Intervenant : Chouki TIBERMACINE

TP
Websockets avec JavaScript et Node.js

Objectifs du TP.
- Écrire un serveur Node.js simple capable de communiquer avec le protocole Websockets
- Écrire un client JavaScript qui communique avec ce serveur

Mini-tutoriel Websockets.
Afin, de créer une application qui utilise les Websockets, il nous faudra un client et un serveur.
Le serveur :
Il peut être écrit dans n'importe quel langage (Php, JS, …).
L'essentiel est d'utiliser les librairies nécessaires pour créer et démarrer un programme serveur, qui peut écouter et émettre des requêtes Websockets.
Dans ce TP, nous allons utiliser le langage JavaScript pour écrire un serveur de Websockets.
Il nous faudra donc lancer des programmes JavaScript, côté serveur.
Cela peut être fait grâce à l'environnement Node.js.
Nous devons écrire un programme JavaScript, qui utilise l'API fournie par cet environnement, et l'exécuter en utilisant la commande node ou nodejs.
Pour exécuter un programme Node.js, il suffit de taper la commande :

```
node votre_script_serveur.js
```

Sous certains OS, remplacer la commande node par nodejs.

Pour vous servir de Node.js sur vos machines personnelles, il faudrait le télécharger, accompagné de npm (*Node.js package manager*), sur le site officiel de Node.js, puis l'installer (il faut avoir les droits d'administrateur).
Sous Linux : `sudo apt-get install nodejs npm`

Avant d'exécuter un script qui démarre un serveur Websocket, il faudra installer le module ws en tapant la commande suivante :

```
npm install ws
```

Pour commencer, nous allons écrire un script qui permet de créer et démarrer un serveur HTTP (pas Websockets). L'objectif est de tester des programmes serveurs Node.js.
Notez que, contrairement au module ws, le module http est fourni par défaut dans la version actuelle de Node.js. Pas besoin donc de l'installer avec npm.

Le script serveur doit comporter les instructions suivantes :

```
// Création d'un serveur avec une fonction callback qui attend
// des requêtes HTTP et qui renvoie une réponse HTTP
var serveur = require("http");// Charger le module http de Node.js
serveur.createServer(function (req, rep) { // Callback
    rep.writeHead(200,{"Content-Type":"text/html; charset=utf-8"});
    rep.write("Bonjour à tous");
    rep.end();
}).listen(8080);
```

Ce code crée un serveur qui, une fois démarré avec la commande node est à l'écoute de requêtes HTTP sur le port 8080.
Dès la réception d'une requête, il renvoie au client le message « Bonjour à tous », en invoquant les méthodes de l'objet reçu en paramètre (rep) de la callback. (Il ne fait cependant rien avec la requête du client, le paramètre req.)
Pour tester ce serveur HTTP, il suffit de taper sur le navigateur (le client) l'URL :
`http://localhost:8080`

Pour écrire un serveur Websockets, les instructions sont légèrement différentes :

```
var WebSocketServer = require("ws").Server;// Charger le module ws
var serveur = new WebSocketServer( { port: 8100 } );
console.log("Serveur démarré...");
serveur.on("connection", function (ws) {
    console.log("Client connecté...");
    ws.on("message", function (str) {
        console.log("Message reçu : " + str);
        ws.send("Serveur a bien reçu votre message.");
    });
    ws.on("close", function() {
        console.log("Client déconnecté.")
    });
});
```

