01/10/2015

Super tuto ➔ Blog.bibi.io

Bonjour !
Je démarre ici un blog sur le développement, j'espère que les quelques articles et tutoriaux que j'y proposerai vous serons utiles !
J'aborderai divers sujets, du backend (NodeJS, Java EE...) mais aussi du front (CSS, SASS, Angular...) et peut-être quelques articles un peu plus portés "essais" si l'inspiration me vient.

Je commence donc ici avec un article sur le protocole WebSocket et la bibliothèque socket.io.

De quoi ça parle ?

Je vais commencer par expliquer ce qu'est le protocole WebSocket avant de présenter socket.io. Dans un deuxième article nous verrons comment construire de façon simple une application basée sur socket.io.

Dis tonton, c'est quoi WebSocket ?

Et bien avant tout, regardons ce que Wikipédia en dit :

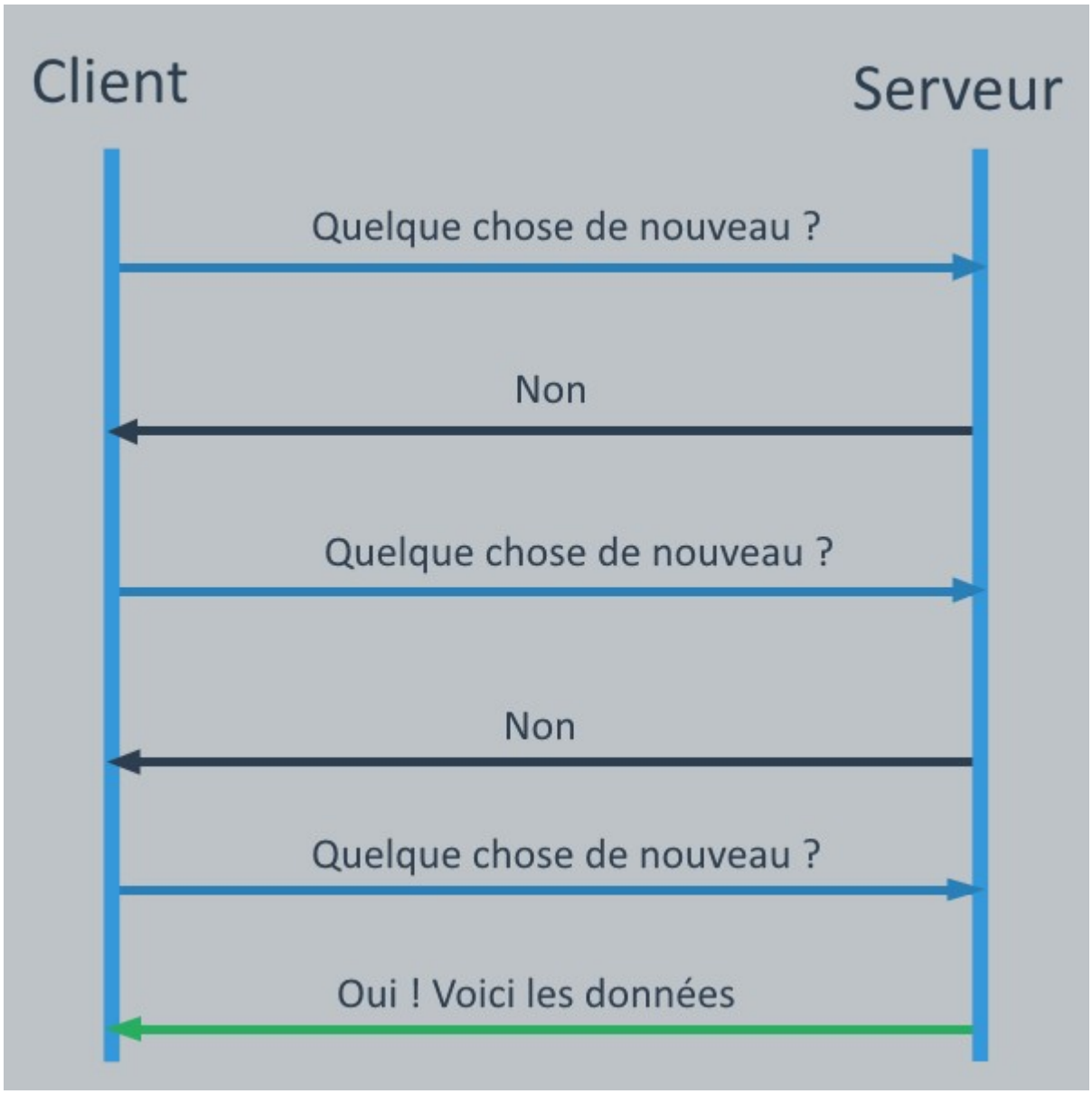
> *Le protocole WebSocket vise à développer un canal de communication full-duplex sur un socket TCP pour les navigateurs et les serveurs web.*

Outch. Dur.
Voyons ça plus simplement.

En une phrase simple : les WebSockets permettent de créer des applications temps-réel sur le web.

Le web classique

Classiquement, un client envoie une requête à un serveur qui lui répond en lui envoyant les données demandées (ou une erreur). Ce fonctionnement a évolué avec l'apparition d'Ajax, qui a permis de rendre ce processus de communication asynchrone (le client envoie une requête au serveur, qui répond en renvoyant uniquement les données nécessaires et sans avoir besoin de recharger sa page).
On voit bien qu'avec ce fonctionnement le serveur a besoin de recevoir une requête du client afin de lui envoyer des données.
Ça limite pas mal les choses si l'on souhaite faire des applications en temps réel ! Imaginez cela pour une application de chat : afin de savoir si de nouveaux messages sont arrivés, le client est obligé d'envoyer une requête de façon périodique au serveur pour savoir si de nouveaux messages sont arrivés. Ou alors il faut que le client actualise la page ou clique sur un bouton pour récupérer les éventuels nouveaux messages.



