

# INTERACTIVE CONFERENCE

RAOUL Bastien - SAUNIER Nicolas - CLAIRAND Maxime - ROBIN Eliott - PEDICONE Doucelin

## Guide Développeur

---

### SOMMAIRE

|   |           |
|---|-----------|
| <b>I - Introduction</b>                       | <b>2</b>  |
| <b>II - Technologies utilisées</b>            | <b>2</b>  |
| NODE.JS                                       | 2         |
| SOCKET.IO                                     | 3         |
| POSTGRESQL                                    | 3         |
| Outils utilisés pour le projet                | 3         |
| <b>III - Arborescence du projet</b>           | <b>4</b>  |
| <b>IV - Modélisation</b>                      | <b>5</b>  |
| Diagramme de cas d'utilisation                | 5         |
| Diagramme de classe (Base de données)         | 5         |
| Diagramme de séquence                         | 6         |
| A - Diagramme de séquence : Connexion         | 6         |
| B - Diagramme de séquence : Messages          | 6         |
| Diagramme d'activité (événements)             | 7         |
| <b>V - Version et Bibliothèques utilisées</b> | <b>7</b>  |
| <b>VI - Explication code</b>                  | <b>8</b>  |
| Design  | 8         |
| Système de salons                             | 8         |
| Chat  | 10        |
| Quizz   | 10        |
| Base de données                               | 11        |
| Exportation des données du salon              | 11        |
| Les votes                                     | 12        |
| <b>VII - Fonctionnement du responsive</b>     | <b>13</b> |
| <b>VIII - Évolutions possibles</b>            | <b>14</b> |

---

## I - Introduction

Ce guide a pour objectif d'expliquer le projet "Interactive Conference" afin d'indiquer la démarche à suivre pour chaque personne souhaitant comprendre et reprendre le projet dans le cadre d'une poursuite de développement.

L'outil doit fournir une plate-forme web anonyme de discussion ouverte où les étudiants peuvent parler librement, poser des questions et exprimer leurs opinions avec les autres en votant pour leurs messages. Cette plate-forme doit fournir plusieurs espaces de discussions qui sont indépendants les uns des autres que l'on appellera "salon".

Interactive Conference est un site web. Ce site web adapte sa forme selon la taille de l'interface que l'utilisateur utilise, car durant les cours magistraux, la majorité des utilisateurs (les étudiants) seront amenés à utiliser Interactive Conference sur mobile.

Naturellement, le site web est codé en HTML & CSS, ainsi qu'en Javascript. Le site web est installé sur un serveur avec Node.js qui est lui-même hébergé sur la plateforme Heroku, qui nous permet d'héberger gratuitement notre solution. À terme, l'utilisateur pourra prévoir un abonnement payant pour héberger la solution sur une autre plateforme. Quant au système de chat, le site utilise Socket.IO ainsi que Postgresql pour la base de données. Les différentes bibliothèques utilisées seront décrites plus tard dans ce guide.

## II - Technologies utilisées

### NODE.JS

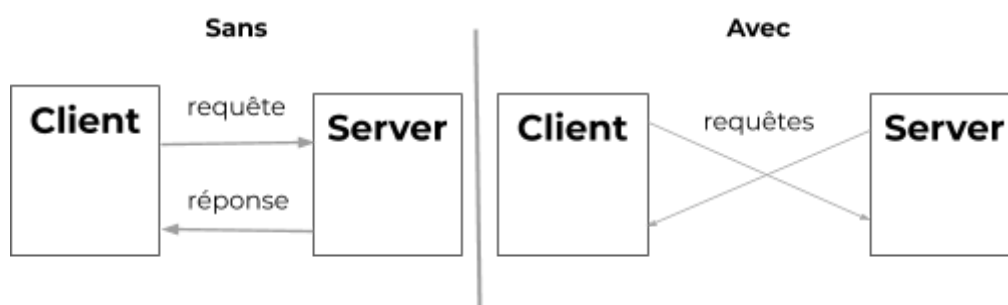
Node.js permet de développer un site web dynamique et rapide en utilisant le langage Javascript côté serveur. Le choix pour Node.js s'est naturellement fait car notre projet utilise beaucoup d'événements ("un utilisateur s'est connecté", "un message arrive sur le serveur", "envoyer le message aux autres utilisateurs", etc...).



## SOCKET.IO

Socket.IO est une bibliothèque Javascript qui permet de développer des applications dites "en temps réel". Cette caractéristique signifie qu'il y a une prise en compte de contraintes temporelles le système ne doit donc pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés. Socket.IO est également utilisé pour le système de diffusion (en anglais "broadcasting") qui est une méthode de transmission de données à l'ensemble des machines d'un réseau. Cette bibliothèque est donc adaptée pour créer une application de chat.

