Le Protocole WebSocket et Socket.io : Fondamentaux et Avancé

Sommaire

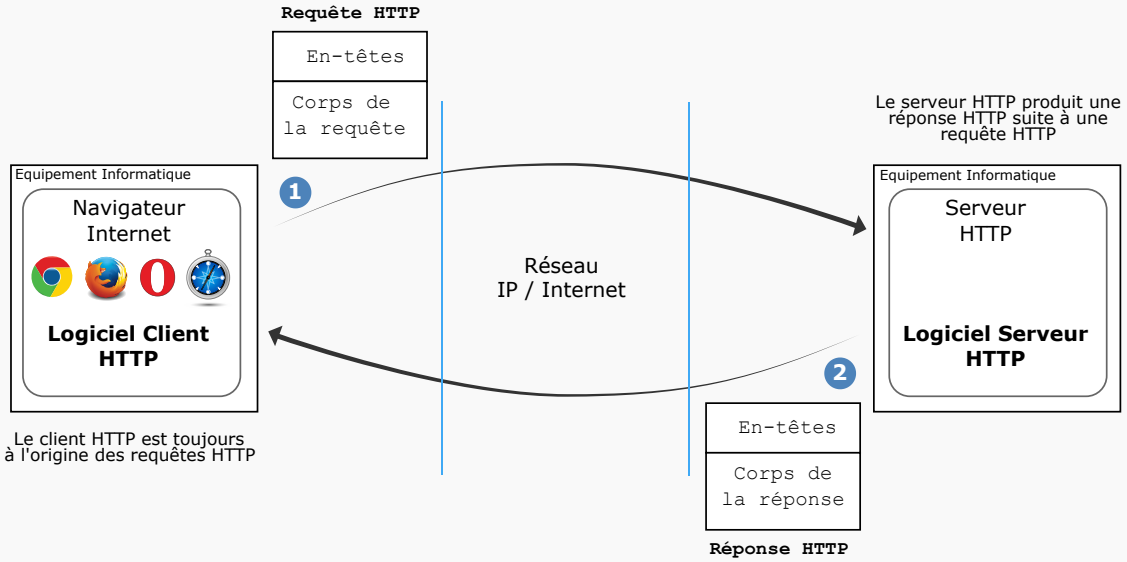
1. WebSocket

1. **Notion de WebSocket**
2. **Le protocole WebSocket**
3. **Client WebSocket en HTML5/JS & serveur WebSocket sur Node JS**

1.1. Notion de WebSocket

Le protocole HTTP propose d'échanger des données à l'initiative du client HTTP. C'est toujours le client HTTP qui initie une requête HTTP et qui reçoit une réponse HTTP en retour de la part du serveur HTTP. Le serveur HTTP est **passif**, il est perpétuellement en attente d'une nouvelle requête HTTP. A aucun moment le serveur HTTP n'est à l'origine d'un échange de données. Entre chaque échange HTTP, la connexion entre client et serveur est coupée.

