Frédéric YASSI
Pierre AUFFRAY

29 / 11 / 2021

Rapport Projet TWEB

Création d'un tchat client/serveur avec Node.js

ECOLE SUPERIEURE D'INGENIEURS DE RENNES

UNIVERSITÉ DE RENNES 1

Table des matières

Dans le cours de développement web, nous avons appris à manier l'outil JavaScript Node.js [1]. Avec cet outil, nous avons donc créé un projet qui consiste à réaliser un tchat client/serveur de messagerie instantanée regroupant plusieurs fonctionnalités qui nous a été demandé d'implémenter. Avant de se lancer dans les différentes étapes de la conception de ce projet, il est important de définir ce qu'est Node.js. Node.js est un environnement d'exécution JavaScript basé sur le moteur JavaScript V8 de Chrome. Il faut savoir que la plupart des navigateurs internet sont équipés d'un moteur JavaScript qui va permettre de traduire le code JavaScript qu'on produit en code machine. L'idée de la conception de Node.js a été de récupérer le moteur JavaScript V8, celui que l'on retrouve dans le navigateur Chrome, et de l'utiliser en dehors d'un navigateur. C'est ainsi qu'est né Node.js. Cet outil regroupe donc plusieurs fonctionnalités. Il permet notamment de créer et de faire le lien entre des clients et des serveurs web par le biais de différents protocoles. C'est cette fonctionnalité qui va particulièrement nous intéresser car, nous vous le rappelons, le but de notre projet est de créer une messagerie instantanée basée sur la structure client/serveur. Tout au long des différents TD (Travaux Dirigés) du cours de développement web, on a donc pris en main petit à petit cet environnement d'exécution JavaScript et ainsi compris comment fonctionnait Node.js. Avant de rentrer dans l'aspect technique du projet, on va voir sur quelles bases ce projet repose. Ensuite nous parlerons de la découverte de certaines fonctionnalités de Node.js et comment nous les avons utilisés et enfin nous entamerons l'aspect plus pratique en expliquant le cheminement du projet et les différentes étapes de conception en détaillant chaque point important sur lesquels nous avons rencontré des difficultés ou au contraire les points clés qui nous ont fait avancer de manière importante. Cette dernière partie se découpera d'ailleurs en plusieurs sous-parties.

Les bases du projet

Les sockets TCP

Comme évoqué précédemment, le projet se base sur la création de clients et serveurs web afin de communiquer entre eux et ainsi s'échanger des messages qui vont permettre d'établir des connexions entre différents clients et ainsi arriver à ce qui se rapproche le plus d'une messagerie instantanée. Pour ce faire, on a donc utilisé les sockets TCP (Transmission Control Protocol). Les sockets permettent d'établir une connexion TCP/IP entre deux

programmes. TCP désigne un protocole de transmission utilisé sur les réseaux IP. C'est un des principaux protocoles de la couche transport du modèle TCP/IP. Il est également important de comprendre qu'une connexion TCP/IP est identifiée par les 4 données suivantes :

- L'adresse IP du premier programme
- Le port du premier programme
- L'adresse IP du deuxième programme
- Le port du deuxième programme

On a donc utilisé les sockets que fournit Node.js et ainsi créé notre premier serveur TCP et notre premier client TCP (figure 1).

```
var net = require('net');
var client = new net.Socket();

client.connect(8080, '127.0.0.1', function() {
    console.log('Connected');
    process.stdout.write('>');
});

client.on('data', function(data) {
    console.log('Received: ' + data);
    process.stdout.write('>');
});

process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdout.write('>');
process.stdin.on('data', (text) => {
        switch(text){
            case 'hello\n':
                client.write(JSON.stringify(
                    {
                        "action":"client-hello"
                    }))
                break;
            }
});
```

```
var net = require('net');

var server = net.createServer((socket)=>{
    console.log('CONNECTED from ' + socket.remoteAddress + ':' + socket.remotePort);
    socket.on('data', function(data) {
            msg = JSON.parse(data);
            console.log('DATA received from ' + socket.remoteAddress + ':' + data);

            switch(msg.action){
                case 'client-hello':
                    socket.write(JSON.stringify(
                        {"sender-ip":socket.remoteAddress+':'+socket.remotePort,
                        "action":"server-hello",
                        "msg":"Hello"}));
                break;
                default:
                    console.log(msg);
            }
        });
    }).listen(8080, '127.0.0.1');
```

Figure 1 : Client et serveur de base utilisant la librairie **net** de Node.js

Comme on peut le remarquer sur la figure 1, notre serveur « écoute » ce qu'il se passe sur le localhost (interface réseau locale de l'ordinateur, port 8080 et adresse IP 127.0.0.1). Quant au client, celui-ci se connecte au localhost et envoie des requêtes au serveur qui les interprète suivant le type de requête envoyé et ce qu'elles contiennent. Le client interprètera à son tour ce que le serveur lui renvoie et ainsi de suite. C'est donc cette gestion des requêtes et de ce qu'elles contiennent qui va constituer le cœur du projet.