Socket.IO utilise WebSocket, une API de Javascript, qui permet un échange bilatéral synchrone entre le client et le serveur. En pratique, cela permet au serveur et aux clients, d'échanger des requêtes simultanément et sans avoir à attendre de réponse. Websocket permet donc de laisser une sorte de "tuyau" de communication ouvert entre le client et le serveur.



## POSTGRESQL

















Postgresql est la solution de base de données qui s'est imposé à nous. En effet, nous voulions implémenter une base de données et Postgresql est l'un des seuls modules gratuits et compatibles avec la plateforme Heroku. Nous avons utilisé une base de données plutôt qu'un système de fichiers pour stocker nos données car nous avons besoin d'avoir des relations et des contraintes entre nos tables.

## Outils utilisés pour le projet

Pour la gestion du projet nous avons utilisé:

- **Github:** un service web d'hébergement et de gestion de développement.
- **Heroku:** une plateforme gratuite d'hébergement de site web.
- **Visual Studio Code/Atom/SublimeText:** des éditeurs de textes pour le développement.
- **Google Drive/Doc/Draw.io:** pour le stockage et la création des différents graphiques et documents associés au projet.

### III - Arborescence du projet

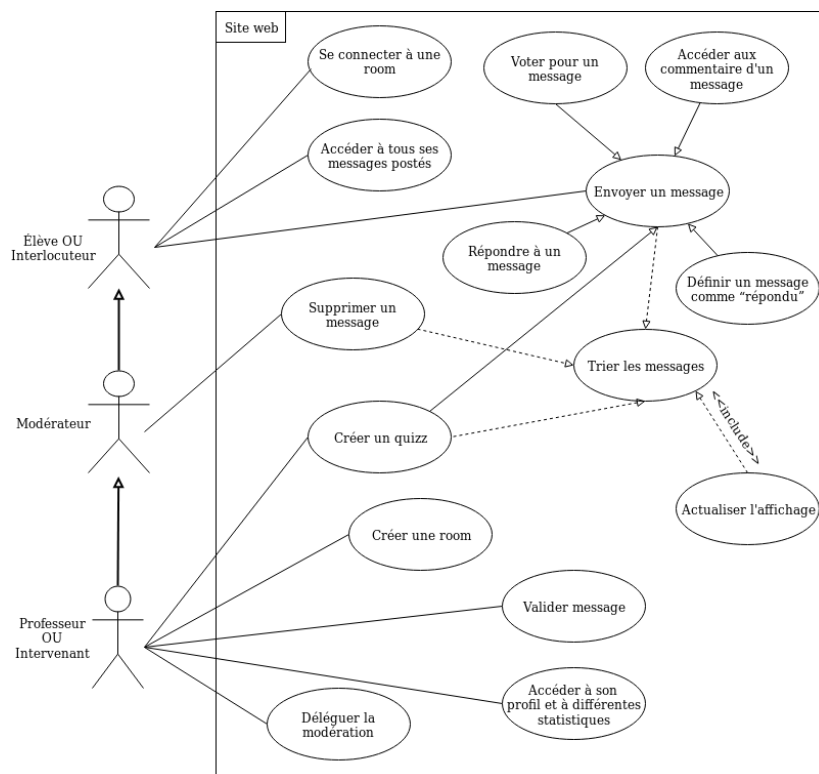
|   |  |
|---|--|
| interactive_conference  |  |
|  | .git   |
|  | home ————— Page principale (chat et creation de salon)                     |
|  | clientScript.js ————— Script groupant les éléments graphiques              |
|  | clientSocket.js ————— Script groupant les événements du socket             |
|  | index.html   ————— Page d'accueil (HTML+CSS)                               |
|  | style.css  |
|  | node_modules   |
|  | statistiques ————— Page «Statistiques»                                     |
|  | index.html   |
|  | script.js  |
|  | style.css  |
|  | npm-debug.log  |
|  | package-lock.json  |
|  | package.json   |
|  | README.txt ————— Fichier expliquant les manipulation liées au dépôt Github |
|  | server.js ————— Script exécuté sur le serveur                              |

