communication entre les components

En programmant avec ReactJS, il faut prendre en compte la notion de props, de state, de component Parent et de component Enfant notamment. La compréhension de ces notions permet de gérer la communication entre les components. Par exemple, pour communiquer une donnée d'un component parent à un component enfant, on utilisera les props comme expliqué sur pluralsight : https://www.pluralsight.com/guides/react-communicating-between-components

Améliorations possibles

La première amélioration que l'on aimerait apporter est la gestion des commandes écrites dans le chat comme on pouvait le faire avec le client au premier semestre dans le terminal en wob. Voici les commandes que l'on aurait utilisé. On remarque l'utilisation d'un point d'exclamation au début qui aurait permis de signaler dans le code du client qu'une commande écrite va être faite.

Intitulé	Commande
Broadcast	!b;[messages]
List	!ls;
Quit	!q;[groupe]
Create Group	!cg;[groupe]
Join	!j;[groupe]
Broadcast Group	!bg;[groupe]
List members	!members;[groupe]
List messages	!messages;[groupe]
List groups	!groups;
Leave	!leave;[groupe]
Ban	!ban;[pseudo];[groupe]
Unban	!unban;[pseudo];[groupe]

- Une amélioration majeure que l'on aimerait également apporter serait de gérer toutes les données primordiales (groupes, messages, utilisateurs...) avec des states et des props. En effet, nous avions commencé à gérer cela avec du code javascript pur ce qui n'est pas approprié. Nous avons commencé à apporter des modifications que nous n'avons pas pu terminer.

- Enfin, une dernière amélioration qu'il serait opportun d'ajouter est la rédaction de fonctions de tests en utilisant les technologies Jest, Chai, et Enzyme. Nous nous sommes un peu penché sur la question, mais nous n'avons pas eu le temps d'assimiler le concepts et de les implémenter.

Conclusion

Ainsi, au cours de ce projet, nous avons pu nous familiariser avec la programmation ReactJS et ses concepts en concevant une interface de chat avec un serveur conçu en wob. Ce projet nous a permis de consolider nos compétences en matière de travail d'équipe, qui plus est, complètement différent que dans un contexte habituel puisque entièrement fait à distance.