Ce code crée un serveur, qui est à l'écoute de demandes de connexion Websockets, sur le port 8100. Il est aussi capable d'envoyer des messages aux clients qui sont connectés à lui.
Dans ce code, on a associé à l'objet serveur un écouteur d'événement de type « demande de connexion ». A la réception d'une telle demande, le serveur affiche sur la console le message « Client connecté... ». Ensuite, il attend, grâce à l'écouteur d'événements de type « réception de message », la réception d'un message du client. Cet écouteur est associé à l'objet WebSocket (ws) reçu en paramètre. Dès réception d'un message, il affiche « Message reçu : », suivi du message envoyé par le client (str, paramètre de la fonction callback). Il envoie après un message au client : « Serveur a bien reçu votre message. ». Ceci est fait grâce à la méthode send de l'objet WebSocket.
Enfin, le script ci-dessus définit un écouteur d'événement de type « fermeture de connexion » (close). Dès qu'un client se déconnecte, le serveur affiche sur la console « Client déconnecté. ».
Pour arrêter le serveur, il faut tuer le processus node en faisant un Ctrl-C, par exemple.
Le client :
Comme nous l'avons précisé dans le cours, l'un des avantages les plus importants des Websockets est la possibilité pour le serveur d'envoyer un message à un client (« *server push* ») sans que ça soit fait en réponse à une requête provenant de ce dernier (comme c'est le cas avec HTTP, qui est un protocole de requête-réponse).
De ce fait, le serveur peut à tout moment décider d'envoyer un message au client (connecté à lui) pour qu'il mette à jour, par exemple, des données affichées à l'utilisateur.

Le script côté client ne nécessitera l'utilisation d'aucune librairie particulière. L'interface de programmation (API) des Websockets est standardisée et fournie par défaut par les interprètes JavaScript intégrés aux navigateurs Web (Chrome et Firefox testés).

Voici un exemple de script côté client : (à insérer dans un document HTML)

```
var ws = new WebSocket("ws://localhost:8100");
$(ws).on("open", function (event) {
 $("#message").append("Connexion établie avec le serveur. <br/>");
 ws.send("Client prêt");
});
$(ws).on("message",function(event) {
 $("#message").append(event.originalEvent.data+"<br/>");
});
$(ws).on("close",function(event) {
 $("#message").append("Le serveur s'est déconnecté.");
});
```

Dans ce script écrit avec jQuery, on crée d'abord un objet WebSocket, qui va envoyer une demande de connexion au serveur Websocket démarré sur localhost:8100.
Ensuite, dans ce script, on a associé 3 écouteurs d'événements, qui vont effectuer des actions, à l'ouverture de la connexion (écouteur onopen), à la réception d'un message du serveur (onmessage) et à la fermeture de la connexion (onclose).
A l'établissement de la connexion, le client insère le message « Connexion établie avec le serveur. » dans un div (ayant l'ID « message »). Par la suite, le client envoie un message au serveur « Client prêt », grâce à la méthode send de l'objet WebSocket.
A la réception d'un message du serveur, le client insère le message dans le même div.
Le client récupère le message envoyé par le serveur en utilisant l'objet événement (paramètre de la fonction callback). Étant donné que l'objet événement est un objet jQuery (écouteur associé avec la méthode on() de jQuery), il a fallu dans le code ci-dessus accéder à sa propriété originalEvent pour obtenir l'objet événement JS original (non enveloppé par jQuery). Dans ce dernier, il y a une propriété data, qui contient le message envoyé par le serveur, d'où l'expression : `event.originalEvent.data`
A la fermeture de la connexion par le serveur, le client affiche le message « Le serveur s'est déconnecté ».
- Ouvrez le document HTML, contenant le script, sur votre navigateur. (Je suppose que le serveur est déjà démarré.)
- Qu'est-ce qui est affiché ? Quels écouteurs d'événements sont exécutés ?
Qu'est-ce qui est affiché sur la console dans laquelle vous avez démarré le serveur ?
- Quand vous actualisez la page Web du client, votre navigateur ferme d'abord la connexion avec le serveur, avant d'envoyer une demande de connexion. Quels écouteurs sont exécutés ?
- Arrêtez le serveur (Ctrl-C sur le terminal). Qu'est-ce qui se passe sur le navigateur ?

Exercice 1.

Écrire une page Web contenant un compteur : un bloc div contenant le nombre 0, accompagné de trois boutons : + + (pour incrémenter), - - (pour décrémenter) et RAZ (pour le remettre à zéro).