L'objet JSON

Pour que les clients et le serveur communiquent entre eux, on doit imposer un format de message spécifique qui va définir un standard de message pour ainsi faciliter les échanges et rendre les transmissions dynamiques. Pour ce projet, on a donc utilisé le format JSON (JavaScript Object Notation). C'est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Pour résumer, Le JSON est un format qui stocke des informations structurées et est principalement utilisé pour transférer des données entre un serveur et un client. Un objet JSON comporte deux éléments essentiels : les clés et les valeurs.

- Les clés doivent être des chaînes de caractères. Elles contiennent une séquence de caractères qui sont entourés de guillemets.
- Les valeurs sont un type de données JSON valide. Il peut se présenter sous la forme d'un tableau, d'un objet, d'une chaîne de caractères, d'un booléen, d'un nombre ou null.

Un objet JSON commence et se termine par des accolades {}. Il peut contenir deux ou plusieurs paires clé/valeur, séparées par une virgule. Chaque clé est suivie d'un deux-points pour la distinguer de la valeur. Voici un exemple d'objet JSON que pourrait envoyer un client à un serveur (figure 2) :

```
{
    "from": client,
    "action": 'client-hello',
    "message": "hello"
}
```

Figure 2 : Exemple objet JSON

Nous avons ici trois paires clé/valeur : from, action et message sont les clés. Client, client-hello et hello sont les valeurs. Un objet JSON étant un objet, on utilise des méthodes de l'interface JSON pour manipuler ces objets et ainsi les envoyer, les décortiquer. Ce sont les méthodes *JSON.stringify()* et *JSON.parse()*. *JSON.stringify()* convertit une valeur JavaScript en une chaîne JSON. *JSON.parse()* convertit une chaîne JSON en un objet JavaScript. Donc on remarque directement que l'une sert à envoyer et l'autre sert à interpréter.

On a donc désormais toutes les bases et les standards de développement qu'impose le projet. Nous allons maintenant le découvrir un peu plus en détail.

Une messagerie instantanée

Le déroulement du projet a été progressif. Avant de commencer à vouloir faire envoyer des messages entre deux clients directement et se lancer têtes baissées, on a d'abord pris étudier Node.js et ses possibilités. Ce qui veut dire, dans notre cas, qu'on s'est sérieusement penché sur les sockets TCP et leur fonctionnement dans un premier temps. C'est le sujet du premier TD du projet. Ce n'est qu'en deuxième partie de ce TD qu'on a implémenté nos premières requêtes d'un client vers un serveur et ainsi démarrer les messages. Cela fut une première expérience qui commence à poser les bases du projet global et qui nous a permis de nous orienter. Une fois le mécanisme compris et assimilé, on a enrichi le panel des requêtes que peut produire le client en incorporant au projet la notion de groupe et de ces nombreuses possibilités (privé/public, messages, invitations, bannissements etc.). Et en dernière partie de ce TD, on a commencé à introduire une base de données **sqlite** avec le module **sqlite3**. On a stocké toutes les infos importantes relatives au bon fonctionnement du projet. On a par la suite complété et amélioré au fil du développement du projet cette base de données et le code en général. Ensuite est venu la notion de sécurité. Créer une messagerie instantanée avec des requêtes simples et lisibles entre des clients et un serveur peut être une source de problème car ce qui se passe entre ces connexions n'est pas protégé. C'est ainsi qu'on a naturellement intégrer la notion de cryptage. On a crypté les données qui circulaient sur ce tchat par des protocoles qui seront expliqués plus en détail plus loin dans le rapport.

Les sockets TCP et les premières requêtes

Les sockets TCP

Comme expliqué un peu plus haut dans les bases du projet, on retrouve les sockets TCP. Pour démarrer, il fallait donc bien comprendre leur fonctionnement et la logique derrière. On a donc vu (figure 1) que le serveur écoute sur un port et une adresse IP. Il reçoit des messages via une socket qui est passée en paramètre du serveur. Celui-ci traite la requête suivant son type et ce qu'elle contient. Nous avons défini les caractéristiques d'une requête par son type et son

action. Son type est l'entête en quelque sorte de la transmission, son « event » pour être plus précis. Il correspond au premier argument de la méthode *socket.on()*. Ici, on identifie le type de la requête qui est 'data'. Son action est tout simplement la valeur de la clé action qui est transmise dans l'objet JSON d'une requête. Avec ces deux informations, on peut déterminer précisément quelle requête un client à envoyer et ainsi agir en conséquence. C'est cette méthode que nous avons décidé d'adopter. Donc pour orienter le type d'une requête, nous avons choisi d'utiliser la méthode *socket.emit()* du module net de Node.js. La méthode *socket.emit()* peut permettre de passé un « event » en premier argument qui correspondra tout au long du projet au type de requête envoyé.