#### Précisions:

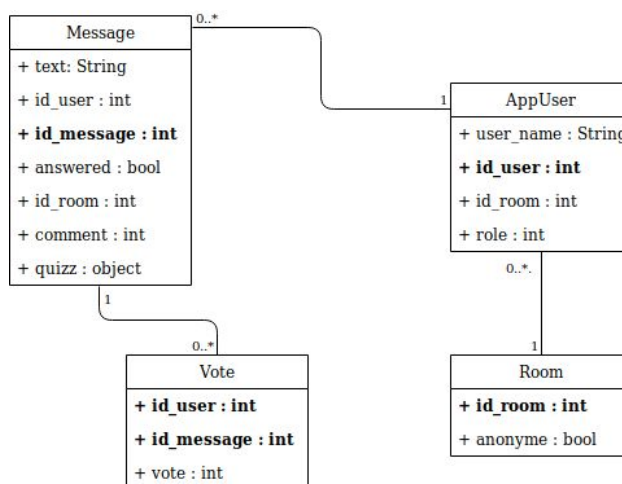
- **package.json**: Ce document décrit toutes les dépendances du projet (modules) accompagnées de leurs versions. Egalement, nous retrouvons les détails du projet: son nom, une courte description, sa version, et le script exécuté lors de l'exécution de la commande *npm start* (à savoir: **server.js**).
- Les fichiers **clientScript.js** et **clientSocket.js** pourraient être un seul fichier, nous avons créé cette séparation pour bien différencier les scripts "graphiques" (animations, système d'onglets, tiroir de navigation "drawer") des scripts liés aux différents événements déclenchés par le serveur ou par le client lui-même (envoi/réception de message, connexion à un salon, ...).