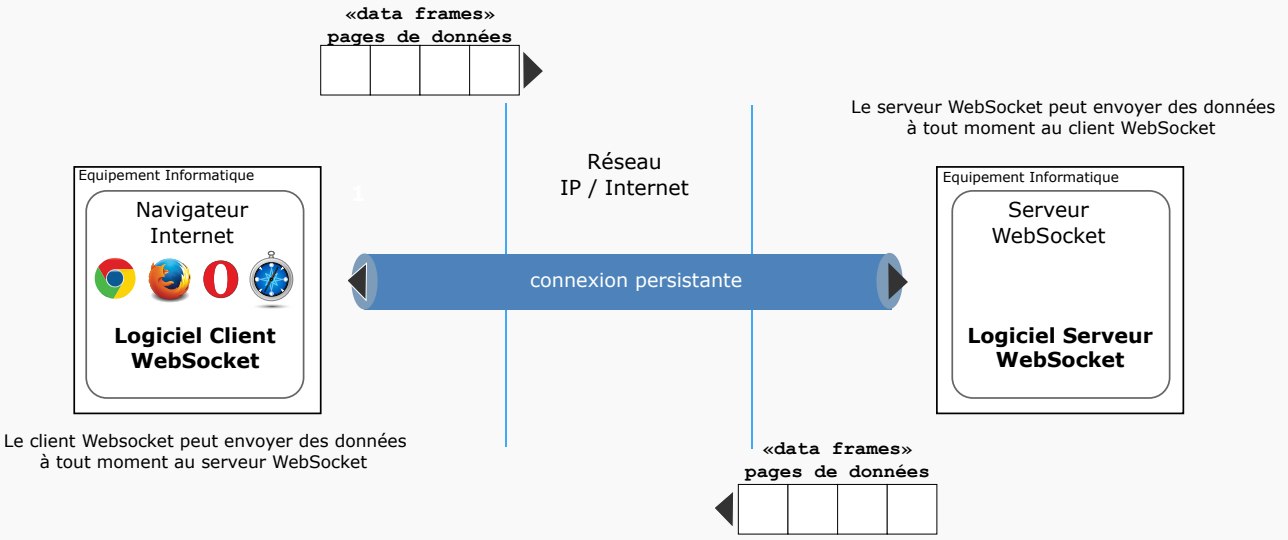
Le mécanisme du HTTP pourrait se schématiser comme suit :



Contrairement au HTTP, le **WebSocket** est un protocole réseau **en cours** de standardisation et proposé par la **RFC 6455** de l'IETF.

Le protocole WebSocket vise à normaliser les échanges bidirectionnels de données entre les clients WebSocket et le serveur WebSocket. Il propose d'établir un canal de communication persistant entre des clients WebSocket et un serveur WebSocket. A tout moment un client WebSocket peut envoyer des données au serveur WebSocket et surtout, à tout moment, **un serveur WebSocket peut envoyer des données à ses clients**. Les données sont transmises de façon fragmentée sous la forme de pages de données (en anglais, « **data frames** »)

Le mécanisme du WebSocket pourrait se schématiser comme suit :



Cette connexion (persistante) est appelée «**socket**» en anglais.

1.2. Le protocole WebSocket

Le protocole WebSocket ressemble au protocole HTTP, au moins lors de la connexion initiale, de telle sorte qu'il soit plus facile de créer un serveur WebSocket à partir d'un serveur HTTP. On notera cependant que les URLs destinés à communiquer en utilisant le protocole WebSocket commencent par le préfixe `ws` pour les échanges non sécurisés et `wss` pour les échanges sécurisés. Par exemple :

- Pour initier un échange en HTTP, on écrirait un URL sous la forme : `http://[hôte][:port][/ressource]` ou `https://[hôte][:port][/ressource]`
- Alors que pour initier un échange en WebSocket on écrirait un URL sous la forme : `ws://[hôte][:port][/ressource]` ou `wss://[hôte][:port][/ressource]`

Intéressons nous maintenant aux étapes nécessaires à l'échange de données en utilisant le protocole WebSocket.

Etape 1 : Établissement de la connexion

Pour établir une connexion basée sur le protocole WebSocket, le client WebSocket doit envoyer une requête de connexion WebSocket (en anglais, « **request handshake** »). En retour, le serveur WebSocket envoie une réponse de connexion WebSocket (en anglais, « **response handshake** »).

