Chapitre 3 : Node.js, Express, socketIO

Objectifs :

- ✓ Ddécouvrir et comprendre le framework Express et la bibliothèque SocketIO
- ✓ Configurer l'environnement pour socketIO et Express
- ✓ Gérer facilement les paramètres de requêtes et les routes
- ✓ Créer une application de tchat

Prérequis : Maitriser la séquence 1

1. Express

1.1. Introduction

Express.js est un **framework** pour **construire des applications web basées sur Node.js**. C'est de fait le framework standard pour le développement de serveur en Node.js. L'auteur original, TJ Holowaychuck, le décrit comme un serveur inspiré de Sinatra (en) dans le sens qu'il est relativement minimaliste tout en permettant d'étendre ses fonctionnalités via des plugins.
Express vous permet donc d'être "un peu moins bas niveau" et de gérer par exemple plus facilement les routes (URL) de votre application et d'utiliser des templates. Rien que ça, ça va déjà être une petite révolution pour nous !

1.2. Installation

Express s'installe très facilement en utilisant NPM comme suivant :

```
npm install express
```

Après cette installation vous pouvez commencer à utiliser express

1.3. Les routes

Dans la séquence 1 Nous avons vu à quel point il était fastidieux de vérifier l'URL demandée avec Node.js. On aboutissait à du code spaghetti du type :

```
if (page == '/') {
    // ...
}
else if (page == '/sous-sol') {
    // ...
}
else if (page == '/etage/1/chambre') {
    // ...
}
```

Lorsqu'on crée des applications web, on manipule des routes comme ici. Ce sont les différentes URL auxquelles notre application doit répondre.
La gestion des routes un sujet important qui doit être traité avec sérieux. Bien gérer les URL de son site est important, surtout lorsque celui-ci grossit. Express nous aide à faire ça bien.

1.3.1. Routes simples

Avec express les route deviennent très facile à gérer avec la méthode **get()** qui prend en paramètre l'url et la fonction callback à exécuter.

Voici une application très basique utilisant Express pour commencer :

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

var app = express();

app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.send('Vous êtes à l\'accueil');
});

app.listen(8080);
```

Il vous suffit d'indiquer les différentes routes (les différentes URL) à laquelle votre application doit répondre. Dans cet exemple on a créé une seule route, **la racine "/".** Une fonction de callback est appelée quand quelqu'un demande cette route.

Ce système est beaucoup mieux fait que les **"if" imbriqués.** On peut écrire autant de routes de cette façon qu'on le souhaite :

```
app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.send('Vous êtes à l\'accueil, que puis-je pour vous ?');
});

app.get('/sous-sol', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.send('Vous êtes dans la cave à vins, ces bouteilles sont à moi !');
});

app.get('/etage/1/chambre', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.send('Hé ho, c\'est privé ici !');
});
```

Pour gérer les erreurs 404, vous devez inclure les lignes suivantes à la fin de votre code obligatoirement (juste avant **app.listen**) :

```
// ... Tout le code de gestion des routes (app.get) se trouve au-dessus

app.use(function(req, res, next){
    res.setHeader('Content-Type', 'text/plain');
    res.status(404).send('Page introuvable !');
});
app.listen(8080);
```

Express vous permet de chaîner les appel à **get()** et **use().** Cela revient à faire app.get().get().get()... Ca marche parce que ces fonctions se renvoient l'une à l'autre l'objet app, ce qui nous permet de raccourcir notre code.

```
app.get('/', function(req, res) {

})
.get('/sous-sol', function(req, res) {

})
.get('/etage/1/chambre', function(req, res) {

})
.use(function(req, res, next){

});
```

1.3.2. Routes dynamiques

Express vous permet de gérer des routes dynamiques, c'est-à-dire des routes dont certaines portions peuvent varier. Vous devez écrire: **nomvariable** dans l'URL de la route, ce qui aura pour effet de créer un paramètre accessible depuis **req.params.nomvariable**

Exemple :