Du côté client, on se connecte donc au port et à l'adresse IP choisie et on envoie des requêtes tapées en ligne de commande au serveur. Pour récupérer ces commandes, on a commencé par utiliser les entrées et sorties standards du processus, **stdin** et **stdout**. Ces requêtes ont bien-sûr un format et une typologie bien spécifique qui est nécessaire afin d'interpréter correctement ce qu'un utilisateur tape au clavier. Il ne faut pas oublier qu'un client attend également une réponse du serveur car le serveur répond aux demandes du client donc il lui envoie les informations que le client demande ou transmet les informations qu'il doit transmettre depuis un autre client. Voici ci-dessous un schéma récapitulatif du fonctionnement des sockets avec le module **net** (figure 3) :

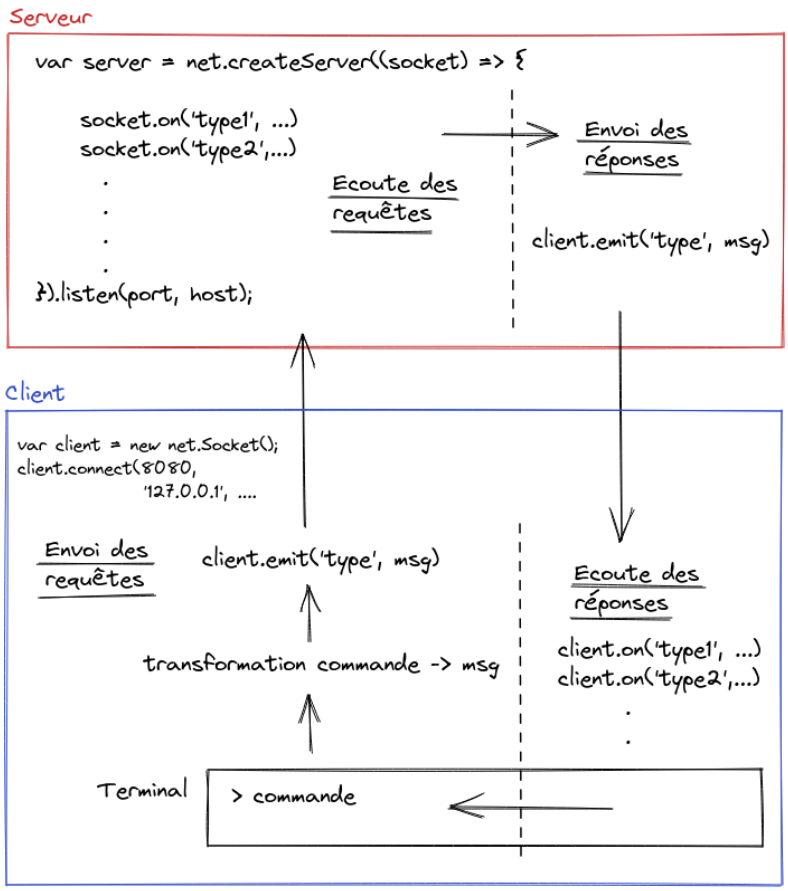

Figure 3 : Schéma représentatif de l'interaction entre un client et un serveur

Maintenant que le fonctionnement des envois et des réceptions est globalement compris, une question vient naturellement : qu'est-ce qu'on envoie ? C'est là qu'intervient le format des commandes à taper dans le terminal et le format des messages avec notamment l'action à associer à chaque commande valide tapée.

Les premières requêtes

Avant même de commencer à envoyer des messages, il serait tout de même intéressant de connaître son expéditeur. On a donc instauré le fait que lorsqu'on lance le script correspondant au client, il fallait donner un nom d'utilisateur en argument du programme après l'argument --name de la commande qui lance le programme dans le terminal. Pour le récupérer, on a utilisé la librairie **yargs**. On a tout ce qu'il nous faut. On peut donc élaborer nos premiers

messages et nos premières commandes associées. Voici un tableau récapitulatif des premières commandes et les messages associés qu'on a dû implémenter dans le code (figure 4) :

Commande	Message : JSON format - Client	Message : JSON format - Serveur	Signification
s;dest;msg	{"from":sender_name, "to": dest, "msg": msg, "action": 'client-send'}	{"from":sender_name, "sender-id": id, "msg": msg, "action": 'client-send'}	Envoyer un message privé à un autre utilisateur
b;msg	{"from": sender_name, "msg": msg, "action": 'client-broadcast'}	{"from":sender_name, "sender-id": id, "msg": msg, "action": 'client-broadcast'}	Envoyer un message à tout le monde
ls;	{"from": sender_name, "action": 'client-list-clients'}	{"from":sender_name, "sender-id": id, "msg": msg, "action": 'client-list-clients'}	Lister les utilisateurs
q;	{"from": sender_name, "action": 'client-lquit'}	{"from":sender_name, "sender-id": id, "msg": msg, "action": 'client-quit'}	Quitter

Figure 4 : Tableau des différentes commandes possibles et leur message JSON associé

Avec toutes ces informations établies, on peut désormais implémenter un client fonctionnel (figure 5) qui reflétera le squelette de tout le projet par la suite.

```
var net = require('net');
var process = require('process');
const yargs = require('yargs/yargs')
const { hideBin } = require('yargs/helpers')
const argv = yargs(hideBin(process.argv)).argv

var sender_name = argv.name;

var client = new net.Socket();

client.connect(8080, '127.0.0.1', function () {
    console.log('Connected');
    process.stdout.write('>');
});

client.on('message', function (data) {
    console.log("messages")
    console.log('Received: ' + data);
    process.stdout.write('>');
});

client.on('event', function (data) {
    console.log("event")
    console.log('Received: ' + data);
    process.stdout.write('>');
});

client.on('list', function (data) {
    console.log("list")
    console.log('Received: ' + data);
    process.stdout.write('>');
});
```

```
process.stdin.resume();
process.stdin.setEncoding('utf8');
process.stdout.write('>');
process.stdin.on('data', (text) => {
    text = text.split(';');
    switch (text[0]) {
        case 's':
            if (text.length == 3) {
                client.emit('message', JSON.stringify({
                    "from": sender_name,
                    "to": text[1],
                    "msg": text[2],
                    "action": "client-send" }))
            }
            break;
        case 'b':
            if (text.length == 2) {
                client.emit('message', JSON.stringify({
                    "from": sender_name,
                    "msg": text[2],
                    "action": "client-broadcast" }))
            }
            break;
        case 'ls':
            client.emit('list', JSON.stringify({
                "from": sender_name,
                "action": "client-list-clients" }))
            break;
        case 'q':
            client.emit('event', JSON.stringify({
                "from": sender_name,
                "action": "client-quit" }))
            break;
    }
});
```