- Exemple de requête de connexion WebSocket émise par un client WebSocket :

```
GET / HTTP/1.1
Connection: keep-alive, Upgrade
Sec-WebSocket-Extensions : permessage-deflate
Sec-WebSocket-Key: fs2lqKN4CTkP75Otb9EqXA==
Sec-WebSocket-Version: 13
Upgrade: websocket
```

- Exemple de réponse de connexion WebSocket retournée par un serveur WebSocket:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: zSxqr2CKwpodKyPq4qnuJV5hWeE=
```

L'en-tête de requête « `Sec-WebSocket-Key` » produite par le client WebSocket et l'en-tête de réponse « `Sec-WebSocket-Accept` » produite par le serveur WebSocket sont intéressantes puisque leur

rôle est de permettre aux deux interlocuteurs de s'assurer qu'ils sont bien en mesure de dialoguer en utilisant le protocole WebSocket.

Coté client WebSocket, l'en-tête de requête « `Sec-WebSocket-Key` » est l'encodage, en utilisant l'algorithme Base64, de 16 octets aléatoires.

On pourrait proposer la formule suivante : `Sec-WebSocket-Key` = `Base64`(16 Octets Aléatoires).

Coté serveur WebSocket, l'en-tête de réponse « `Sec-WebSocket-Accept` » doit être généré comme suit :

- On concatène la valeur de l'en-tête de requête « `Sec-WebSocket-Key` » avec le GUID (**G**lobally **U**nique **I**dentifier - Identifiant globalement unique) « `258EAFA5-E914-47DA-95CA-C5AB0DC85B11` »;
- La chaîne de caractère obtenue doit être ensuite encodée en utilisant l'algorithme de cryptage SHA-1;
- Et enfin, la chaîne de caractère cryptée obtenue doit être encodée en utilisant l'algorithme d'encodage Base64.

On pourrait proposer la formule suivante : `Sec-WebSocket-Accept` = `Base64`(`SHA1`(`Sec-WebSocket-Key` + `258EAFA5-E914-47DA-95CA-C5AB0DC85B11`)).