```
app.get('/etage/:etagenum/chambre', function(req, res) {
  res.setHeader('Content-Type', 'text/plain');
  res.end('Vous êtes à la chambre de l\'étage n°' + req.params.etagenum);
});
```

Cela vous permet de créer de belles URL et vous évite d'avoir à passer par le suffixe **("?variable=valeur")** pour gérer des variables. Ainsi, toutes les routes suivantes sont valides :

- ✓ /etage/1/chambre
- ✓ /etage/2/chambre
- ✓ /etage/3/chambre
- ✓ /etage/nawak/chambre

1.4. Les templates

Jusqu'ici, nous avions renvoyé le code HTML directement en JavaScript. Cela nous avait donné du code lourd et délicat à maintenir qui ressemblait à ça :

```
res.write('<!DOCTYPE html>'+
'<html>'+
   '<head>'+
      '<meta charset="utf-8" />'+
      '<title>Ma page Node.js !</title>'+
   '</head>'+
   '<body>'+
         '<p>Voici un paragraphe <strong>HTML</strong> !</p>'+
   '</body>'+
'</html>');
```

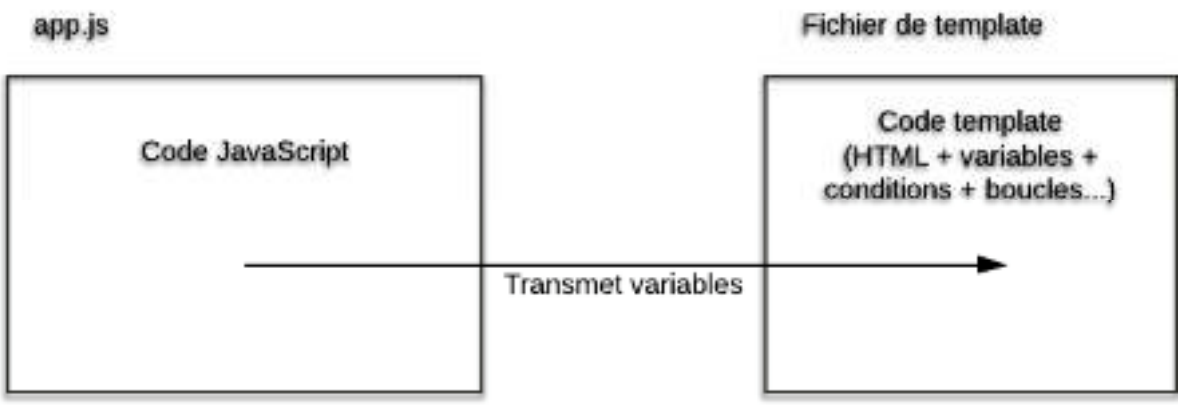
Express nous permet d'utiliser des templates pour sortir de ce surcharge de code**. Les templates sont en quelque sorte des langages faciles à écrire qui nous permettent de produire du HTML et d'insérer au milieu du contenu variable**.

PHP lui-même est en fait un langage de **template** qui nous permet de faire ceci :

```
<p> Bonjour  <?php echo $user; ?></p>
```

Il existe beaucoup d'autres langages de templates, comme **Twig, Smarty, Haml, JSP, Jade, EJS**... Express vous permet d'utiliser la plupart d'entre eux, chacun ayant son lot d'avantages et d'inconvénients. En général ils gèrent tous l'essentiel, à savoir : **les variables, les conditions, les boucles, etc**.

Le principe est que depuis votre fichier JavaScript, vous appelez le template de votre choix en lui transmettant les variables dont il a besoin pour construire la page.



1.5. Les bases du système de gestion de templates EJS

Il existe de nombreux systèmes de templates mais dans ce cours nous allons voir EJS (Embedded JavaScript).

EJS permet une séparation claire entre vue et contrôleur et évite de dupliquer du code HTML

Les vues sont enregistrés dans des fichiers spécifique de type fichier.ejs

1.5.1. Installation

EJS s'installe très facilement en utilisant NPM comme suivant :

```
npm install ejs
```

Après cette installation vous pouvez commencer à utiliser EJS

1.5.2. Déléguer la gestion de la vue a un fichier EJS

Nous pouvons maintenant déléguer la gestion de la vue (du HTML) à notre moteur de template.

Nous avons plus besoin d'écrire du HTML au milieu du code JavaScript

Exemple :