Figure 5 : Code client et serveur pour des requêtes simples

Du côté du serveur, on fait un *socket.on()* pour chaque type construit et faire par exemple un switch case pour le type ‘message’ qui regroupe les messages privés et le broadcast. Et en fonction de la requête, le serveur redirige la réponse où il faut et avec les éléments qu’il faut. C’est ainsi que nous avons fondé notre projet en implémentant nos premières requêtes et en les traitant dans la partie serveur.

Pour un début, ce code est parfaitement fonctionnel mais il reste un détail à prendre en compte : le volume de requêtes possibles. Car pour un nombre assez faible de requêtes, il est tout à fait possible d’implémenter directement dans le code du client ou du serveur de simple commande et de les convertir en message JSON pour les envoyer. Mais si le nombre de requêtes possibles devient conséquent, cela peut devenir délicat. C’est pourquoi nous avons intégré des modules à notre projet.

Optimisation du code

Pour optimiser le code et le rendre plus lisible et surtout beaucoup plus pratique, on a intégré différents modules. Nous avons donc créé un module **Commande** qui traite les entrées du processus où l'utilisateur tape ses commandes et les envoie ensuite. Il y a également un module **FormatMessage** qui construit et retourne les messages suivant le statut de celui qui envoie (serveur ou client) et le type de message qu'il veut envoyer. Et enfin un module **SendMessage** qui récupère le message généré par le module précédent et l'envoie à ceux qui sont concernés. On peut donc visualiser l'imbrication de ces différents modules à travers ce schéma (figure 6) :

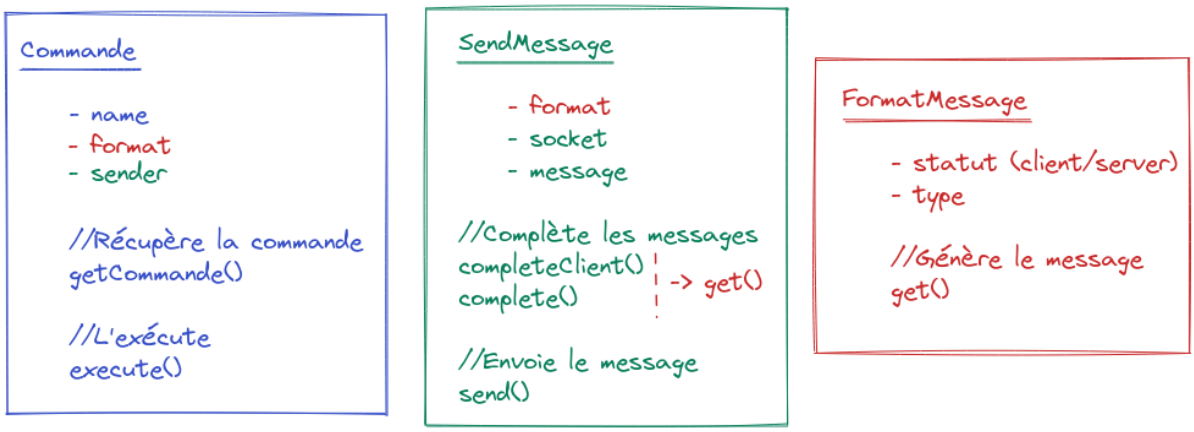

Figure 6 : Schéma des différents modules créés

En plus d'optimiser le code, on a également amélioré l'interface du client. Comme proposé dans les sujets de TD, on a choisi d'utiliser la librairies **Inquirer** [2]. Son fonctionnement est très simple. Il est à base de questions et de réponses sous différentes formes, sous différents types ce qui correspond bien aux possibilités du projet. Voici un exemple (figure 7) :

```
const inquirer = require('inquirer');
const output = [];

const questions = [
  {
    type: 'input',
    name: 'tvShow',
    message: "What's your favorite TV show?",
  },
  {
    type: 'confirm',
    name: 'askAgain',
    message: 'Want to enter another TV show favorite (just hit enter for YES)?',
    default: true,
  },
];

function ask() {
  inquirer.prompt(questions).then((answers) => {
    output.push(answers.tvShow);
    if (answers.askAgain) {
      ask();
    } else {
      console.log('Your favorite TV Shows:', output.join(', '));
    }
  });
}

ask();
```

Cet exemple montre les questions qu'on pose et la fonction qui les traite. Elles peuvent être de type « input » pour les saisies au clavier ou par exemple de type « confirm » qui est un choix binaire (oui ou non). Ici l'exemple est récursif et correspond tout à fait à nos besoins pour la messagerie instantanée.