Etape 2 : Échanger des messages

Un fois l'échange initial effectué, le client WebSocket et le serveur WebSocket peuvent échanger à tout moment des messages. Ces messages sont composés de pages de données (en anglais, « **data frames** »). Les pages de données sont des fragments du message. Toutes les pages de données doivent être obtenues par le destinataire pour pouvoir reconstituer le message entier.

Les pages de données sont codées au minimum sur 2 octets (16 bits) et au maximum sur 2^{63} octets (9223372036854776000 bits). Cependant, du point de vue du protocole un message peut être de taille infinie puisqu'il peut être constitué de 1 ou plusieurs pages de données. Le détails des bits correspondant à une page de données et leur signification est expliquée dans la RFC 6455 **ici**.

Lorsqu'un client WebSocket envoi des page de données à un serveur WebSocket celles-ci sont obligatoirement (pour des raison de sécurité) masquées et doivent être décodées par le serveur WebSocket.

En résumé, pour qu'un destinataire puisse recevoir et exploiter un message se conformant au protocole WebSocket, il doit :

- Recevoir toutes les pages de données;
- Décoder toutes les pages de données;
- Reconstituer le message.

Ces dernières étapes de l'implémentation du protocole WebSocket sont assez complexe à réaliser lors de la création d'un serveur WebSocket. C'est pourquoi nous ne développerons pas directement un serveur WebSocket mais nous utiliserons des serveurs WebSocket fournis sous la forme de modules complémentaires de la plateforme Node JS. Cependant, nous étudierons quand même une implémentation basique de l'établissement d'une connexion WebSocket entre un client WebSocket simple et un serveur WebSocket simple sur Node JS.