```
app.get('/etage/:etagenum/chambre', function(req, res) {
  res.render('test.ejs', {nom: req.params.nom});
});
```

Ce code fait appel à un fichier **test.ejs** qui doit se trouver dans un sous-dossier appelé **"views".**

Le fichier **test.ejs** contient le code suivant :

```
<p>Bonjour <%= nom %></p>
```

a balise<%= nom %>sera remplacée par la variable **nom** que l'on a transmise au template avec **{nom: req.params.nom}**!

1.5.3. Les paramètres et les boucles

La force des moteur de Template et la possibilité de de rendre les vues dynamiques. EJS offre la possibilité de faire des boucles et aussi de récupérer et afficher des variables passé en paramètre. Ce qui n'est pas possible avec le HTML seulement.

Le principe est que tout ce qui se trouve dans la balise **<% ...%>** est exécuté en tant que code **JavaScript** et tout ce qui se trouve dans la balise **<% = ...%>** est remplacé par la valeur évaluée.

Exemple: Application qui compte jusqu'à un nombre envoyé en paramètre et qui affiche un nom au hasard au sein d'un tableau.

Code javascript :

```
app.get('/compter/:nombre', function(req, res) {
   var noms = ['Robert', 'Jacques', 'David'];
   res.render('page.ejs', {compteur: req.params.nombre, noms: noms});
});
```

On transmet le **nombre** envoyé en paramètre et **une liste de noms** sous forme de tableau. Ensuite, dans le templates EJS on aura:

```
<h1>Je vais compter jusqu'à <%= compteur %></h1>

<p><%
   for(var i = 1 ; i <= compteur ; i++) {
   %>

   <%= i %>...

<% } %></p>

<p>Tant que j'y suis, je prends un nom au hasard qu'on m'a envoyé :
<%= noms[Math.round(Math.random() * (noms.length - 1))] %>
</p>
```

1.5.4. Utilisation de middleware

Express est une infrastructure web middleware et de routage, qui a des fonctions propres minimes : **une application Express n'est ni plus ni moins qu'une succession d'appels de fonctions middleware.**

Les fonctions de middleware sont des fonctions qui peuvent accéder à **l'objet Request (req), l'objet response (res)** et à la fonction middleware suivant dans le cycle demande-réponse de l'application. La fonction middleware suivant est couramment désignée par une variable nommée **next**.

Les fonctions middleware effectuent les tâches suivantes :

- ✓ Exécuter tout type de code.
- ✓ Apporter des modifications aux objets de demande et de réponse.
- ✓ Terminer le cycle de demande-réponse.
- ✓ Appeler la fonction middleware suivant dans la pile.

Si la fonction **middleware** en cours ne termine pas le cycle de demande-réponse, elle doit appeler la fonction **next()** pour transmettre le contrôle à la fonction middleware suivant. Sinon, la demande restera bloquée.

Une application Express peut utiliser les types de middleware suivants :

- ✓ **Middleware niveau application**
- ✓ **Middleware niveau routeur**
- ✓ **Middleware de traitement d'erreurs**
- ✓ **Middleware intégré**
- ✓ **Middleware tiers**

Vous pouvez charger le middleware niveau application et niveau routeur à l'aide d'un chemin de montage facultatif. Vous pouvez également charger une série de fonctions **middleware** ensemble, ce qui crée une s**ous-pile** du système de middleware à un point de montage.

Middleware niveau application

On peut lier le middleware niveau application à une instance de l'objet **app object** en utilisant les fonctions app.use() et app.METHOD(), où METHOD est la méthode HTTP de la demande que gère la fonction middleware (par exemple GET, PUT ou POST) en minuscules.

Cet exemple illustre une fonction middleware sans chemin de montage. La fonction est exécutée à chaque fois que l'application reçoit une demande.

```
var app = express();

app.use(function (req, res, next) {
  console.log('Heure:', Date.now());
  next();
});
```

Exemple 1 : Fonction middleware montée sur le chemin **/user/:id**. La fonction est exécutée pour tout type de demande **HTTP** sur le chemin **/user/:id**.