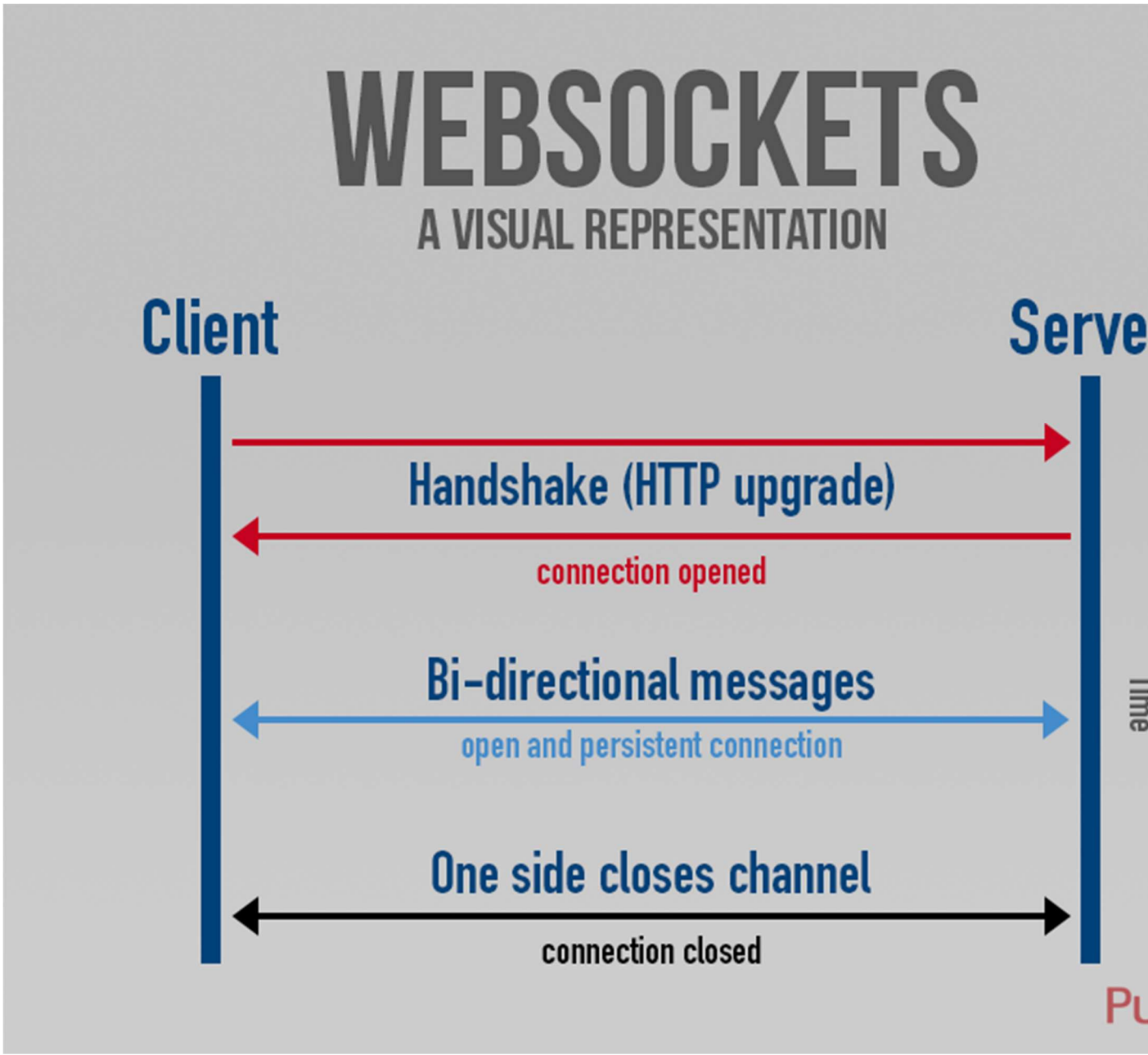
Ce n'est pas du *vrai* push. Ce n'est pas du *vrai* temps réel. Pas génial du tout donc.

WebSocket

C'est là qu'arrive le protocole WebSocket. Ce protocole autorise une communication bidirectionnelle entre le client et le serveur. En clair, le serveur peut envoyer directement des données au client sans que celui-ci n'ait effectuer de requête (et vice-versa, ça marche toujours dans l'autre sens). Les limites d'Ajax ne sont donc plus d'actualité.
Plutot cool, non ? Oui, plutôt cool.

Comment ça marche ?

Je ne vais pas rentrer dans les détails techniques, mais en bref, le client envoie un "Handshake" au serveur pour notifier son désir d'ouvrir une connection WebSocket avec lui. Ce "Handshake" est en fait une requête HTTP de type UPGRADE. Si le serveur l'accepte une connection est alors ouverte entre eux. Cette connection est persistante et basée sur le protocole TCP. Elle autorise la transmission de messages bi-directionnels (client vers serveur et vice-versa) et reste ouverte jusqu'à ce qu'un des protagoniste décide de la clore. En schéma ça donne ça (source de l'image).



Ca marche partout ?

Non, malheureusement pas encore. Ce protocole est supporté par les navigateurs suivants :

- Chrome 16+
- Firefox 11+
- Internet Explorer 10+
- Opera 12.10+
- Safari 6+

Socket.io

Socket.io est une librairie Javascript qui permet d'effectuer non seulement des communications asynchrones bidirectionnelles entre client et serveur (comme prévu par le protocol WebSocket) mais également bien plus !

Socket.io n'est **pas** une librairie uniquement basée sur WebSocket, ce qui a des avantages et des inconvénients.
Pour être clair : si votre navigateur supporte WebSocket, socket.io utilisera WebSocket pour communiquer. Sinon, il cherchera d'autres moyens de le faire (sans rentrer dans les détails : Flash, Ajax Long Polling...). Cela permet d'augmenter considérablement les navigateurs compatibles avec socket.io.
Mais comme je le disais juste avant, socket.io ajoute également plsueurs fonctionnalités non prévues par le protocole (broadcast de message par exemple — on verra ce que c'est dans le prochain article), ce qui a comme inconvénient de perdre la conformité avec le protocole de base et donc l'interopérabilité avec les autres librairies qui peuvent exister. Ainsi, si on développe une application avec socket.io le client **et** le serveur doivent utiliser socket.io. Côté client, la librairie consiste à un simple fichier Javascript à inclure, côté serveur c'est une librairie pour Node.js.

Pour suivre un tuto CHAT en trois parties – sources téléchargées

Développer un chat en temps réel avec Socket.io – Partie 1

Après avoir introduit les WebSocket et Socket.io dans un précédent billet, nous allons cette fois voir comment développer une application basique mais fonctionnelle avec cet outil. Comme expliqué dans la première partie, Socket.io permet de développer des applications temps-réel. L'application temps-réel la plus simple à développer pour se faire la main semble évidente : un chat.

Trève de bavardages, au boulot.

Sources

Vous pouvez retrouver les sources de ce tutoriel sur GitHub. Il est possible de les télécharger directement depuis cette page ou de cloner la branche Git correspondant à cette partie du tutoriel.

```
git clone --branch part-1 https://github.com/BenjaminBini/socket.io-chat.git
```

Prérequis

Je supposerai dans ce tutoriel que vous avez Node.js et npm installés sur votre machine et que vous savez un minimum les utiliser.
Ce billet est une adaptation de la documentation officiel de Socket.io.

Initialisation du projet

Comme tout bon projet basé sur Node.js, nous allons créer un fichier `package.json` avec les informations de base sur notre programme.

```
{
    "name": "my-first-chat",
    "version": "0.0.1",
```

```
    "description": "my first socket.io app",
    "dependencies": {}
}
```

Afin de nous faciliter la tâche nous allons utiliser le framework Express. Si vous ne le connaissez pas, sachez qu'Express est un mini-framework web permettant de développer beaucoup plus rapidement certaines tâches fastidieuses avec Node.js nu (la gestion des routes, entre autres).