1.3. client WebSocket en HTML5/JS & serveur WebSocket sur Node JS

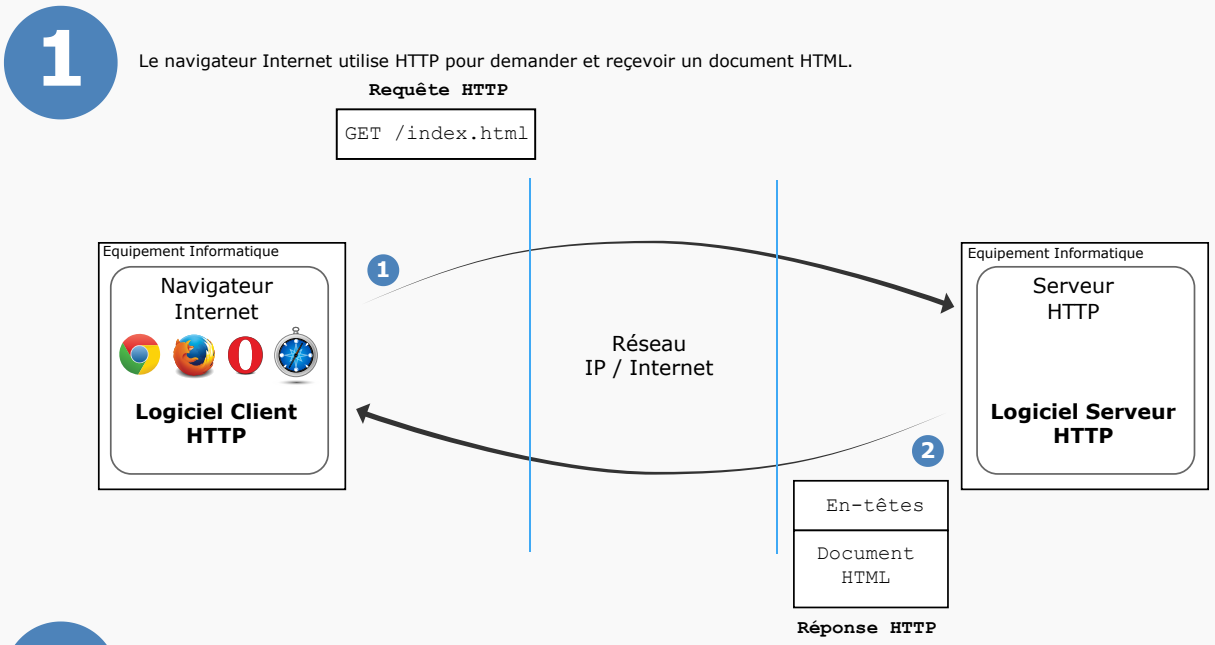
Les navigateurs Internet sont programmés, par défaut, pour échanger en respectant le protocole HTTP. Cependant, l'API de JavaScript nous propose d'utiliser l'objet `WebSocket`. Cet objet est disponible et respecte le protocole RFC6455 du WebSocket à partir de :

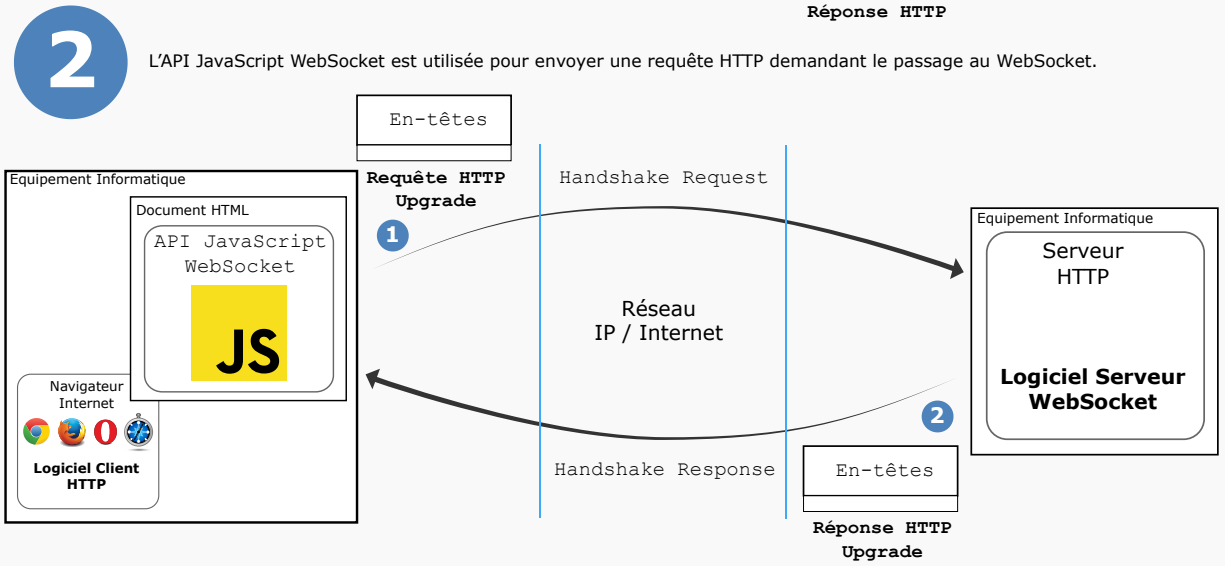
- Internet Explorer 10 et supérieur;
- Mozilla Firefox 11 et supérieur;
- Google Chrome 16 et supérieur;

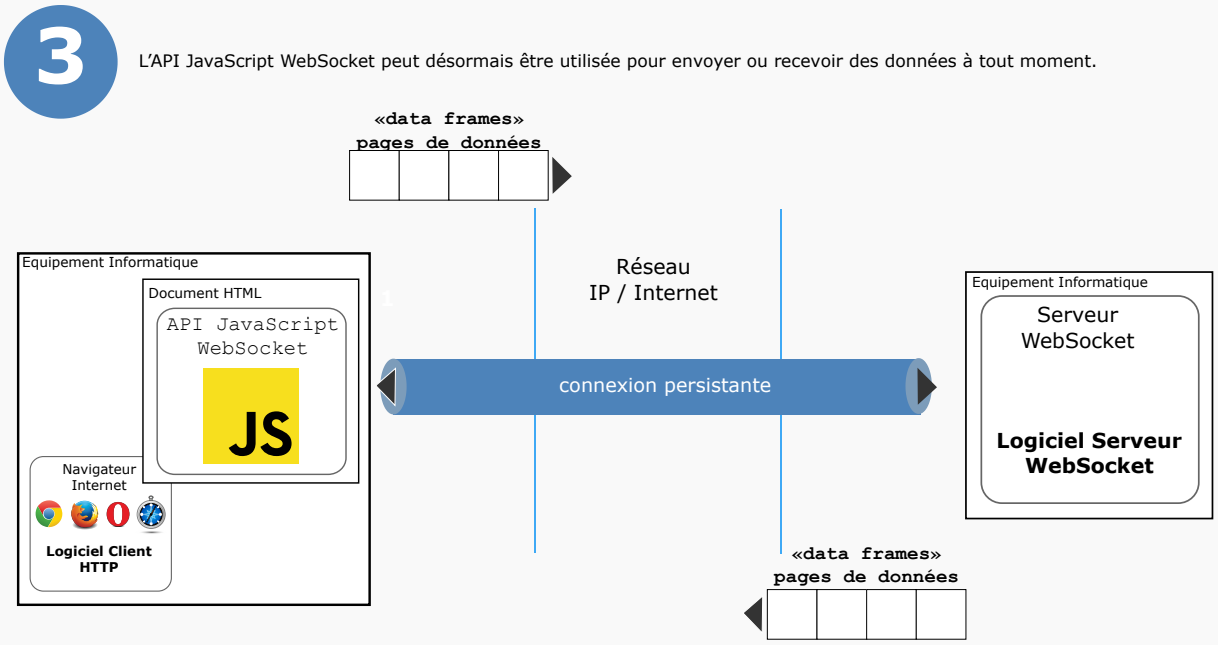
- Safari 6 et supérieur;
- Opera 12 et supérieur.

La documentation officielle de cet objet est disponible **ici** sur le MDN.

Il s'agira donc, pour créer un client WebSocket, de créer un document HTML qui sera téléchargé selon le protocole HTTP par le navigateur Internet et, au sein de ce document HTML, d'utiliser l'objet WebSocket fourni par l'API de JavaScript pour ouvrir un canal de communication vers un serveur WebSocket. On pourrait schématiser cela en 3 temps comme suit :







Le code ci-après est celui d'un serveur HTTP et WebSocket fonctionnant à l'aide de Node JS. La partie HTTP produit une réponse qui est un fichier HTML détaillé plus loin. Le serveur WebSocket présenté ici en exemple est rudimentaire :

- Il n'envoie aucune données au(x) client(s) WebSocket connectés;
- II ne fonctionne que si il est sollicité depuis l'API JavaScript de Mozilla Firefox;
- Il fonctionne uniquement lorsque les messages envoyés via le protocole WebSocket sont fragmentés en 1 seule page de données.

JavaScript