## IV - Modélisation

### Diagramme de cas d'utilisation

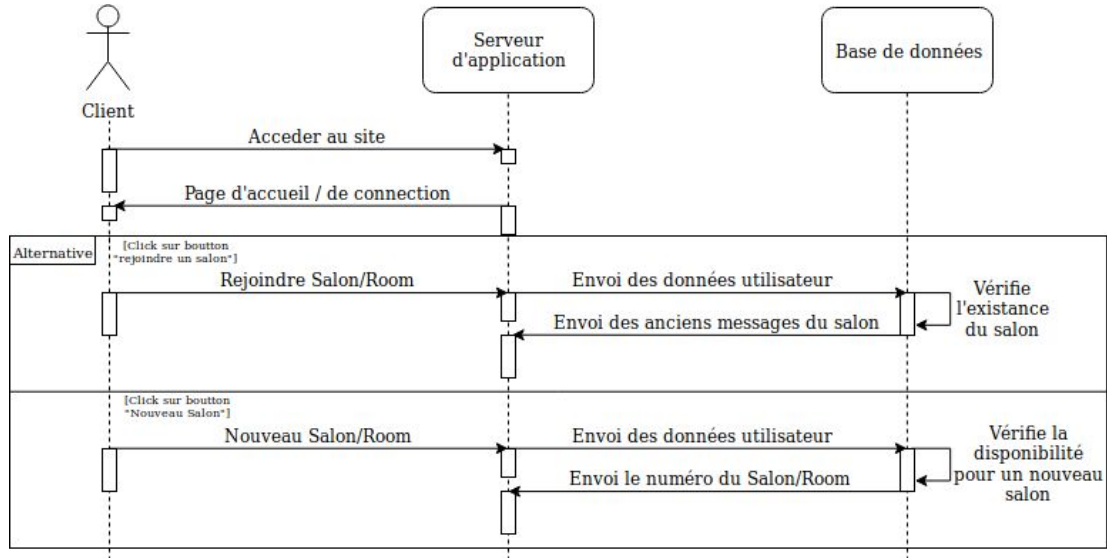


### Diagramme de classe (Base de données)

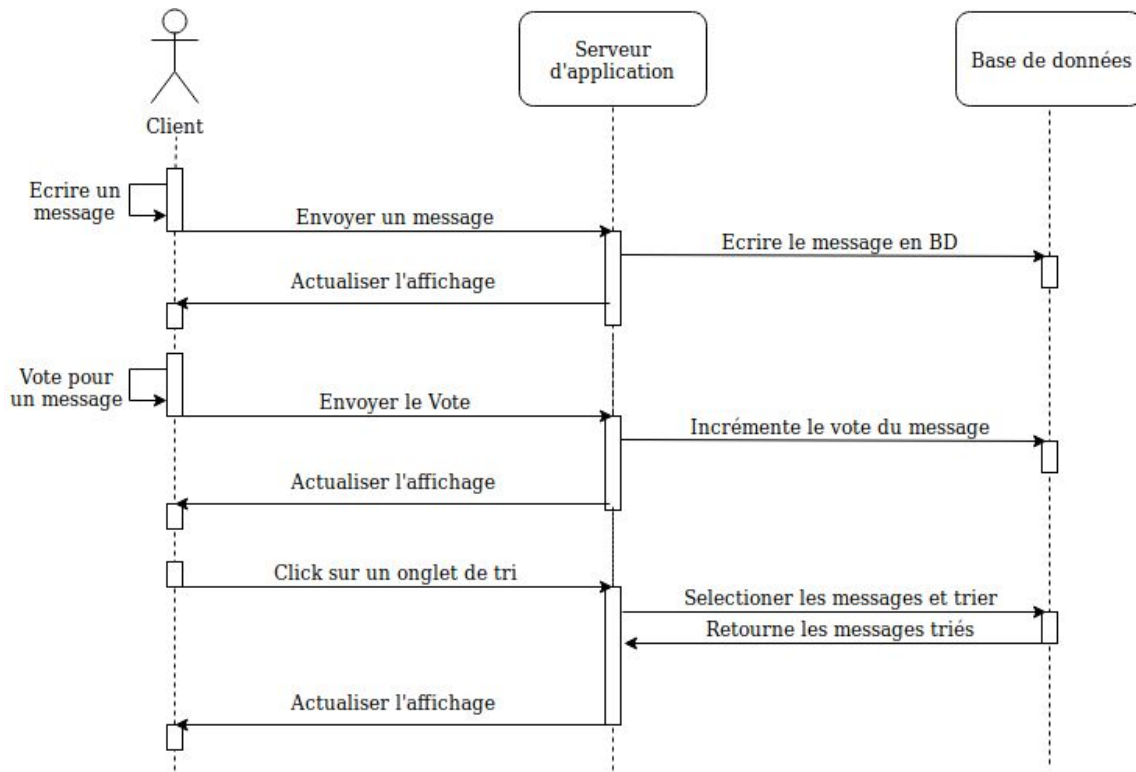


## Diagramme de séquence

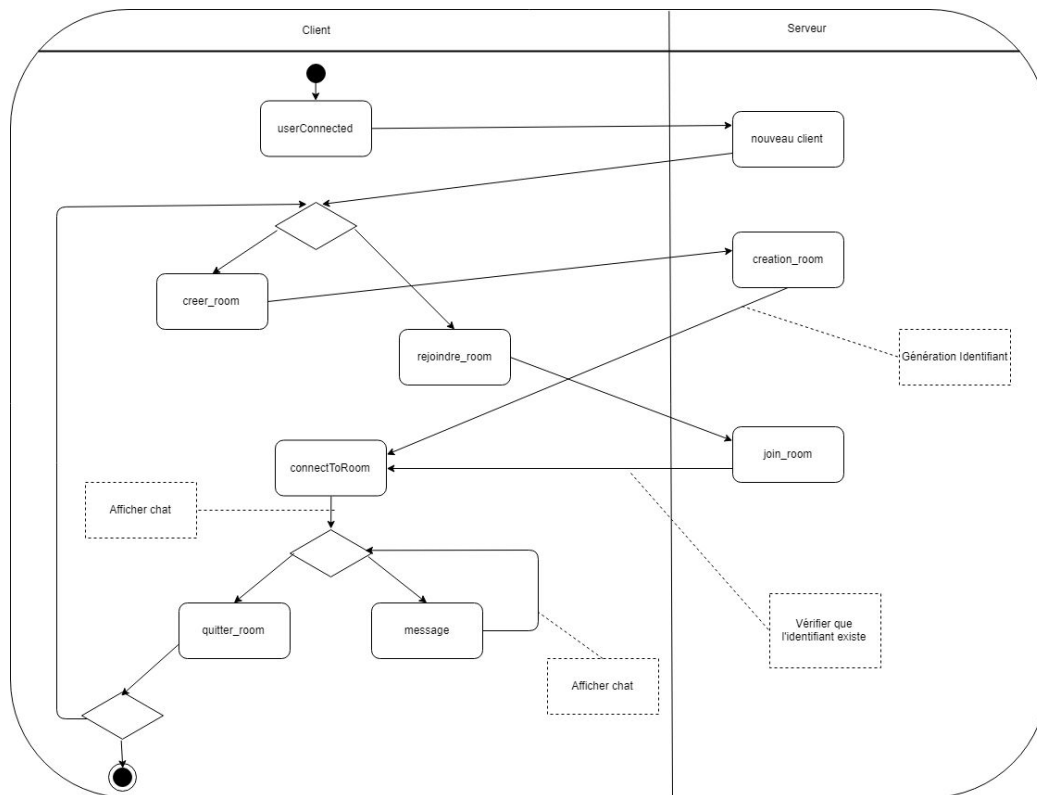
### A - Diagramme de séquence : Connexion



### B - Diagramme de séquence : Messages



## Diagramme d'activité (événements)



## V - Version et Bibliothèques utilisées

[github.com](https://github.com) - Projet

[Heroku](https://interactive-conference2018.herokuapp.com/) - Application (<https://interactive-conference2018.herokuapp.com/> )

[Google drive](#) - Documentation concernant le projet

[Material.io](#) - version 0.44.0

- [material-components-web.min.js](#)
- [material-components-web.min.css](#)
- [fonts.googleapis.com/icon?family=Material+Icons](https://fonts.googleapis.com/icon?family=Material+Icons)