```
app.use('/user/:id', function (req, res, next) {
  console.log('Type de requete:', req.method);
  next();
});
```

Exemple 2: Route et sa fonction de gestionnaire (système de middleware). La fonction gère les demandes **GET** adressées au chemin **/user/:id**.

```
app.get('/user/:id', function (req, res, next) {
  res.send('USER');
});
```

Exemple 3: chargement d'une série de fonctions middleware sur un point de montage, avec un chemin de montage. Il illustre une **sous-pile** de middleware qui imprime les infos de demande pour tout type de demande **HTTP** adressée au chemin **/user/:id**.

```
app.use('/user/:id', function(req, res, next) {
  console.log('Request URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
```

Les gestionnaires de routage vous permettent de définir plusieurs routes pour un chemin. L'exemple ci-dessous (Exemple 4) définit deux routes pour les demandes GET adressées au chemin **/user/:id**. La deuxième route ne causera aucun problème, mais ne sera jamais appelée puisque la première route boucle le cycle demande-réponse.

Exemple 4 : Une sous-pile de middleware qui gère les demandes GET adressées au chemin **/user/:id**.

```
app.get('/user/:id', function (req, res, next) {
  console.log('ID:', req.params.id);
  next();
}, function (req, res, next) {
  res.send('User Info');
});
app.get('/user/:id', function (req, res, next) {
  res.end(req.params.id);
});
```

Pour ignorer les fonctions middleware issues d'une pile de middleware de routeur, appelez **next('route')** pour passer le contrôle à la prochaine route.

NB : **next('route')** ne fonctionnera qu'avec les fonctions middleware qui ont été chargées via les fonctions **app.METHOD()** ou **router.METHOD().**

Exemple 5 : sous-pile de middleware qui gère les demandes **GET** adressées au chemin **/user/:id**.

```
app.get('/user/:id', function (req, res, next) {
        // si id==0, allez sur la prochaine route
        if (req.params.id == 0) next('route');
        // Passe le control au prochain midleware
        else next(); //
}, function (req, res, next) {
        res.render('regular');
});
app.get('/user/:id', function (req, res, next) {
  res.render('special');
});
```

Middleware niveau routeur

Le **middleware niveau routeur** fonctionne de la même manière que le **middleware niveau application,** à l'exception près qu'il est lié à une instance de **express.Router()**.

On charge le middleware niveau routeur par le biais des fonctions **router.use()** et **router.METHOD()**.

Exemple : Le code d'exemple suivant réplique le système de middleware illustré ci-dessus pour le middleware niveau application, en utilisant un middleware niveau routeur :