```
/**
  Le Serveur HTTP.
  URL : http://[adresse IP/nom de domaine]:8888/

  Ce serveur produit une réponse HTTP contenant un document
  HTML suite à une requête HTTP provenant d'un client HTTP.
**/

// Chargement du module HTTP.
const http = require('http');

// Création du serveur HTTP.
var httpServer = http.createServer();

// Fonction qui produit la réponse HTTP.
var writeResponse = function writeHTTPResponse(HTTPResponse, responseCode, responseBody)
    HTTPResponse.writeHead(responseCode, {
      'Content-type':'text/html; charset=UTF-8',
      'Content-length':responseBody.length
    });
    HTTPResponse.write(responseBody, function(){
      HTTPResponse.end();
    });
};

// Fonction qui produit une réponse HTTP contenant un message d'erreur 404 si le document
var send404NotFound = function(HTTPResponse){
  writeResponse(HTTPResponse, 404, '<doctype html!><html><head>Page Not Found</head><body
};

/**
  Gestion de l'évènement 'request' : correspond à la gestion
  de la requête HTTP initiale permettant d'obtenir le fichier
  HTML contenant le code JavaScript utilisant l'API WebSocket.
**/
httpServer.on('request', function(HTTPRequest, HTTPResponse){
    console.log('Événement HTTP \'request\'.');
    var fs = require('fs');
    //le fichier HTML que nous utiliserons dans tous les cas.
    var filename = 'websocket-client.html';
    fs.access(filename, fs.R_OK, function(error){
      if(error){
        send404NotFound(HTTPResponse);
      }else{
        fs.readFile(filename, function(error, fileData){
```

```
            if(error){
              send404NotFound(HTTPResponse);
            }else{
              writeResponse(HTTPResponse, 200, fileData);
            }
        });
      }
    });
});

/**
  Le Serveur WebSocket associé au serveur HTTP.
  URL : ws://[adresse IP/nom de domaine]:8888/

  Ce serveur accepte une requête HTTP upgrade et établit
  une connexion persistante basée sur WebSocket.
**/

/**
  Gestion de l'évènement 'upgrade' : correspond à une demande
  d'établissement de connexion basée sur WebSocket.
**/
httpServer.on('upgrade', function (WebSocketRequest, socket, head) {
    console.log('Événement WebSocket \'upgrade\'.');

    /**
      Création de la 'response handshake' : signale au client
      que le passage au protocole WebSocket est accepté.
    **/
    var secWebsocketKey = WebSocketRequest.headers['sec-websocket-key'];
    var webSocketProtocolGUID = '258EAFA5-E914-47DA-95CA-C5AB0DC85B11';

    var crypto = require('crypto');
    var hash = crypto.createHash('sha1');
    hash.update(secWebsocketKey + webSocketProtocolGUID);
    var secWebSocketAccept = hash.digest('base64');

    var webSocketServerHeaders = [
      'HTTP/1.1 101 Switching Protocols',
      'Upgrade: websocket',
      'Connection: Upgrade',
      'Sec-WebSocket-Protocol: diwjs',
      'Sec-WebSocket-Accept: ' + secWebSocketAccept
    ].concat('', '').join('\r\n') + '\r\n';

    socket.setTimeout(0);
    socket.setNoDelay(true);
    socket.setKeepAlive(true);

    // Envoi de la 'response handshake' au client.
```

```
    socket.write(webSocketServerHeaders, 'ascii');

    /**
     Gestion l'évènement 'data' : Cet évènement survient lorsque le serveur
     WebSocket reçoit des données en provenance du client WebSocket.
    **/
    socket.on('data',function(chunk){
      //chunk : Variable qui contient 1 "data frame" (page de données).