On va donc l'installer et l'ajouter aux dépendances définies dans notre `package.json`.

```
npm install --save express@4.10.2
```

Passons à l'initialisation du front-end !

Initialisation du client

La partie cliente va être placée dans un dossier `/public`, où l'on va mettre l'ensemble des fichiers accessibles à l'utilisateur (en gros l'html, le css et le js).

Créez donc un répertoire `/public` à la racine de votre projet et mettez-y trois fichiers :

- index.html
- client.js
- style.css

L'HTML va contenir simplement la structure de notre application avec une liste prête à accueillir nos messages ainsi qu'un champ de saisie. On en profite également pour importer notre fichier de style (`style.css`), JQuery et notre fichier js (`client.js`).

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="style.css" />
    <title>Socket.IO chat</title>
```

```
    <style>
    </style>
  </head>
  <body>
    <section id="chat">
      <ul id="messages"></ul>
      <form action="">
        <input id="m" autocomplete="off" /><button>Send</button>
      </form>
    </section>
    <script src="http://code.jquery.com/jquery-1.11.1.js"></script>
    <script src="client.js"></script>
  </body>
</html>
```

Un peu de CSS pour rendre le tout moins moche.

```
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}
body {
    font: 13px Helvetica, Arial;
    transition: all 0.5s;
}

section#chat form {
    background: #000;
    padding: 3px;
    position: fixed;
    bottom: 0;
    width: 100%;
    height: 50px;
}
section#chat form input {
```

```
    border: 0;
    padding: 5px 10px;
    width: 90%;
    height: 100%;
    margin-right: .5%;
    font-size: 20px;
}
section#chat form button {
    width: 9%;
    height: 100%;
    background: #e67e22;
    float: right;
    border: none;
    margin-right: 0.5%;
    font-size: 17px;
    color: white;
}
section#chat #messages {
    list-style-type: none;
    margin: 0;
    padding: 0;
    font-size: 15px;
}
section#chat #messages li {
    padding: 5px 10px;
}
section#chat #messages li:nth-child(odd) {
    background: #eee;
}
```

Pour l'instant on laisse `client.js` vide, ce sera pour la suite.

Initialisation du serveur

On doit créer le fichier principal de notre application, `server.js`.
Ca va rester très basique, on va écouter sur un port (3000) et renvoyer le contenu du dossier `/public` aux utilisateurs.

```
// Tout d'abbord on initialise notre application avec le framework Express
// et la bibliothèque http integrée à node.
var express = require('express');
var app = express();
var http = require('http').Server(app);

// On gère les requêtes HTTP des utilisateurs en leur renvoyant les fichiers du dossier 'public'
app.use("/", express.static(__dirname + "/public"));

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

Exécutez `node server` depuis le dossier où se trouve votre application pour la démarrer.
Si tout s'est bien passé, vous devriez voir le message `Server is listening on *:3000` s'afficher. Par ailleurs si vous vous rendez à l'adresse `http://localhost:3000`, vous devriez voir votre fenêtre de chat avec une zone de saisie en bas.

Buenissimo, mais jusqu'à maintenant on n'a pas encore vu de Socket.io là-dedans ! Alors allons-y.

Intégration de Socket.io

Comme notre application a deux parties (client et serveur), de même, Socket.io se présente en deux parties :

- La partie serveur, qui vient s'intégrer comme module à l'application Node.js
- La partie client, qui est une bibliothèque Javascript à intégrer à notre client web

Ajoutons donc Socket.io à notre projet.

```
npm install --save socket.io
```

Plus qu'à l'initialiser dans `server.js`. Modifiez les premières lignes de votre fichier pour qu'elles ressemblent à cela :

```
var express = require('express');
var app = express();
var http = require('http').Server(app);
var io = require('socket.io')(http);
```

On a ici simplement créé la variable `io`, qui va nous permettre de travailler avec Socket.io !

Côté client rien de plus simple : la bibliothèque est automatiquement mise à disposition à l'URL `/socket.io/socket.io.js`. Ajoutez donc la ligne suivante juste avant `<script src="client.js"></script>` dans votre fichier `index.html`.

```
<script src="/socket.io/socket.io.js">
```

Et initialisions le tout dans notre fichier `client.js` :

```
var socket = io();
```

Pas besoin de spécifier l'URL à laquelle le client Socket.io doit se connecter, par défaut il va tenter de se connecter sur le serveur qui héberge la page cliente.

L'événement 'connection'

Pour vérifier que tout fonctionne, nous allons utiliser un premier événement spécifique à Socket.io, l'événement `connection`. A chaque fois qu'un utilisateur se connecte sur la page, l'événement est déclenché. Nous allons donc l'écouter

et afficher un message dans la console à chaque déclenchement. Ajoutez ce code à `server.js`.

```
io.on('connection', function(socket){
  console.log('a user connected');
});
```

Redémarrez l'application (`node server`) et rendez-vous sur votre page (`http://localhost:3000/`).
Regardez la console :

```
Server is listening on *:3000
a user connected
```

Ouvrez d'autres onglets (ou actualisez la page), le message réapparaît à chaque fois. Parfait !

L'événement 'disconnect'

On peut également écouter l'événement `disconnect` qui est déclenché à chaque fois qu'un utilisateur se déconnecte de la socket sur laquelle il était connecté. Logiquement, cet événement est rattaché à la socket d'un utilisateur en particulier et non au module Socket.io en général.
C'est donc sur l'instance de l'objet `socket` (passé en argument à la fonction de callback de l'événement `connection`) qu'il faut écouter.

```
io.on('connection', function(socket){
  console.log('a user connected');
  socket.on('disconnect', function(){
    console.log('user disconnected');
  });
});
```

Relançez l'application, ouvrez un onglet, refermez-le, la console affiche bien `user disconnected` à chaque déconnexion (fermeture du navigateur, de l'onglet, actualisation de la page).

Emettre des événements

Bien, on voit donc que le serveur reçoit des événements du client en temps réel, par exemple la connexion et la déconnexion d'un utilisateur. Maintenant, le principe de Socket.io c'est que le client **et** le serveur peuvent émettre et recevoir des événements en temps réel et ainsi réagir en conséquence. De plus, il est possible de transmettre des données avec ces événements ! La plupart du temps il s'agira d'objets encodés en JSON mais des données binaires sont aussi une option.