Licenses:

- [authoring-components-for-mdc-web](#)
- [material-design-icons/LICENSE](#)

[jQuery](#) - version 1.11.1

- [jquery-1.11.1.js](#)

License:

- [jquery/LICENSE](#)

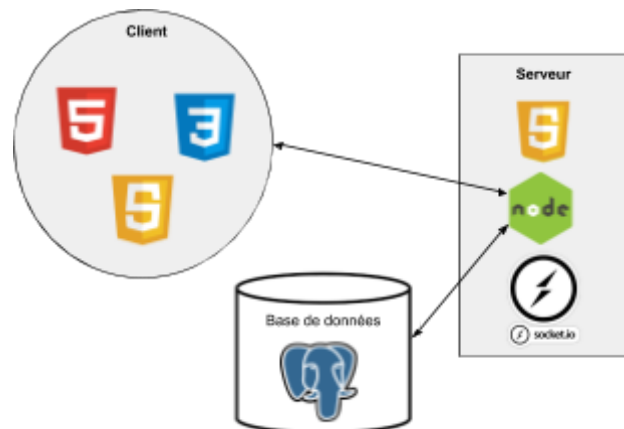
[Socket.IO](#) - version 2.1.1

[PostgreSQL](#) - version 7.7.1

[HTML 5](#) - [CSS 3](#)

[NodeJS](#) - version 8.10.0

[ExpressJS](#) - version 4.16.4

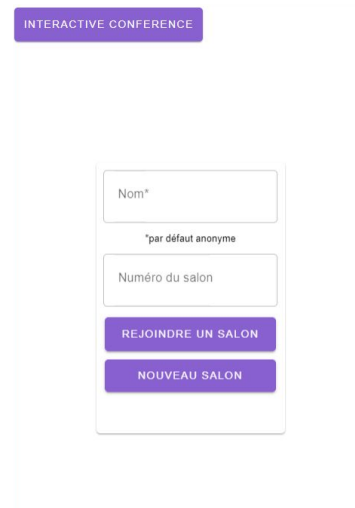


## VI - Explication code

### Design

Pour le design, nous avons décidé d'utiliser la bibliothèque "Material Design". C'est un ensemble de règles de design établies par Google et correspondant à une évolution du "Flat design". Pour faire du design qui suit les principes du "Material Design", Google a inventé la bibliothèque "[material.io](#)" que nous avons choisi d'utiliser, les avantages principaux de cette bibliothèque sont :

- La responsivité. En effet, notre site étant amené à être utilisé sur différents supports (ordinateur, tablette, smartphone...), il est important d'intégrer une certaine portabilité entre les différents supports et la bibliothèque permet de garder une cohérence globale.
- Le facilitement de la gestion des changements de l'interface initiés par les actions de l'utilisateur.
- Material Design fut premièrement développé pour smartphone, pour les applications Google puis fut développé pour le web. En utilisant Material Design nous permettons à l'utilisateur de retrouver des modèles de design connus et familiers pour lui.



### Système de salons

Créer un système de salon avec Socket.IO est assez facile. Dans notre application nous appelons salon un espace, caractérisé par un numéro d'identifiant, où tous les utilisateurs présents ont la possibilité d'exploiter les fonctionnalités de notre système tel que communiquer à travers des échanges de messages.



Comment avons-nous fait pour créer des salons, une fois que l'utilisateur appuie sur le bouton "Nouveau salon" ?

La première chose à savoir est que lorsque le client appuie sur un bouton peu importe lequel, un événement est déclenché côté client et une requête avec le nom de l'événement, et éventuellement différents paramètres, est envoyée au serveur. Ensuite, le serveur va effectuer la fonction associée et éventuellement appeler un nouvel événement ce qui va déclencher une action côté client.

```
//clientSocket.js
//Quand un utilisateur clic sur le bouton d'id "creer_salon"
$('#creer_salon').on('click', function() {
    //On déclenche l'événement "creation_salon", avec le pseudo en
    //paramètre, sur le serveur
    socket.emit('creation_salon', pseudo);
    //Puis on écoute la réponse du serveur avec les données qu'il nous
    //donne, grâce à l'événement "connectTosalon"
    socket.on('connectTosalon', function(salonID, userId) {
        //On met à jour la vue
        //...
    })
});
```

Revenons à la création d'un nouveau salon. Lors de la réception de la requête, le serveur va exécuter la fonction qui correspond à l'événement initier. Fonctionnellement, le serveur va générer un nouveau numéro aléatoire compris entre 1 et 1000 et qui est encore inutilisé par les salons déjà existants. Nous avons choisi 1000 de façon arbitraire car c'est un nombre suffisant pour réaliser nos tests. Dans le futur ce nombre devra être augmenté, mais devra rester assez court car chaque utilisateur voulant se connecter aura besoin de ce nombre identifiant, et donc moins il est long, plus il est simple à retenir et à diffuser.