Figure 7 : Exemple d'une fonctionnalité du module inquirer

Extension de l'application

On a désormais tout ce dont nous avons besoin pour développer notre application et ainsi accroître notre panel de requêtes. Comme dans la plupart des messageries instantanées, on retrouve la notion de groupe. Cette partie est donc dédiée à l'implémentation de la gestion des groupes. Voici l'ensemble des possibilités concernant les groupes (figure 8) :

Commande	Message : format JSON - Client	Message : format JSON - Serveur	Signification
cg;group_name	{"from": sender_name, "group": group_name, "public": state, "action": 'cgroupe'}	{"from": sender_name, "sender-id": id, "group": group_name, "action": 'cgroupe', "msg" : msg}	Création d'un groupe

j;group_name	{"from": sender_name, "group": group_name, “invited”: invited, "action": “join”}	{"from": sender_name, “sender-id”: id, "group": group_name, "action": 'cgroupe', “msg” : msg}	Rejoindre un groupe
bg;group_name;msg	{"from": sender_name, "group": group_name, “msg”: msg, "action": “gbroadcast”}	{"from": sender_name, “sender-id”: id, "group": group_name, "action": “gbroadcast”, “msg” : msg}	Envoyer un message à un groupe
members;group_name	{"from": sender_name, "group": group_name, "action": “members”}	{"from": sender_name, “sender-id”: id, "group": group_name, "action": “members”, “msg” : msg}	Lister les membres d’un groupe
messages;group_name	{"from": sender_name, "group": group_name, "action": “messages”}	{"from": sender_name, “sender-id”: id, "group": group_name, "action": “messages”, “msg” : msg}	Lister les messages d’un groupe
groups;	{"from": sender_name, "action": “groups”}	{"from": sender_name, “sender-id”: id, "action": 'groups', “msg” : msg}	Lister les groupes
leave;group_name	{"from": sender_name, "group": group_name, "action": “leave”}	{"from": sender_name, “sender-id”: id, "group": group_name, "action": 'leave', “msg” : msg}	Quitter un groupe
invite;group;dest	{"from": sender_name, "group": group, “dest”: dest, "action": “invite”}	{"from": sender_name, “sender-id”: id, "group": group, “dest”: dest, "action": 'invite', “msg” : msg}	Inviter quelqu’un dans un groupe

kick;group;dest;reason	{"from": sender_name, "group": group, “dest”: dest, “reason” : reason, "action": “kick”}	{"from": sender_name, “sender-id”: id, "group": group, “dest”: dest, "action": 'kick', “reason”: reason, “msg” : msg}	Exclure quelqu’un d’un groupe
ban;group;dest;reason	{"from": sender_name, "group": group, “dest”: dest, “reason” : reason, "action": “ban”}	{"from": sender_name, “sender-id”: id, "group": group, “dest”: dest, "action": 'ban', “reason”: reason, “msg” : msg}	Bannir quelqu’un d’un groupe
unban;group;dest	{"from": sender_name, "group": group, “dest”: dest, "action": “unban”}	{"from": sender_name, “sender-id”: id, "group": group, “dest”: dest, "action": 'unban'}	Ne plus bannir quelqu’un d’un groupe
states;group	{"from": sender_name, "group": group, "action": “states”}	{"from": sender_name, “sender-id”: id, "group": group, “msg”: msg, "action": “states”}	Lister les évènements d’un groupe

Figure 8 : Tableau des différentes commandes pour la gestion de groupe

Toutes ces nouvelles requêtes déterminent les évènements de groupe dans l’application. Pour traiter et même envoyer ces commandes, il est nécessaire de modifier le code client et serveur et également tous les modules créés. Cela implique donc de créer de nouveaux formats de messages dans le module FormatMessage du côté client et du côté serveur. De récupérer ces messages dans le module SendMessage sans oublier de compléter le module Commande avec la mise en place des nouvelles commandes à interpréter. Côté client et serveur, il faut désormais

développer des « écouteurs » pour chaque type de message. Rappelez-vous, la méthode *socket.on()* permet de traiter les requêtes dont le type de message correspond au premier argument de la fonction (voir figure 3). Il est temps de faire un point sur ces différents types.

Pour le moment, on a la gestion des messages privés, des messages broadcast, des groupes et le reste qui contient les connexions/déconnexions ainsi que le listage des groupes, des clients etc. On a donc défini quatre « écouteurs » :

- PrivateMessage : les messages privés
- BroadcastMessage : les messages broadcast
- GroupMessage : les évènements de groupe
- data : le reste des évènements cités juste au-dessus (list, groups etc.)

Ces quatre « écouteurs » vont donc traiter les messages qui leur sont destinés. Comme un écouteur regroupe potentiellement plusieurs actions comme GroupMessage, on effectue un switch case des actions et on agit en conséquence. C'est ainsi que nous avons implémenté l'ensemble des fonctionnalités de notre application. Il reste néanmoins un point extrèmement important qui n'a toujours pas été évoqué jusqu'à présent mais qui pourtant est d'une importance capitale : la gestion des données. En effet, développer les fonctionnalités d'une application est une chose, mais recueillir l'ensemble de ses données pour faire fonctionner ses rouages et ainsi obtenir une application fonctionnelle en est une autre. C'est pourquoi il est maintenant indispensable d'intégrer une base de données.