Quel est l'événement le plus important à émettre dans une application de chat ? L'envoi d'un message évidemment.
Il nous faut donc émettre un événement quand l'utilisateur va valider le formulaire (c'est à dire cliquer sur "Send"). Il faut également envoyer le texte du message. Cela se fait très simplement ! Modifions `client.js`.

```
var socket = io();


$('form').submit(function(e) {
    e.preventDefault(); // On évite le recharchement de la page lors de la validation du formulaire
    // On crée notre objet JSON correspondant à notre message
    var message = {
        text : $('#m').val()
    }
    socket.emit('chat-message', message); // On émet l'événement avec le message associé
    $('#m').val(''); // On vide le champ texte
    if (message.text.trim().length !== 0) { // Gestion message vide
      socket.emit('chat-message', message);
    }
    $('#chat input').focus(); // Focus sur le champ du message
});
```

Maintenant que l'événement est émis, il faut le réceptionner côté serveur. Pour cela nous allons modifier `server.js`.

```
[...]
io.on('connection', function (socket) {

  /**
   * Log de connexion et de déconnexion des utilisateurs
   */
  console.log('a user connected');
  socket.on('disconnect', function () {
    console.log('user disconected');
  });

  /**
   * Réception de l'événement 'chat-message' et réémission vers tous les utilisateurs
   */
  socket.on('chat-message', function (message) {
    console.log('message : ' + message.text);
  });
});
[...]
```

Relancez votre application, rendez-vous sur votre page et envoyez un message. Regardez la console, les messages devraient s'afficher.

```
node server
Server is listening on *:3000
a user connected
message : Hello !
message : How are you ?
message : Socket.io is great !
```

Voilà un bon début. Ce n'est bien sûr pas terminé, il faut maintenant gérer l'affichage des messages chez les utilisateurs.

Les événements envoyés par le serveur

Pour l'instant, nos événements ont été envoyés d'un client vers le serveur. L'inverse est bien sûr tout aussi important. Il existe trois types d'événements envoyés par le serveur :

- l'événement simple, envoyé au travers d'une seule socket (vers un seul utilisateur)
    - Exemple :

```
socket.emit('random-event', randomContent);
```

- le broadcast, envoyé à tout le monde **sauf** à la socket courante (c'est à dire sauf à l'utilisateur courant)
    - Exemple :

```
socket.broadcast.emit('random-event', randomContent);
```

- la combinaison des deux : l'émission d'un événement à tous les clients connectés au serveur
    - Exemple :

```
io.emit('random-event', randomContent);
```

Ici, nous devons transmettre les messages à tous les utilisateurs. On va donc modifier `server.js` comme suit.

```
[...]
  /**
   * Réception de l'événement 'chat-message' et réémission vers tous les utilisateurs
   */
  socket.on('chat-message', function (message) {
    io.emit('chat-message', message);
  });
```

```
[...]
```

Comprenez bien ce qu'il se passe :

- un utilisateur accède à la page et se connecte au server via socket.io
- l'utilisateur envoie un message et émet donc un événement `chat-message`
- le serveur reçoit l'événement, ce qui déclenche l'émission d'un **autre** événement (aussi appelé `chat-message`, mais cela aurait pu être ce que l'on veut) qui lui, sera envoyé à **tous** les utilisateurs connectés

Il faut donc réceptionner l'événement côté client et afficher le message. Très simple, ajoutez ces quelques lignes à `client.js`.

```
/**
 * Réception d'un message
 */
socket.on('chat-message', function (message) {
  $('#messages').append($('<li>').text(message.text));
});
```

Ouvrez deux fenêtres côtes à côtes, tapez un message dans l'une, validez... il s'affiche dans l'autre. Et vice-versa ! Le chat est fonctionnel.

Source de l'animation : SOCKET.IO (MIT)

Sources

Vous pouvez retrouver les sources de ce tutoriel sur GitHub. Il est possible de les télécharger directement depuis cette page ou de cloner la branche Git correspondant à cette partie du tutoriel.

```
git clone --branch part-1 https://github.com/BenjaminBini/socket.io-chat.git
```

Conclusion

Je ne sais pas ce que vous en pensez, mais la première fois que j'ai découvert ces WebSocket et socket.io, je ne m'attendais pas à faire quelque chose comme cela en si peu de lignes de codes et si facilement ! Là est toute la puissance de Socket.io, une programmation événementielle basée sur la transmission de données en temps réel entre un serveur et des clients.

La suite

C'est un bon début, mais on a de nombreuses limitations avec cette version ! Pas de pseudos, pas d'identification, pas de liste des utilisateurs connectés... Il faut désormais aller plus loin en ajoutant des fonctionnalités. Celles proposées par le tutoriel officiel sont :

- Afficher un message lors de la connexion et la déconnexion des utilisateurs
- Ajouter le support des noms d'utilisateurs
- Affichage de la liste des utilisateurs connectés
- Affichage d'un message "xxx is typing" quand un utilisateur est en train d'écrire
- Gestion d'un historique de messages (affichage des derniers messages envoyés avant la connexion de l'utilisateur)
- ...

Dans les deux prochains billets, nous nous occuperons d'ajouter plusieurs de ces fonctionnalités !

Cliquez ici pour accéder à la partie 2

Développer un chat en temps réel avec Socket.io – Partie 2

Pour rappel, dans la première partie nous avons développé un chat temps-réel avec Socket.io permettant d'envoyer des messages de façon synchronisée entre différents utilisateurs. Mais il nous restait de nombreuses fonctionnalités à implémenter :

- Afficher un message lors de la connexion et la déconnexion des utilisateurs
- Ajouter le support des noms d'utilisateurs
- Affichage de la liste des utilisateurs connectés
- Affichage d'un message "xxx is typing" quand un utilisateur est en train d'écrire
- Gestion d'un historique de messages (affichage des derniers messages envoyés avant la connexion de l'utilisateur)
- ...

Aujourd'hui, nous allons nous intéresser aux deux fonctionnalités suivantes :

- Ajouter le support des noms d'utilisateurs
- Afficher un message lors de la connexion et la déconnexion des utilisateurs

Base de code

Nous repartons du code de la partie 1 pour développer cette deuxième partie. Si vous n'avez pas suivi la première partie de ce tutoriel, je vous invite soit à la suivre soit à télécharger son code source sur mon compte GitHub ou de cloner la branche git correspondante :

```
git clone --branch part-1 https://github.com/BenjaminBini/socket.io-chat.git
```

Bien, commençons !

Support des noms d'utilisateurs

Un chat n'a que peu d'intérêt si on ne peut pas identifier les différents émetteurs des messages. On va donc intégrer le support des noms d'utilisateurs à notre chat. L'utilisateur devra, s'il souhaite participer à la conversation, indiquer un pseudonyme.

Formulaire de connexion

On va donc ajouter un formulaire à notre page HTML demandant à l'utilisateur son nom d'utilisateur. Lorsque l'utilisateur cliquera sur 'Login', s'il a saisi un nom d'utilisateur on lui donnera accès au chat. De plus lorsqu'il enverra un message, son message sera précédé de son nom.