```
var app = express();
var router = express.Router();
router.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});
router.use('/user/:id', function(req, res, next) {
  console.log('Request URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
router.get('/user/:id', function (req, res, next) {
  if (req.params.id == 0) next('route');
  else next(); //
}, function (req, res, next) {
  res.render('regular');
});
router.get('/user/:id', function (req, res, next) {
  console.log(req.params.id);
  res.render('special');
});
app.use('/', router);
```

2. Socket.io

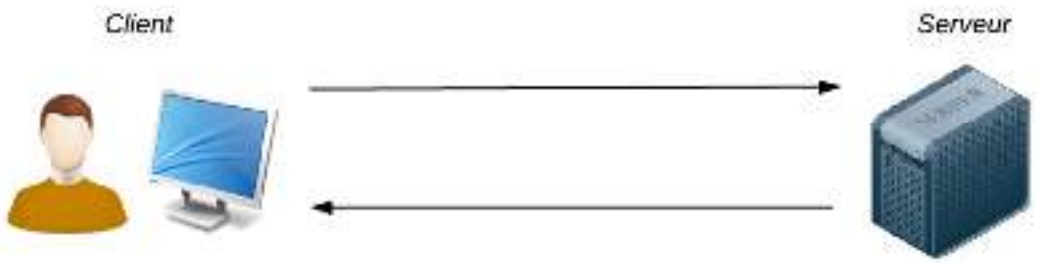
Socket.io est un module de **Node.js** qui permet de créer des **Web Sockets**, c'est-à-dire des connections **bi-directionnelles** entre clients et serveur qui permettent une communication en temps réel sur un autre protocole que le protocole **http** normalement utilisé dans les pages web. Ce type de technique est utilisée pour créer des applications telles que des systèmes de communication en temps réel **(i.e. des Chats),** des jeux multi-utilisateur, des applications de collaboration, etc.

2.1. principe

Les possibilités que vous offre socket.io sont en réalité immenses et vont bien au-delà du Chat : tout ce qui nécessite une communication immédiate entre les visiteurs de votre site peut en bénéficier. Ca peut être par exemple une brique de base pour mettre en place un jeu où on voit en direct les personnages évoluer dans le navigateur, le tout sans avoir à recharger la page .
socket.io se base sur plusieurs techniques différentes qui permettent la communication en temps réel (et qui pour certaines existent depuis des années !). La plus connue d'entre elles, et la plus récente, est **WebSocket.**

WebSocket est une API JavaScript apparue plus ou moins en même temps que HTML5. Elle est une fonctionnalité supportée par l'ensemble des navigateurs récents. Elle permet un échange bilatéral synchrone entre le client et le serveur.

Le Web a toujours été conçu comme suit : le client demande et le serveur répond.



C'était suffisant aux débuts du Web, mais c'est devenu trop limitant ces derniers temps. On a besoin d'une communication plus réactive et immédiate. Dans ce schéma par exemple, le serveur ne peut pas décider de lui-même d'envoyer quelque chose au client (par exemple pour l'avertir "il y a un nouveau message !"). Il faut que le client recharge la page ou fasse une action pour solliciter le serveur, car celui-ci n'a pas le droit de s'adresser au client tout seul.

WebSocket est une nouveauté du Web qui permet de laisser une sorte de "tuyau" de communication ouvert entre le client et le serveur. Le navigateur et le serveur restent connectés entre eux et peuvent s'échanger des messages dans un sens comme dans l'autre dans ce tuyau. Le serveur peut donc lui-même décider d'envoyer un message au client.

socket.io nous permet d'utiliser les WebSockets très facilement. Et, comme tous les navigateurs ne gèrent pas WebSocket, il est capable d'utiliser d'autres techniques de communication synchrones si elles sont gérées par le navigateur du client. Sur le site de socket.io, la section "Browser support", détermine pour chaque client quelle est la méthode de communication temps réel la plus adaptée pour le client :

- WebSocket
- Adobe Flash Socket
- AJAX long polling
- AJAX multipart streaming
- Forever Iframe
- JSONP Polling

Grâce à toutes ces différentes techniques de communication, socket.io supporte un très grand nombre de navigateurs, même des anciens.

2.2. Installer socket.io

Socket.io s'installe très facilement en utilisant NPM comme suivant :

```
npm install socket.io
```

Après cette installation vous pouvez commencer à utiliser socket.io

2.3. Connexion d'un client

Dans une application avec socket.io, on doit toujours s'occuper de deux fichiers en même temps:

- **Le fichier serveur (ex : app.js) :** c'est lui qui centralise et gère les connexions des différents clients connectés au site.

- **Le fichier client (ex : index.html) :** c'est lui qui se connecte au serveur et qui affiche les résultats dans le navigateur.

Exemple : Création d'une application qui envoie un message « Bonjour » au client.

- **Le serveur(app.js) :** Dans le fichier serveur, on charge le serveur (et on récupère et renvoie le contenu de la page index.html) ; ensuite, on charge socket.io et on gère les évènements de socket.io.