La base de données

Pour gérer toutes les données dont a besoin l'application pour son bon fonctionnement, on a utilisé le module **sqlite3**. En voici son schéma (figure 9) :

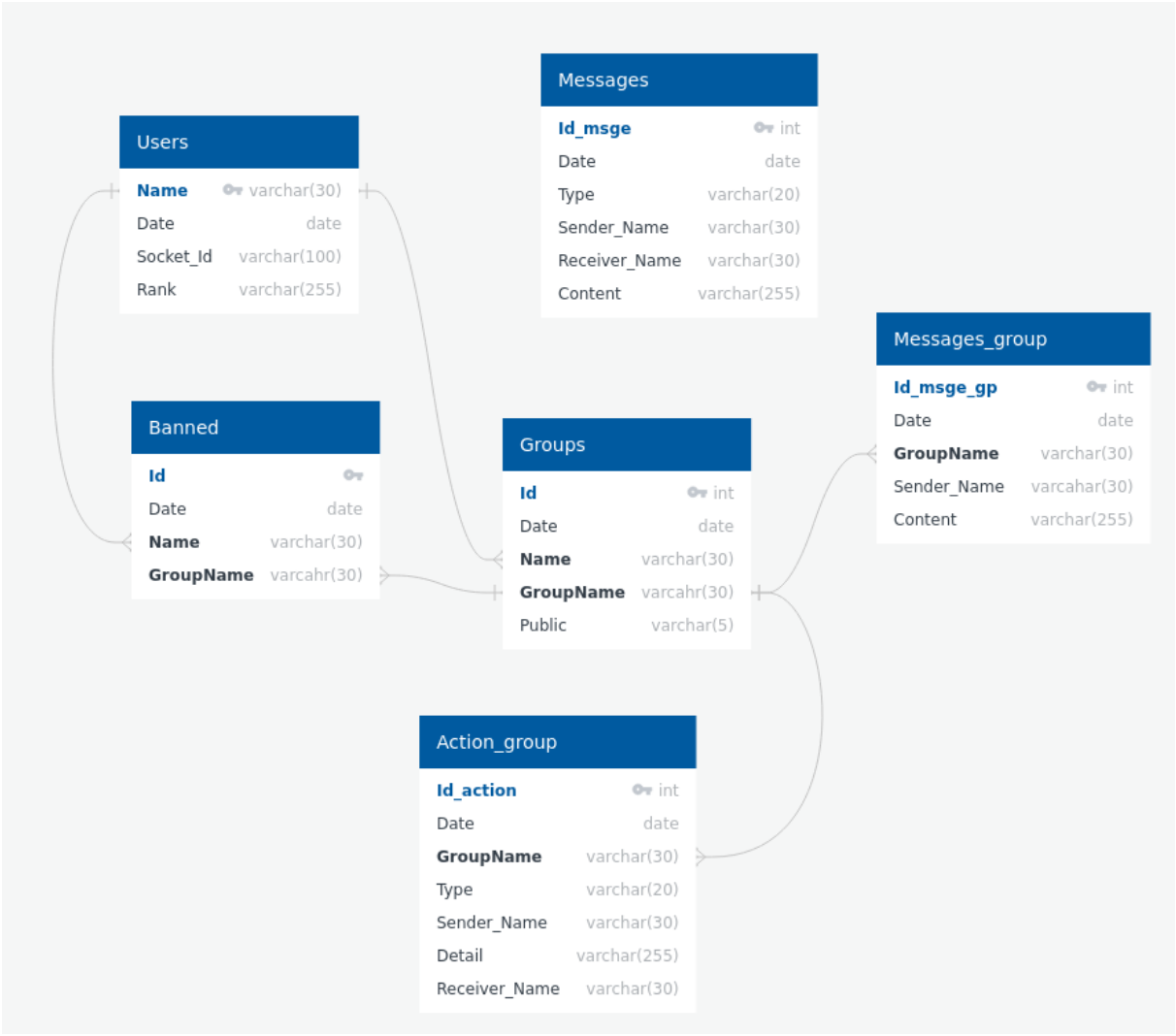

Figure 9 : Schéma de la base de données sqlite utilisée pour le projet

Grâce à cette architecture, on peut lier les différentes actions au groupe ou à l'utilisateur en question en inversement et ainsi aller chercher les données qui participent au bon fonctionnement de l'application. On peut d'ailleurs remarquer, en regardant un peu plus en détail le schéma, qu'on a ajouté quelques éléments à notre application comme la notion de groupes privés ou publics avec l'attribut Public. L'attribut Rank de la table Users fait référence au mot de passe de l'utilisateur. Pour des questions de sécurité, il est préférable de ne pas expliciter montrer où sont stockés les mots de passe des utilisateurs dans la base de données. Même si dans notre application, ce « camouflage » reste assez réduit. Pour ajouter de la sécurité

à notre application, on peut y ajouter des éléments qui la renforcent. On en vient donc à une autre partie essentielle dans l'échange de données : la cryptographie.