Tout d'abord ajoute à notre `index.html` le formulaire de connexion.

```
<!doctype html>
<html>
  <head>
    <link rel="stylesheet" href="style.css" />
    <title>Socket.IO chat</title>
    <style>
    </style>
  </head>
  <body id="logged-out">
    <section id="chat">
      <ul id="messages"></ul>
      <form action="">
        <input id="m" autocomplete="off"  /><button>Send</button>
      </form>
    </section>
    <section id="login">
```

```
      <form action="">
        <label for="u">Username</label>
        <input id="u" autocomplete="off" autofocus />
        <p>
          <button>Login</button>
        </p>
      </form>
    </section>
    <script src="http://code.jquery.com/jquery-1.11.1.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="client.js"></script>
  </body>
</html>
```

On a ajouté la section `login` et on a donné au `body` l'id `logged-out` afin de différencier l'affichage entre le mode connecté et le mode déconnecté.
En mode déconnecté, nous souhaitons afficher le formulaire par-dessus le chat et *flouter* le chat en CSS.
Pour cela, ajoutons ce code à `style.css`.

```
body#logged-out {
    background: rgb(223, 223, 223);
}
body#logged-out section#chat {
    filter: blur(5px);
    -webkit-filter: blur(5px);
}
body#logged-out section#login {
    opacity: 1;
}

section#login {
    transition: all 0.5s;
    opacity: 0;
    top: 45%;
    text-align: center;
```

```
    position: absolute;
    width: 100%
}
section#login label[for="u"] {
    display: block;
    font-size: 24px;
    margin-bottom: 10px;
}
section#login input#u {
    font-size: 25px;
    text-align: center;
    padding: 5px;
    border: 5px solid rgb(158, 158, 158);
}
section#login input#u:focus {
    outline: none;
}
section#login button {
    background: #e67e22;
    border: none;
    padding: 5px 80px;
    color: white;
    font-size: 20px;
    margin-top: 20px;
    cursor: pointer;
}
```

Reste donc à modifier l'id `logged-out` du `body` en Javascript. Lors de la soumission du formulaire, on va vérifier que le nom d'utilisateur n'est pas vide et si c'est le cas, retirer l'id du `body`. Par ailleurs il serait très utile à ce moment de prévenir le serveur que l'utilisateur s'est connecté et de lui fournir le nom de cet utilisateur ! On va donc émettre un événement qui sera géré par la suite par le serveur.

Rien de très sorcier ! Ajoutez ce code à `client.js` :

```
/**
 * Connexion d'un utilisateur
 */
$('#login form').submit(function (e) {
  e.preventDefault();
  var user = {
    username : $('#login input').val().trim()
  };
  if (user.username.length > 0) { // Si le champ de connexion n'est pas vide
    socket.emit('user-login', user);
    $('body').removeAttr('id'); // Cache formulaire de connexion
    $('#chat input').focus(); // Focus sur le champ du message
  }
});
```

Plutôt que d'envoyer simplement une chaîne de caractères contenant le nom de l'utilisateur, on l'encapsule dans un objet. En effet, si on souhaite plus tard ajouter d'autres informations sur l'utilisateur il sera plus simple d'avoir déjà un objet `user` de prêt.

Lancez votre application avec un petit coup de `node server` et testez-la (http://localhost:3000). Le formulaire de connexion s'affiche, entrez un nom d'utilisateur et validez : vous vous retrouvez bien sur le chat. Reste à gérer l'événement `user-login` côté utilisateur afin de conserver et d'afficher le nom de l'utilisateur aux côtés de ses messages.

Côté js

Côté serveur on va avoir besoin d'une variable `user` pour stocker les informations sur notre utilisateur. Cette variable est locale, elle diffère pour chaque socket (chaque utilisateur connecté). Il faut donc la placer dans la fonction de callback de l'événement 'connection'. Modifiez donc `server.js`.

```
[...]
io.on('connection', function (socket) {
```

```
  /**
   * Utilisateur connecté à la socket
   */
  var loggedUser;
[...]
```

Lorsque que le serveur reçoit l'événement `user-login`, il faut stocker l'utilisateur dans cette variable. Ajoutez donc ces quelques lignes à l'intérieur de la fonction de callback de l'événement 'connection'.

```
  /**
   * Connexion d'un utilisateur via le formulaire
   */
  socket.on('user-login', function (loggedUser) {
    console.log('user logged in : ' + loggedUser.username);
    user = loggedUser;
  });
```

On va également modifier la fonction qui gère la réception des messages. Je vous rappelle le fonctionnement : à chaque message reçu par le serveur, celui-ci émet un événement vers tous les utilisateurs avec le fameux message joint. Nous devont maintenant envoyer également le nom de l'utilisateur émetteur du message.
Modifions donc cette fonction pour y intégrer nos évolutions.

```
  /**
   * Réception de l'événement 'chat-message' et réémission vers tous les utilisateurs
   */
  socket.on('chat-message', function (message) {
    message.username = loggedUser.username; // On intègre ici le nom d'utilisateur au message
    io.emit('chat-message', message);
    console.log('Message de : ' + loggedUser.username);
  });
```

Plus qu'à gérer ce nouvel élément côté client. Lors de la réception d'un message, nous n'allons plus uniquement afficher le message mais également le nom d'utilisateur !

```
/**
 * Réception d'un message
 */
socket.on('chat-message', function (message) {
  $('#messages').append($('<li>').html('<span class="username">' + message.username + '</span> ' + message.text));
});
```

Un peu de style et c'est fini :

```
section#chat #messages li span.username {
    display: inline-block;
    padding: 6px 10px;
    margin-right: 5px;
    color: white;
    background: #e67e22;
    border-radius: 5px;
}
```

On est bons !

Testez en ouvrant plusieurs onglets et en vous connectant avec des noms d'utilisateur différents : cela fonctionne.

Affichage d'un message informant de la connexion/déconnexion d'un utilisateur

Ceci va être bien plus simple. On veut, lorsqu'un utilisateur se connecte, afficher un message dans le chat. De même lorsqu'il se déconnecte.