Ensuite, le serveur ajoute le salon et l'utilisateur à la base de données, déclenche un nouvel événement et envoie une requête au client avec l'identifiant généré en paramètre. Le client stock ensuite localement l'identifiant du salon puis son interface est mise à jour pour lui faire "rejoindre" le salon créé. La méthode utilisée pour rejoindre un salon est la suivante : "socket.join(idsalon)" lors de l'appel, le client est ajouté à une liste côté serveur.

```
//Le serveur attend la réception de l'événement "creation_room"
socket.on('creation_room', function(pseudo) {
    var tempoId;
    // tempoId est numéro identifiant du salon généré ici
    socket.join(tempoId);
    console.log("Creation d'une room ID: "+tempoId);
    //...
    io.sockets.in(tempoId).emit('connectToRoom', tempoId);
    //...
```

```
});
```

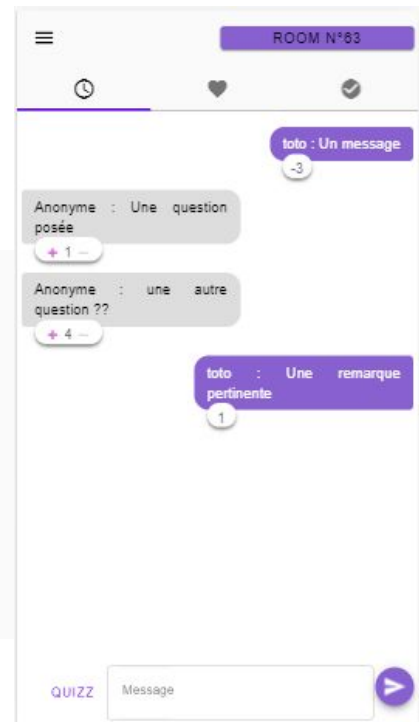
Au niveau de l'interface, l'utilisateur accède à plusieurs informations. S'il n'a pas encore rejoint de salon ou qu'il vient de le quitter, il verra affiché en haut à droite de son écran "accueil" sinon le message "salon N°ID" est affiché à la place.

Et si je veux rejoindre un salon ? Rien de plus simple, vous n'avez qu'à entrer le numéro du salon auquel vous voulez accéder. Et pour finir comment faisons-nous pour quitter un salon ? Il suffit d'utiliser le bouton "quitter" en haut à droite de l'écran qui fait déclencher la méthode : "socket.leave(idsalon)" lors de l'appel, le client est retiré de la liste côté serveur.

## Chat


Le système de chat est opérationnel. Il utilise, comme indiqué plus haut, Socket.IO qui nous permet l'envoi de messages en diffusion. Un utilisateur connecté à un salon recevra donc tous les messages envoyés par les autres utilisateurs du salon.

```
//server.js
socket.on('chat_message', function(id, message,
userId){
    //Écrit dans la console le msg
    console.log("(salon: "+id+" "+message);
    //Version simplifiée sans BDD
    socket.broadcast.to(id).emit('message',
    {pseudo: socket.pseudo, message: message});
});
```



## Quizz

L'implémentation du quizz a été commencée, en effet son design est réalisé, l'affichage se fait pour les différents utilisateurs cependant il manque la possibilité à l'utilisateur qui a créé le quizz de pouvoir le "fermer" et ainsi afficher la réponse. Une valeur est affichée en face de chaque proposition, et correspond aux nombres d'utilisateur ayant répondu avec ce choix, cette fonctionnalité n'est actuellement pas encore créée.

```
//server.js
socket.on(chat_quizz, function(id, question, userId){
  //Écrit dans la console le quizz
  console.log("(salon: "+id+") "+question);
  //Version simplifiée sans BDD
  socket.broadcast.to(id).emit('quizz', {question: question});
});
```

## Base de données

Dans notre projet nous avons choisi d'utiliser une base de données pour stocker et manipuler toutes les données relatives au fonctionnement de l'application. Actuellement, nous avons donc créé les tables présentées dans le diagramme de la quatrième partie. La base de données intervient dans la majorité des événements côté serveur et même dès le lancement du serveur pour connaître les salons existants dans la base de données :

```
//actualise le tableau roomno a chaque lancement du serveur grâce à la base de données
client.query("SELECT id_room FROM room;", (err, res) => {
  if (err) throw err;
  console.log(res);
  res.rows.forEach(function(element) {
    roomno.push(element.id_room);
  });
});
```

Nous sélectionnons tous les numéros de salons présents dans la base de données puis nous les ajoutons dans un tableau pour les stocker côté serveur et ainsi limiter le nombre de requêtes dans le cadre d'une création ou connexion à un salon.