La cryptographie : toute une sécurité

Dans tout projet informatique, l'aspect de la sécurité est un facteur important notamment dans notre cas où il s'agit d'une application client/serveur de messagerie. Il est primordial pour nous de veiller à la confidentialité et l'intégrité des messages échangés et de la protection des données personnelles de chaque utilisateur.

Pour cela, on utilise la cryptographie à clé privée qui est celle qui convient pour sa rapidité et son efficacité. L'algorithme utilisé est le AES-256-cbc, c'est-à-dire que la clé fait 192 bits soit 32 caractères. Le principe de la cryptographie à clé privée est le suivant : les deux entités qui communiquent doivent avoir la même clé servant à chiffer et à déchiffrer. Cette clé privée est appelée « le secret ».

Un problème se pose alors : comment peut-on échanger le secret sans qu'il soit intercepté par un individu malveillant ? C'est en cela qu'intervient la cryptographie à clé publique à travers l'algorithme de Diffie-Hellman. Il permet de créer un secret commun entre deux entités tout en ignorant les clés privées. Comment est-ce possible ? Chaque entité génère une paire de clé, l'une publique permettant de chiffrer l'information et l'autre privée permettant de déchiffrer l'information. Les deux entités se partagent leurs clés privées et utilisent la clé publique de l'autre entité ainsi que leur clé privée pour générer un secret commun. Il est impossible de deviner ce secret commun en interceptant les clés publiques à travers le réseau. C'est ce qui caractérise même cet algorithme.

Une fois que chaque entité obtient le secret commun, tous les messages qui s'en suivent sont chiffrés. Quiconque ne détient pas le secret ne peut comprendre.

C'est cette même logique qui a été appliqué à notre projet tel qu'il est décrit dans le schéma suivant (figure 10) :

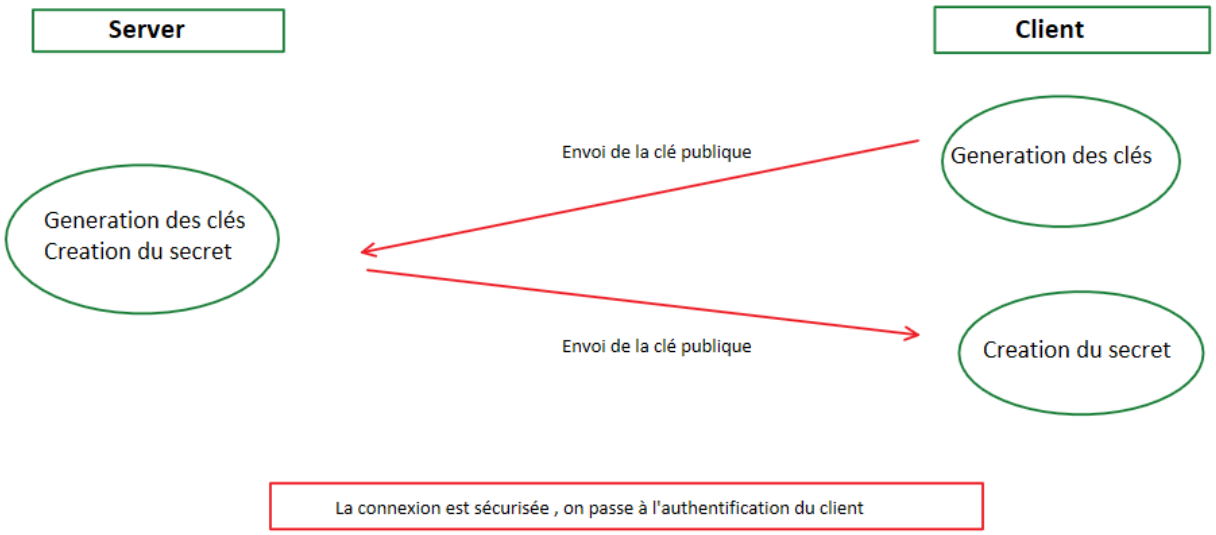

Figure 10 : Schéma du cryptage appliqué au projet

Aussi aucun utilisateur ne doit avoir accès à des informations qui ne sont pas les siennes. On utilise donc des mots de passe d'au moins 8 caractères et un nom d'utilisateur. Les mots de passe ne sont pas stockés en clair dans la base de données afin de rendre la base de données plus robuste en cas d'attaque. Pour le stockage des mots de passe, on utilise une fonction de hachage. Les fonctions de hachage sont des fonctions mathématiques très complexes et à un seul sens. C'est-à-dire pour un hash donné correspond une infinité de combinaisons correspondantes. Il n'est donc pas possible de récupérer la valeur même d'un mot de passe en récupérant son hash dans la base de données. Ainsi chaque ancien utilisateur est soumis à la phase d'authentification avant de se connecter. Une fois le mot de passe saisi, celui-ci traverse la fonction de hachage et est comparé au hash du mot de passe correcte correspondant au compte dont le nom a été donné. S'il y a une correspondance, l'authentification est validée, sinon elle est refusée. Le schéma suivant (figure 11) explique ce cheminement.