Côté serveur il suffit d'émettre un événement lors de la connexion d'un utilisateur et un autre lors de la déconnexion (en broadcast, c'est à dire à tous le monde sauf à l'utilisateur connecté à la socket courante).
Nous allons appelé cet événement `service-message`. On joint à l'événement un objet `serviceMessage` qui contient un type (login ou logout) et un texte (qui s'affichera en front-end.
Modifions donc le comportement du serveur lors de la réception des événements `user-login` et `disconnect`.

```
[...]
  /**
   * Déconnexion d'un utilisateur : broadcast d'un 'service-message'
   */
  socket.on('disconnect', function () {
    if (loggedUser !== undefined) {
      console.log('user disconnected : ' + loggedUser.username);
      var serviceMessage = {
        text: 'User "' + loggedUser.username + '" disconnected',
        type: 'logout'
      };
      socket.broadcast.emit('service-message', serviceMessage);
    }
  });

  /**
   * Connexion d'un utilisateur via le formulaire :
   *   - sauvegarde du user
   *   - broadcast d'un 'service-message'
   */
  socket.on('user-login', function (user) {
    loggedUser = user;
    if (loggedUser !== undefined) {
      var serviceMessage = {
```

```
        text: 'User "' + loggedUser.username + '" logged in',
        type: 'login'
      };
      socket.broadcast.emit('service-message', serviceMessage);
    }
  });
[...]
```

Côté client nous allons réceptionner cet événement `service-message` et l'afficher dans la liste des messages.

```
/**
 * Réception d'un message de service
 */
socket.on('service-message', function (message) {
  $('#messages').append($('<li class="' + message.type +
'">').html('<span class="info">information</span> ' +
message.text));
});
```

Un peu de stayyyle, parce que quand même.

```
section#chat #messages li.logout {
    background: #E5A6A6;
}
section#chat #messages li.login {
    background: #A8E5A6;
}
section#chat #messages li span.info {
    display: inline-block;
    padding: 3px 10px;
    margin-right: 5px;
    color: white;
    background: #e67e22;
    border-radius: 5px;
}
```

Redémarrez votre appli, tout devrait rouler :)

On a terminé !

Sources

Vous pouvez retrouver les sources de ce tutoriel sur GitHub. Il est possible de les télécharger directement depuis cette page ou de cloner la branche Git correspondant à cette partie du tutoriel.

```
git clone --branch part-2 https://github.com/BenjaminBini/socket.io-chat.git
```

Conclusion

En ajoutant quelques événements à droite et à gauche on peut réellement commencer à avoir simplement, en quelques lignes de javascript, une application réellement fonctionnelle et utilisable dans le monde réel.

La suite

Dans la suite nous verrons trois dernières fonctionnalités qui paraissent utiles à ajouter :

- Affichage de la liste des utilisateurs connectés
- Affichage d'un message "xxx is typing" quand un utilisateur est en train d'écrire
- Gestion d'un historique de messages (affichage des derniers messages envoyés avant la connexion de l'utilisateur)

Cliquez ici pour accéder à la partie 3

Développer un chat en temps réel avec Socket.io – Partie 3

Nous voici donc arrivés dans cette troisième partie du développement d'un chat temps-réel utilisant Node.js et Socket.io. Nous avons pour l'instant un chat fonctionnelle permettant de se connecter avec un pseudonyme et d'envoyer des messages aux autres utilisateurs. Dans cette partie nous allons faire trois choses :

- Rester en bas de page lorsque de nouveaux messages arrivent
- Afficher la liste des utilisateurs connectés
- Afficher un message "xxx is typing" quand un utilisateur est en train d'écrire
- Gérer un historique de messages (affichage des x derniers messages envoyés avant la connexion de l'utilisateur)

Base de code

Nous repartons du code de la partie 2 pour développer cette troisième partie. Si vous n'avez pas suivi les deux premières parties de ce tutoriel, je vous invite soit à les suivre (première partie, deuxième partie) soit à télécharger son code source sur mon compte GitHubou en clonant la branche git correspondante :

```
git clone --branch part-2 https://github.com/BenjaminBini/socket.io-chat.git
```

Rester en bas de page

Vous avez peut-être remarqué un petit soucis dans notre chat : lorsque les messages partagés sont trop nombreux et qu'ils dépassent de la page, il est nécessaire de scroller afin de les lire. De plus, le dernier message envoyé sera masqué par le formulaire de saisie d'un message.

Ce n'est pas compliqué à corriger.
Pour que le formulaire ne masque pas le dernier message envoyé nous allons ajouter un `padding-bottom` à notre liste de messages.

```
section#chat #messages {
```

```
    list-style-type: none;
    margin: 0;
    padding: 0;
    padding-bottom: 50px;
}
```

Nous souhaitons également que l'on reste en bas de page lors de l'arrivée de nouveaux messages. Mais nous voulons également pouvoir remonter dans la liste des messages reçus sans être dérangé par la réception de ceux-ci !
Nous allons donc créer une fonction qui sera appelée après l'ajout d'un message (une balise `li`) utilisateur ou d'un message de service. Celle-ci va permettre de scroller automatiquement vers le bas de la page uniquement dans le cas où l'utilisateur n'est pas remonté lire les anciens messages (on va laisser une petite marge, disont qu'on scrollera vers le bas si l'utilisateur est remonté d'une distance supérieur à la hauteur d'un message).

Voici la fonction à ajouter au début de `client.js`.

```
/**
 * Scroll vers le bas de page si l'utilisateur n'est pas remonté pour lire d'anciens messages
 */
function scrollToBottom() {
  if ($(window).scrollTop() + $(window).height() + 2 * $('#messages li').last().outerHeight() >= $(document).height()) {
    $("html, body").animate({ scrollTop: $(document).height() }, 0);
  }
}
```

Si comme moi vous utilisez JSLint, je vous invite à ajouter ce commentaire en haut de fichier afin qu'il reconnaisse l'existence de `window`.

```
/*jslint browser: true*/
```

Il n'y a plus qu'à appeler cette fonction lors de la réception d'un message ou d'un message de service.

```
/**
 * Réception d'un message
 */
socket.on('chat-message', function (message) {
```

```
  $('#messages').append($('<li>').html('<span class="username">' +
message.username + '</span> ' + message.text));
  scrollToBottom();
});


/**
 * Réception d'un message de service
 */
socket.on('service-message', function (message) {
  $('#messages').append($('<li class="' + message.type +
'">').html('<span class="info">information</span> ' +
message.text));
  scrollToBottom();
});
```

Et voilà la résultat !

Trop facile. Passons à la suite.

Afficher la liste des utilisateurs connectés

Afin d'afficher la liste des utilisateurs connectés nous devons tout d'abord modifier le front-end (HTML, CSS).
Ajoutons la liste des utilisateurs au fichier `index.html`.

```
<section id="chat">
   <ul id="messages">
   </ul><ul id="users">
   </ul>
   <form action="">
     <input id="m" autocomplete="off" /><button>Send</button>
   </form>
</section>
```

On ne met pas d'espace entre `</ul>` et `<ul id="users">` car cela crée un espace entre les deux div au rendu, ce que nous ne souhaitons pas.