Avec l'état d'avancé du projet, la base de données intervient donc dans : la création de salon, la connexion à un salon, l'envoi de messages, les votes et l'affichage des anciens messages.

La base de données nous sert également à savoir quels salons existent, qui sont les utilisateurs s'étant connectés à chaque salon, quels messages ont été envoyés, par qui et dans quel salon et enfin, quel type de vote a reçu chaque message et par qui.

## Exportation des données du salon

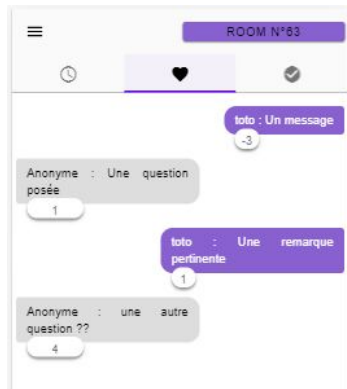
Lors de l'ouverture du volet gauche dans un salon, l'utilisateur pourra télécharger un fichier .csv contenant des informations sur le salon. Les informations sont principalement les messages postés, et les utilisateurs qui se connectent et



déconnectent ainsi que les insertions et les suppressions des tuples de la base de données.

## Les votes

Nous avons ajouté un tri en fonction du nombre de votes que reçoit un message, actuellement le tri se fait en ordre croissant. Une amélioration serait de se limiter au 15 messages les plus voté positivement par exemple pour limiter le coût et le temps d'affichage des messages lorsqu'il y en a beaucoup. En effet, tous les messages sont triés et affichés lors de



l'appuie sur l'onglet vote. Pour afficher les messages triés, nous récupérons les informations des tables vote et message. Il nous suffit ensuite de faire le lien entre les deux, de trier les messages selon le nombre de vote puis de les afficher.

Lorsque le client appui sur un bouton de vote cela lance l'événement 'click'. Mais il y a deux boutons de votes ainsi comme vous pouvez le voir ci-dessous. L'identifiant html d'un bouton de vote est de la forme : "UP\_X" ou "DOWN\_X" avec X l'identifiant du message dans la base de données. Il suffira donc côté serveur de récupérer "UP" ou "DOWN" ainsi que X pour trouver le message dans la base de données et savoir si le vote est positif ou négatif. Ci-dessous se trouve la fonction côté client qui permet de gérer les couleurs des boutons de vote.

```
$(document).on("click", ".vote", function() {
    socket.emit("votes", idIntoDB, this.id);
    var couleur = document.getElementById(this.id).style.color;

    //Restitution de l'identifiant du bouton opposé (- si on appuis sur +
    / + si on appuis sur -)
    var numMsg = this.id.split("_");
    var idBtnOpp;
    if (numMsg[0] == "UP") {
        idBtnOpp = "DOWN_"+numMsg[1];
    } else {
        idBtnOpp = "UP_"+numMsg[1];
    }

    //Changement de couleur
    if (couleur == "lightgray") { //Si le bouton est gris
        document.getElementById(this.id).style.color = "orchid"; //le bouton
        devient violet (orchid)
        document.getElementById(idBtnOpp).style.color = "lightgray"; //le
        bouton opposé devient gris
    }
});
```

```
    } else{  
        document.getElementById(this.id).style.color = "lightgray"; //sinon  
        Le bouton redevient gris  
    }  
});
```

Désormais la partie serveur, dans celle-ci on déclare une variable nommé 'vote', elle ne peut prendre que deux valeurs soit 1 soit -1. Si le bouton appuyé est négatif la valeur sera de -1 sinon elle sera de 1.

Ensuite on utilise une promesse elle nous permet d'être sûr que les requêtes de la base de données soit bien terminées avant de continuer l'exécution du code.

Dans notre version un utilisateur ne peut pas voter deux fois pour le même message ainsi nous vérifions à l'aide de notre base de donnée. Si la personne à déjà voté alors nous supprimons son précédent vote. Nous comparons son ancien vote avec le nouveau et si c'est le même nous ne faisons rien de plus, sinon nous ajoutons le nouveau vote. Ensuite, nous envoyons un événement aux clients pour que les votes s'actualisent.

## VII - Fonctionnement du responsive

L'implémentation du "responsive" dans notre projet se fait automatiquement grâce à la bibliothèque Material Design.