1. Écrire le script JS côté client nécessaire à l'implémentation du compteur
2. Ajouter le code nécessaire à l'envoi et à la récupération de la valeur du compteur à partir d'un serveur Websocket (localhost:8100)
3. Implémenter le script serveur Node.js qui a pour rôle de recevoir la valeur du compteur et de sa diffusion à tous les clients connectés

4. Gérer dans le code JS client la désactivation des boutons du compteur, tant que le serveur n'est pas encore démarré et n'a pas encore accepté la demande de connexion. Désactiver ces boutons lorsque le serveur est arrêté
5. Tester le client sur plusieurs navigateurs (Chrome et Firefox par exemple) ou sur plusieurs fenêtres d'un même navigateur

Exercice 2.

Écrire un éditeur collaboratif de fichiers texte, .js par exemple (hébergés sur un serveur). Il s'agit d'une simple page HTML, qui affiche dans un champ de type textarea, le contenu d'un fichier texte obtenu via une Websocket à partir d'un serveur. Vous pouvez aussi utiliser un éditeur plus sophistiqué comme Ace (https://ace.c9.io/). (On suppose pour des raisons de simplicité que le fichier ouvert est celui indiqué dans la *query string (editeur.html?file=script.js)*. Le fichier ouvert doit se trouver dans un répertoire particulier : *public_files*, par exemple. Sinon, rien ne s'affichera dans le *textarea ou le * de l'éditeur Ace.) Le texte est éditable, et pour tout caractère saisi (l'événement *change* doit être écouté), le nouveau contenu du textarea doit être transmis au serveur par le script JS. Côté serveur, le nouveau contenu doit être écrit dans le fichier. Pour des raisons de simplicité, on suppose que les fichiers .js édités sont de petite taille et c'est tout le contenu du fichier qui est échangé à chaque fois.
Il est possible de faire mieux que cela, en utilisant des librairies JS qui calculent des diff entre fichiers de type texte, comme jsdiff (https://github.com/kpdecker/jsdiff).

Afin de lire un fichier (texte) avec Node.js, on peut utiliser le module standard fs et la méthode readFile de la façon suivante :

```
var fs = require("fs") ;
var fichier = "<chemin vers le fichier texte>";
fs.readFile(fichier, "utf8", function (err,data) {
    if (err) {
        console.log(err);
    }
    console.log(data) ; // Afficher le contenu du fichier
                        // dans la console du navigateur
                        // ws.send(data) ; // l'envoyer au client
});
```

Le nom du fichier peut être reçu comme message du client. Le script côté client peut le récupérer de la query string de la façon suivante :

```
var url = document.location.toString();
var str = "?file=";
var fichier = url.substring(url.indexOf(str)+str.length);
```

Pour écrire du texte (une chaîne de caractères) dans un fichier, on peut continuer à utiliser le module fs de Node.js, et invoquer la méthode writeFile de la façon suivante :

```
fs.writeFile(fichier, chaine, "utf8", function(err) {
    if(err) {
        console.log(err);
    }
    console.log("Fichier enregistré ");
});
```

Si un autre client, accédant à la même page HTML avec le même URL, ouvre le fichier (le serveur doit gérer plusieurs connexions/clients à la fois), il peut éditer ce même fichier. Dès que le serveur reçoit le nouveau contenu de cet utilisateur, ce contenu est

envoyé à l'autre utilisateur (ou aux autres utilisateurs, le cas échéant).
Nous allons considérer, pour des raisons de simplicité, que le contenu ouvert chez un utilisateur est écrasé par ce que lui envoie le serveur. L'idée dans cet exercice est que plusieurs utilisateurs visualisent un même fichier, édité par une seule personne.

Tester votre application sur des machines différentes.

Si la saisie devient lente, prévoir un temporisateur, qui propage les changements vers le serveur toutes les x secondes.

Exercice 3.

Écrire une application simple de Chat en utilisant les Websockets

L'application doit être composée d'un serveur et d'un client, qui peut être exécuté deux fois au maximum pour un serveur donné (version 1 de l'application).

Le client doit fournir un champ de saisie de texte pour introduire son pseudo.

Une fois connecté à un deuxième client, un bloc div s'affiche pour visualiser les messages échangés.

Le serveur joue le rôle de relais entre les 2 « instances » du client.

Tester l'application en lançant le serveur et les clients sur des machines différentes

Changer l'application pour prendre en compte un nombre quelconque d'utilisateurs