```
var http = require('http');
var fs = require('fs');

// Chargement du fichier index.html affiché au client
var server = http.createServer(function(req, res) {
    fs.readFile('./index.html', 'utf-8', function(error, content) {
        res.writeHead(200, {"Content-Type": "text/html"});
        res.end(content);
    });
});

// Chargement de socket.io
var io = require('socket.io').listen(server);

// Quand un client se connecte, on le note dans la console
io.sockets.on('connection', function (socket) {
    console.log('Un client est connecté !');
});


server.listen(8080);
```

Ce code **renvoie le fichier index.html** quand un client demande à charger la page dans son navigateur. Puis se prépare à recevoir des requêtes via socket.io. Ici, on s'attend à recevoir un seul type de message : la connexion. Lorsqu'on se connecte via socket.io, on affiche l'information dans la console.

- **Le client (index.html) :** Lorsque le client via son navigateur accède à l'adresse sur laquelle tourne le serveur(app.js), lui envoie le fichier index.html. Dans ce fichier se trouve un code JavaScript qui se connecte au serveur, cette fois pas en http mais via socket.io (donc via les WebSockets en général). Le client effectue donc 2 types de connexion :

- ✓ **Une connexion "classique"** au serveur en HTTP pour charger la page index.html
- ✓ **Une connexion "temps réel"** pour ouvrir un tunnel via les **WebSockets** grâce à socket.io

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Socket.io</title>
    </head>

    <body>
        <h1>Communication avec socket.io !</h1>

        <script src="/socket.io/socket.io.js"></script>
        <script>
            var socket = io.connect('http://localhost:8080');
        </script>
    </body>
</html>
```

Dans un premier temps, on fait récupérer au client le fichier socket.io.js. Celui-ci est automatiquement fourni par le serveur node.js via le module socket.io (le chemin vers le fichier est donc `/socket.io/socket.io.js`) :

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

Ce code permet de gérer la communication avec le serveur du côté du client, soit avec les WebSockets, soit avec l'une des autres méthodes dont je vous ai parlé si le navigateur ne les supporte pas.

Ensuite, nous pouvons effectuer des actions du côté du client pour communiquer avec le serveur se tournant sur l'adresse **localhost:8080**

```
var socket = io.connect('http://localhost:8080');
```

2.4. Envoi et réception de messages

Une fois que le client est connecté, on peut échanger des messages entre le client et le serveur. Il y a 2 cas de figure :

- Le serveur veut envoyer un message au client

- Le client veut envoyer un message au serveur

2.4.1. Envoi de message au client par le serveur

Lorsqu'on détecte une connexion, on émet un message au client avec la fonction **socket.emit().** La fonction prend 2 paramètres :

- **Le type de message** qu'on veut transmettre. Cela vous permettra de distinguer les différents types de message. Par exemple dans un jeu, on pourrait envoyer des messages de type **"deplacement_joueur", "attaque_joueur"...**
- Le contenu du message. Là vous pouvez transmettre ce que vous voulez.

```
io.sockets.on('connection', function (socket) {
        socket.emit('message', 'Vous êtes bien connecté !');
});
```

Du côté du fichier **index.html (le client),** on va écouter l'arrivée de messages de type **"message"**:

```
<script>
    var socket = io.connect('http://localhost:8080');
    socket.on('message', function(message) {
        alert('Le serveur a un message pour vous : ' + message);
    })
</script>
```

Avec **socket.on(),** on écoute les messages de type **"message".** Lorsque des messages arrivent, on appelle la fonction de callback qui, ici, affiche simplement une boîte de dialogue.

2.4.2. Envoi de message au serveur par le client

Du coté client, on peut emettre des message en utilisant la fonction **socket.emit().** La fonction prend aussi 2 paramètres:

- **Le type de message** qu'on veut transmettre. Cela vous permettra de distinguer les différents types de message. Par exemple dans un jeu, on pourrait envoyer des messages de type **"deplacement_joueur", "attaque_joueur"...**
- Le contenu du message. Là vous pouvez transmettre ce que vous voulez.

```
socket.emit('message', 'Vous êtes bien connecté !');
```

Du côté du serveur **app.js** on va écouter l'arrivée de messages de type **"message"**:

```
io.sockets.on('connection', function (socket) {
    socket.emit('message', 'Vous êtes bien connecté !');

    // Quand le serveur reçoit un signal de type "message" du client
    socket.on('message', function (message) {
        console.log('Un client me parle ! Il me dit : ' + message);
    });
});
```

2.5. Communiquer avec plusieurs clients

Socket.io permet de diffuser un message à plusieurs clients. Par exemple dans une application de tchat le serveur aura besoins de transférer tous les message reçu sur un client à tous les autres clients connectés sur le tchat.