Cependant nous avons dû faire la page connexion en responsive par nous-mêmes car elle utilise une grille pour l'affichage des différents composants. Pour cela nous utilisons les media query, ainsi lorsque la page est réduite, l'affichage de la page est modifié pour correspondre à l'appareil utilisé.

```
.wrapper {
  display: grid;
  width: 45%;
  grid-template-columns: repeat(2, 1fr);
  grid-auto-rows: minmax(110px, auto);
  grid-template-areas: "a a"
                      "b c";
}
.one {
  grid-area: a;
}
.one p {
  font-size: 0.8em;
  user-select: none;
}
.two {
  grid-area: b;
}
.three {
  grid-area: c;
  margin-bottom: 3%;
}
```

```
@media screen and (max-width: 1000px) {
  #creer_room {
    height: 3em;
  }
  .wrapper {
    display: grid;
    width: 45%;
    grid-template-columns: repeat(1, 1fr);
    grid-auto-rows: minmax(80px, auto);
    grid-template-areas: "a"
                      "b"
                      "c";
  }
  .one {
    grid-area: a;
  }
  .two {
    grid-area: b;
  }
  .three {
    grid-area: c;
    margin-top: 5%;
  }
}
```



Desktop view of the login form. The form is centered and contains a text input for 'Nom\*' with a placeholder '\*par défaut anonyme'. Below it is a text input for 'Numéro du salon'. To the right of the 'Numéro du salon' input is a large purple button labeled 'NOUVEAU SALON'. Below the 'Numéro du salon' input is a purple button labeled 'REJOINDRE UN SALON'.



Mobile view of the login form. The form is centered and contains a text input for 'Nom\*' with a placeholder '\*par défaut anonyme'. Below it is a text input for 'Numéro du salon'. Below the 'Numéro du salon' input are two purple buttons: 'REJOINDRE UN SALON' and 'NOUVEAU SALON'.

## VIII - Évolutions possibles

**La création d'un statut "répondu" pour les messages :** Le créateur du salon ou l'utilisateur qui écrit un message doit pouvoir indiquer si sa question a été répondue. L'utilisateur pourra cliquer sur un bouton (ou cocher une case) sur son message pour lui donner le statut répondu. La table "Message" sera également actualisée et l'attribut answered du message sera modifié.

**Le tri des messages selon le critère "répondu" :** L'utilisateur doit avoir la possibilité de trier les messages selon si ces-derniers ont été répondu ou non. Nous utiliserons également la table "Message" et particulièrement les attributs qui correspondent au numéro du message et au statut répondu, afin d'afficher les messages par date d'envoi ou par statut répondu. Le statut répondu sera donné par le créateur du salon ou l'expéditeur du message en cliquant sur un bouton ou en cochant une case.

**Supprimer un salon :** Le créateur du salon doit avoir la possibilité de supprimer un salon existant. L'utilisateur aura accès à un bouton qui, après confirmation, déclenchera la suppression du salon et donc de tous les tuples associés de toutes les tables ainsi que la déconnexion des utilisateurs encore connectés. En cas d'inactivité prolongée, un salon s'autodétruit.

**Le système de commentaires :** Les utilisateurs doivent avoir la possibilité de répondre aux messages en rédigeant un commentaire. L'utilisateur pourra cliquer sur un message pour actualiser son interface et rédiger un commentaire. Pour lier un commentaire ou retrouver les commentaires liés à un message, nous utiliserons la table "Message" qui contient un attribut comment. Cet attribut vaut null si le message n'est pas un commentaire, sinon il prend l'identifiant du message auquel il est lié. Il suffira donc de parcourir la table pour retrouver les réponses d'un message.

**La suppression des messages :** Un message doit pouvoir être supprimé par le créateur du salon ou l'expéditeur. Pour supprimer un message l'utilisateur aura juste à cliquer sur un bouton et après confirmation le tuple de la table "Message" ainsi que les votes et les commentaires associés seront supprimés.

**L'accès à la liste de ses messages postés :** Un utilisateur doit pouvoir avoir accès à tous ses messages postés et ainsi vérifier facilement s'il a eu une réponse. Un onglet sera disponible sur l'interface de l'utilisateur dans lequel tous ses messages seront affichés. Il suffira de retrouver les messages de l'utilisateur dans la table "Message" (nom utilisateur, numéro du salon, commentaire ou non). Le fonctionnement sera similaire à la fonction de tri des messages.

**La création de statistiques pour le créateur du salon :** Le créateur du salon doit avoir accès à différentes statistiques. Le créateur sera le seul à avoir accès à une page où différentes statistiques seront affichées telles que le nombre de personnes dans le salon, le nombre de messages envoyés par rapport au temps, ... Nous pourrions utiliser ChartJs pour modéliser les statistiques calculées à partir des différentes tables.

**La création d'un grade modérateur :** Le créateur du salon doit pouvoir donner des droits supplémentaires à certains utilisateurs. Le rôle sera attribué via la table "AppUser".

Un utilisateur modérateur pourra : supprimer n'importe quel message ou marquer un message comme répondu (en plus des droits d'un utilisateur standard).