Avec un peu de CSS, le rendu sera sympa.
Ajoutons les règles suivantes :

```
html, body, head {
    height: 100%;
}
section#chat {
    height: 100%;
}
section#chat #users {
    display: inline-block;
    position: fixed;
    vertical-align: top;
    overflow: auto;
    width: 250px;
    list-style-type: none;
    height: 100%;
    padding-bottom: 50px;
    border-left: 3px solid #eee;
}
section#chat #users li.new {
    background: #e67e22;
    color: white;
}
section#chat #users li {
    padding: 6px 10px;
    margin: 10px 10px;
    border-radius: 5px;
    border: 1px solid #e67e22;
    color: black;
    transition: all 0.5s;
}
```

Et ajoutez les règles qui suivent pour le selecteur `section#chat #messages` :

```
    display: inline-block;
    width: calc(100% - 250px);
```

Côté javascript client, on se doute que l'on va devoir gérer deux événements :

- La connexion d'un utilisateur
- Sa déconnexion

Lors de la connexion on va ajouter un élément à notre liste d'utilisateurs alors que l'on va en supprimer un lors de la déconnexion. Afin de rendre le tout un peu plus sexy, on met la classe `new` sur le nouvel élément. Classe que l'on supprime après une seconde (l'animation de la couleur de fond se fait alors grâce au CSS précédent).

```
/**
 * Connexion d'un nouvel utilisateur
 */
socket.on('user-login', function (user) {
  $('#users').append($('<li class="' + user.username + '
new">').html(user.username));
  setTimeout(function () {
    $('#users li.new').removeClass('new');
  }, 1000);
});

/**
 * Déconnexion d'un utilisateur
 */
socket.on('user-logout', function (user) {
  var selector = '#users li.' + user.username;
  $(selector).remove();
});
```

Côté serveur il y a un peu plus de travail, il faut gérer une liste des utilisateurs connectés. En effet, on ne peut pas se contenter de prévenir l'utilisateur qu'un nouvel utilisateur arrive ou qu'un autre s'en va, il faut, lorsqu'il arrive sur la page, qu'il reçoive l'ensemble des utilisateurs connectés.

Pour cela on crée une liste des utilisateurs connectés au début de notre fichier `server.js`.

```
/**
 * List of connected users
 */
var users = [];
```

La gestion de la déconnexion est simple.

```
/**
   * Déconnexion d'un utilisateur
   */
  socket.on('disconnect', function () {
    if (loggedUser !== undefined) {
      // Broadcast d'un 'service-message'
      var serviceMessage = {
        text: 'User "' + loggedUser.username + '" disconnected',
        type: 'logout'
      };
      socket.broadcast.emit('service-message', serviceMessage);
      // Suppression de la liste des connectés
      var userIndex = users.indexOf(loggedUser);
      if (userIndex !== -1) {
        users.splice(userIndex, 1);
      }
      // Emission d'un 'user-logout' contenant le user
      io.emit('user-logout', loggedUser);
    }
  });
```

La connexion est un peu plus compliquée.
Nous allons en effet devoir gérer le fait qu'un utilisateur ne peut pas se connecter avec un pseudonyme déjà existant.
Nous allons donc utiliser une fonctions de callback. Lors de l'émission d'un événement, le client peut ajouter comme paramètre à la fonction `emit` une fonction de callback qui sera appelée par le serveur une fois les traitements nécessaires effectués. On va donc ajouter une fonction de callback à l'événement `user-login` émit par le client qui renverra `true` ou `false` selon le succès ou l'échec de la connexion. Voilà comment gérer cela côté client :

```
/**
 * Connexion de l'utilisateur
 * Uniquement si le username n'est pas vide et n'existe pas encore
 */
$('#login form').submit(function (e) {
  e.preventDefault();
  var user = {
    username : $('#login input').val().trim()
  };
  if (user.username.length > 0) { // Si le champ de connexion n'est pas vide
    socket.emit('user-login', user, function (success) {
      if (success) {
        $('body').removeAttr('id'); // Cache formulaire de connexion
        $('#chat input').focus(); // Focus sur le champ du message
      }
    });
  }
});
```

On n'affiche le chat que si `success` vaut `true`.
Côté serveur il nous reste à appeler la fonction de callback et à émettre un événement contenant l'utilisateur connecté à tous les utilisateurs.

```
  /**
   * Connexion d'un utilisateur via le formulaire :
   */
  socket.on('user-login', function (user, callback) {
    // Vérification que l'utilisateur n'existe pas
    var userIndex = -1;
    for (i = 0; i < users.length; i++) {
      if (users[i].username === user.username) {
        userIndex = i;
      }
    }
    if (user !== undefined && userIndex === -1) { // S'il est bien nouveau
```

```
      // Sauvegarde de l'utilisateur et ajout à la liste des
connectés
      loggedUser = user;
      users.push(loggedUser);
      // Envoi des messages de service
      var userServiceMessage = {
        text: 'You logged in as "' + loggedUser.username + '"',
        type: 'login'
      };
      var broadcastedServiceMessage = {
        text: 'User "' + loggedUser.username + '" logged in',
        type: 'login'
      };
      socket.emit('service-message', userServiceMessage);
      socket.broadcast.emit('service-message',
broadcastedServiceMessage);
      // Emission de 'user-login' et appel du callback
      io.emit('user-login', loggedUser);
      callback(true);
    } else {
      callback(false);
    }
  });
```

J'en profite pour ajouter l'émission d'un message de service indiquant "You logged in as ..." à l'utilisateur qui vient de se connecter.

On doit encore envoyer la liste des utilisateurs connectés aux utilisateurs qui viennent d'arriver sur le chat. Pour cela on va envoyer au nouvaux arrivant un événement 'user-login' pour chaque utilisateur déjà connecté.

```
io.on('connection', function (socket) {
[...]
  /**
   * Emission d'un événement "user-login" pour chaque utilisateur
connecté
   */
  for (i = 0; i < users.length; i++) {
```

```
    socket.emit('user-login', users[i]);
  }
[...]
```

On est bons ! Testez tout cela, ça devrait tourner sans soucis.

Gestion d'un historique de messages

Quand on arrive sur le chat, il serait plutôt cool d'avoir accès aux derniers messages partagés entre les utilisateurs.
Pour cela, tout comme on a une liste des utilisateurs connectés, on va gérer une liste des derniers messages envoyés.

Ajoutez au début de `server.js` :

```
/**
 * Historique des messages
 */
var messages = [];
```

A chaque nouveau message envoyé, on va ajouter ce dernier à la liste. Mais on ne veut pas qu'elle grossisse indéfiniment. On va donc définir une taille maximale de 150 messages à partir de laquelle tout ajout entraîne la suppression du plus ancien message.
Modifions la gestion de l'événement `chat-message`.

```
  /**
   * Réception de l'événement 'chat-message' et réémission vers tous les utilisateurs
   * Ajout à la liste des messages et purge si nécessaire
   */
  socket.on('chat-message', function (message) {
    message.username = loggedUser.username;
    io.emit('chat-message', message);
    messages.push(message);
    if (messages.length > 150) {
```

```
      messages.splice(0, 1);
    }
  });
```

On veut également enregistrer les messages de service (connexion et déconnexion des utilisateurs). Il faut donc aussi les ajouter à la liste.