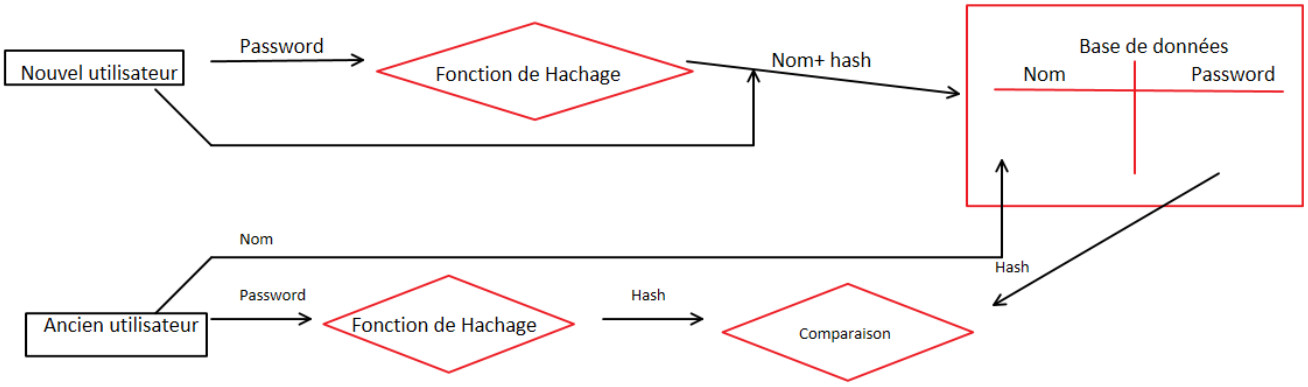

Figure 11 : Schéma sur le fonctionnement de l'authentification avec la technique du hachage

C'est donc en réunissant ces deux techniques de cryptage que nous avons protéger notre base de données et ainsi assurer à nos utilisateurs une certaine confidentialité quant aux messages et aux requêtes qu'ils envoient sur cette application.

Depuis le début, on utilise le module net de Node.js mais celui-ci ne permet pas d'étendre notre application vers d'autres protocoles. C'est pourquoi nous allons terminer ce rapport en regardant comment reconstruire toute notre application avec le protocole websocket.

Utiliser les websocket

Dans cette dernière partie de projet, le but était d'intégrer la librairie **socket.io** [3] avec Node.js. Cette manipulation permet d'étendre les fonctionnalités de notre application et de la rendre potentiellement utilisable par la suite dans un navigateur web par la suite. Ce n'est pas notre objectif pour ce projet mais c'est une des raisons qui justifie le changement de structure. Bien que socket.io et ses fonctionnalités en certains points différents de la structure actuelle du projet, il est tout à fait possible de conserver cette idée d'écouteurs qu'on utilise depuis le début. Donc le passage vers le module socket.io n'a pas affecté notre squelette de base et on a juste fait appel à la librairie par les lignes de codes suivantes dans chacun des codes client et serveur (figure 12 et 13).

```
const ioClient = require('socket.io-client').connect("http://localhost:8080");
```

Figure 12 : Client socket.io connecté ici au localhost

```
const server = require("socket.io")(8080, "127.0.0.1");
```

Figure 13 : Serveur socket.io connecté ici au port 8080 et à l'adresse IP 127.0.0.1 (localhost)

Pour ces exemples, le serveur peut très bien être totalement différent. Le port et l'adresse IP peut d'ailleurs être récupérés en argument de la ligne de commande qui lance le script. Quant au client, il est connecté ici au localhost mais on peut très bien changer son url de connexion.

Conclusion

Ce projet a été une véritable découverte du développement web et du langage JavaScript. A travers une idée de messagerie instantanée, on a pu découvrir les rouages de Node.js, la praticité du format JSON ou encore la force des protocoles de cryptage. Développer ce projet nous a permis de se rendre compte des possibilités qu'offre Node.js et ses multitudes de module. En partant d'un simple code de base, on a pu implémenter petit à petit les différentes commandes qui composent une messagerie instantanée et ainsi arriver à un résultat plutôt complet. On a d'abord vu ce qu'était le module net, puis on a commencé nos premières requêtes. Ensuite on a étendu notre application vers de la gestion de groupe. Cette partie posait des problèmes de gestion de données ce qui nous a forcer à concevoir une base de données. Cela nécessite donc une certaine réflexion quant à sa structure et à son optimisation pour arriver à récupérer les données qui nous intéressent. Une fois la base de données implémentée, il a fallu se pencher sur la thématique de la sécurité car c'est important de crypter les données envoyées afin de garder une certaine confidentialité et surtout protéger les utilisateurs contre de potentielles attaques extérieures. Là encore on a dû se renseigner sur les protocoles à utiliser et leur fonctionnement. C'est très intéressant de savoir comment deux utilisateurs peuvent communiquer sans que personne puisse comprendre le moindre mot de ce qui est échangé. Une ultime partie aurait été de tester notre application avec des tests automatisés qui vérifient le bon fonctionnement de chaque requête afin d'être sûr du résultat.

Bibliographie

[1] «Node.js,» [En ligne]. Available: https://nodejs.org/docs/latest-v15.x/api/.

[2] SBoudrias, «Inquirer,» [En ligne]. Available: https://github.com/SBoudrias/Inquirer.js.

[3] «Socket.io,» [En ligne]. Available: https://socket.io.