Quand on a plusieurs clients, il faut être capable :

- D'envoyer des messages à tout le monde d'un seul coup. On appelle ça les **broadcasts.**
- De se souvenir d'informations sur chaque client (comme son pseudo par exemple). On a besoin pour ça de **variables de session.**

La méthode **socket.emit()** du côté du serveur, envoie uniquement un message au client avec qui on est en train de discuter. Mais on peut envoyer un broadcast, c'est-à-dire un message destiné à tous les autres clients (excepté celui qui vient de solliciter le serveur) en utilisant la méthode **socket.broadcast.emit().**

Syntaxe : socket.broadcast.emit('message', 'Message à toutes les unités. Je répète, message à toutes les unités.')

Exemple :

```
io.sockets.on('connection', function (socket) {
    socket.emit('message', 'Vous êtes bien connecté !');
    // Pour notifier les autre client pa presence d'un nouveau client
    socket.broadcast.emit('message', 'Un autre client vient de se
connecter !');

    // Quand le serveur reçoit un signal de type "message" du client
    socket.on('message', function (message) {
        console.log('Un client me parle ! Il me dit : ' + message);
    });
});
```

Exemple complet : Une petite application qui permet au serveur de diffuser tous les message d'un client aux autres clients

Le client : index.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Socket.io</title>
    </head>
    <body>
        <h1>Communication avec socket.io !</h1>
        <p><input type="button" value="Bonjour serveur" id="envoi" /></p>

        <script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
        <script src="/socket.io/socket.io.js"></script>
        <script>
            var socket = io.connect('http://localhost:8080');
            // On demande le pseudo au visiteur...
            var pseudo = prompt('Quel est votre pseudo ?');
            // Et on l'envoie avec le signal "petit_nouveau" (pour le différencier de
"message")
            socket.emit('petit_nouveau', pseudo);

            // On affiche une boîte de dialogue quand le serveur nous envoie un
"message"
            socket.on('message', function(message) {
                alert('Le serveur a un message pour vous : ' + message);
            })

            // Lorsqu'on clique sur le bouton, on envoie un "message" au serveur
            $('#envoi').click(function () {
                socket.emit('message', 'Salut serveur, ça va ?');
            })
        </script>
    </body>
</html>
```

Le serveur app.js :

```
var http = require('http');
var fs = require('fs');

// Chargement du fichier index.html affiché au client
var server = http.createServer(function(req, res) {
    fs.readFile('./index.html', 'utf-8', function(error, content) {
        res.writeHead(200, {"Content-Type": "text/html"});
        res.end(content);
    });
});

// Chargement de socket.io
var io = require('socket.io').listen(server);

io.sockets.on('connection', function (socket, pseudo) {
    // Quand un client se connecte, on lui envoie un message
    socket.emit('message', 'Vous êtes bien connecté !');
    // On signale aux autres clients qu'il y a un nouveau venu
    socket.broadcast.emit('message', 'Un autre client vient de se connecter !
');

    // Dès qu'on nous donne un pseudo, on le stocke en variable de session
    socket.on('petit_nouveau', function(pseudo) {
        socket.pseudo = pseudo;
    });

    // Dès qu'on reçoit un "message" (clic sur le bouton), on le note dans la
console
    socket.on('message', function (message) {
        // On récupère le pseudo de celui qui a cliqué dans les variables de
session
        console.log(socket.pseudo + ' me parle ! Il me dit : ' + message);
    });
});


server.listen(8080);
```

Références

https://blog.bini.io/developper-une-application-avec-socket-io/

http://expressjs.com/fr/guide/using-middleware.html

https://openclassrooms.com/fr/courses/1056721-des-applications-ultra-rapides-avec-node-js/1057503-le-framework-express-js

https://www.grafikart.fr/formations/nodejs/nodejs-intro

https://makina-corpus.com/blog/metier/2014/introduction-a-nodejs

https://blog.lesieur.name/installer-et-utiliser-nodejs-sous-windows/

http://wdi.supelec.fr/appliouaibe/Cours/JSserveur