Dans la gestion de l'événement `user-login`, ajoutons la troisième ligne ci-dessous.

```
 socket.emit('service-message', userServiceMessage);
 socket.broadcast.emit('service-message', broadcastedServiceMessage);
 messages.push(broadcastedServiceMessage);
```

Dans la gestion de l'événement `disconnect`, ajoutons la dernière ligne ci-dessous.

```
// Suppression de la liste des connectés
var userIndex = users.indexOf(loggedUser);
if (userIndex !== -1) {
    users.splice(userIndex, 1);
}
// Ajout du message à l'historique
messages.push(serviceMessage);
```

Reste à émettre un événement pour chaque message enregistré lors de l'arrivée d'un nouvel utilisateur. Le type de l'événement dépendra du type de message : classique ou de service. On fait la différence via la présence ou nom de la propriété `username` de l'objet.

```
io.on('connection', function (socket) {
[...]
  /**
   * Emission d'un événement "chat-message" pour chaque message de l'historique
   */
  for (i = 0; i < messages.length; i++) {
    if (messages[i].username !== undefined) {
      socket.emit('chat-message', messages[i]);
```

```
    } else {
      socket.emit('service-message', messages[i]);
    }
  }
[...]
```

Dernier petit détail d'ordre graphique : les messages sont affichés sur l'écran (mais floutés "sous" le formulaire de connexion) avant que l'utilisateur saisisse son username. Si la liste des messages enregistrés est grande cela créera un bug du design de la page. Pour le corriger, dans le ficher css on va modifier la propriété `position` de la `section#login`. De `absolute`, on la fait passer à `fixed`.

Voilà ! Terminé.
Next step.

Notification d'une saisie en cours

Une fonctionnalité plutôt utile pour une application de chat est de voir qui est en train d'écrire un message. Où afficher cela ? Dans la liste des utilisateurs, à droite du nom, me semble être une bonne solution.
Comment cela va fonctionner ?
Côté client on va ajouter une balise `span` à droite du nom de l'utilisateur contenant le texte "typing". On ne l'affichera que quand l'utilisateur en question sera en train de taper.

Trois événements sont à gérer :

- `start-typing` : envoyé par le client lorsque l'utilisateur commence à écrire
- `stop-typing` : envoyé par le client lorsque l'utilisateur a arrêter d'écrire
- `update-typing` : envoyé par le serveur quand une modification de la liste des utilisateurs en train d'écrire est arrivée

Commençons par le côté serveur.
Nous avons besoin d'une liste globale des utilisateurs en train d'écrire, à ajouter en début de fichier.

```
/**
 * Liste des utilisateurs en train de saisir un message
 */
var typingUsers = [];
```

Gérons maintenant les deux événements `start-typing` et `stop-typing`.

```
 /**
   * Réception de l'événement 'start-typing'
   * L'utilisateur commence à saisir son message
   */
  socket.on('start-typing', function () {
    // Ajout du user à la liste des utilisateurs en cours de saisie
    if (typingUsers.indexOf(loggedUser) === -1) {
      typingUsers.push(loggedUser);
    }
    io.emit('update-typing', typingUsers);
  });

  /**
   * Réception de l'événement 'stop-typing'
   * L'utilisateur a arrêter de saisir son message
   */
  socket.on('stop-typing', function () {
    var typingUserIndex = typingUsers.indexOf(loggedUser);
    if (typingUserIndex !== -1) {
      typingUsers.splice(typingUserIndex, 1);
    }
    io.emit('update-typing', typingUsers);
  });
```

On ne fait que recevoir les événements, ajouter ou supprimer l'utilisateur de la liste selon l'événement et émettre l'événement `update-typing` en y joignant la liste des utilisateurs en cours de saisie.

Dernier petit détail : gérer la déconnexion des utilisateurs. Si un utilsateur est déconnecté alors qu'il est en train d'écrire un message, il n'est pas supprimé de

la liste. Gérons cela en ajoutant ces quelques lignes à la fin de la fonction gérant l'événement `disconnect`.

```
  socket.on('disconnect', function () {
    if (loggedUser !== undefined) {
      [...]
      // Si jamais il était en train de saisir un texte, on l'enlève
de la liste
      var typingUserIndex = typingUsers.indexOf(loggedUser);
      if (typingUserIndex !== -1) {
        typingUsers.splice(typingUserIndex, 1);
      }
    }
  });
```

Côté client maintenant, on va commencer par ajouter la balise `<span>` à côté du nom des utilisateurs.

```
/**
 * Connection d'un nouvel utilisateur
 */
socket.on('user-login', function (user) {
  $('#users').append($('<li class="' + user.username + '
new">').html(user.username + '<span class="typing">typing</span>'));
  setTimeout(function () {
    $('#users li.new').removeClass('new');
  }, 1000);
});
```

Pour gérer l'émission des événements `start-typing` et `stop-typing`, nous allons utiliser les événements `keypress` et `keyup`. Pourquoi pas `keydown` ?
Parce que cet événement est émis même lors de la frappe des touches `Ctrl`, `Alt`, etc. Ce qui n'est pas désiré ici.
On utilisera setTimeout afin d'autoriser un délais de 0.5 secondes entre la frappe de chaque touche.

```
/**
 * Détection saisie utilisateur
 */
```

```
var typingTimer;
var isTyping = false;

$('#m').keypress(function () {
  clearTimeout(typingTimer);
  if (!isTyping) {
    socket.emit('start-typing');
    isTyping = true;
  }
});

$('#m').keyup(function () {
  clearTimeout(typingTimer);
  typingTimer = setTimeout(function () {
    if (isTyping) {
      socket.emit('stop-typing');
      isTyping = false;
    }
  }, 500);
});
```

Passons à la gestion de la réception de l'événement `update-typing` pour mettre à jour l'UI.

```
/**
 * Gestion saisie des autres utilisateurs
 */
socket.on('update-typing', function (typingUsers) {
  $('#users li span.typing').hide();
  for (i = 0; i < typingUsers.length; i++) {
    $('#users li.' + typingUsers[i].username + '
span.typing').show();
  }
});
```

Et il n'y a plus qu'à rajouter le style de cette balise span.

```
section#chat #users li span.typing {
    float: right;
    font-style: italic;
    color: #eee;
    display: none;
}
```

Enfin terminé !

Ces trois parties nous ont permis de découvrir plein de choses sur le fonctionnement de socket.io avec node.js et jQuery. Vous pouvez bien sûr essayer d'aller plus loin en ajoutant par exemple plusieurs salles de chat ou bien des messages privés entre utilisateurs.

Sources

Comme d'habitude, les sources sont sur Github : à télécharger ou en clonant la branche git correspondante.

```
git clone --branch part-3 https://github.com/BenjaminBini/socket.io-chat.git
```