      /**
        Décodage d'une page de données reçue.

        ATTENTION : ceci ne fonctionne pas si la
        donnée est fragmentée en plusieurs pages.
      **/

      // Position par défaut du masque binaire.
      var maskLocation = 2;
      // Position par défaut du masque binaire.
      var maskingKey = chunk.slice(maskLocation, maskLocation + 4);
      // Position par défaut des données.
      var dataLocation = maskLocation + 4;
      // Tableau pour le stockage des données décodées
      var decodedData = [];
      for (var i = 0; dataLocation < chunk.length; i++) {
        // Décodage octet par octet
        decodedData.push(chunk[dataLocation] ^ maskingKey[i % 4]);
        dataLocation++
      };
      // Création d'un buffer (suite d'octets) à partir du tableau d'octets décodés
      var buffer = new Buffer(decodedData);
      // Conversion du buffer en chaîne de caractères
      var message = buffer.toString();
      // Affichage du message dans la console.
      console.log(message);
    });

});

httpServer.listen(8888);
```

Le code ci-après est celui du fichier HTML contenant le code JavaScript faisant appel à l'API WebSocket du navigateur Internet. Ce fichier est celui fourni en HTTP par le serveur présenté au dessus (*Attention, si vous souhaitez tester ces programmes, pensez à modifier l'adresse IP par la votre*). La documentation officielle de l'API WebSocket de JavaScript pour les navigateurs Internet est disponible **ici.**

Markup

```
<doctype html!>
<html>
  <head>
    <title>HTML5/JS & Websockets</title>
  </head>
  <script type="text/javascript">
    (function(window){

      // Au chargement du document.
      window.addEventListener('DOMContentLoaded',function(){

          // Établissement d'une nouvelle connexion WebSocket vers le serveur WebSocket.
          var webSocketClient = new WebSocket('ws://127.0.0.1:8888/', 'diwjs');
          /**
            Le premier argument est l'URL du serveur WebSocket.
            Le second argument est le "protocole". Cette chaîne de caractère peut
            être utilisée coté serveur pour déterminer le traitement à effectuer.
          **/

          /**
            On attache un gestionnaire d'évènement à l'évènement 'open' qui correspond
            à la fin de l'échange request handshake / response handshake.
          **/
          webSocketClient.addEventListener('open', function(event){

            var HTMLpElement = window.document.getElementById('clickable-element');
            HTMLpElement.addEventListener('click', function(event){

              /**
                A chaque clic de souris sur l'élément HTML considéré
                on envoi un message à travers la connexion WebSocket.
              **/
              webSocketClient.send('Hello !');

            });

            /**
              On attache un gestionnaire d'évènement à l'évènement 'message' qui corresp
              l'événement déclenché lorsqu'un message à été reçu en provenance du serveu
            **/
            webSocketClient.addEventListener('message', function(event){

              /**
                A chaque message reçu, on affiche les données
                obtenues dans la console du navigateur Internet.
              **/
              console.log(event.data);

            });
```

```
        });

      });

    })(window);
  </script>
  <body>
    <h1>HTML5/JS & WebSocket</h1>
    <p id="clickable-element">Envoyer un message à l'aide de WebSocket</p>
  </body>
</html>
```

Le serveur HTTP/WebSocket doit être démarré. Lorsque le fichier HTML est chargé par le navigateur Internet, le script JavaScript tente d'établir une connexion WebSocket avec le serveur. Lorsque ce dernier l'accepte, il est alors possible de cliquer sur un élément du document HTML affiché pour envoyer un message sous la forme d'une page de donnée au serveur. Lorsque le serveur reçoit une page de données, il la décode et l'affiche dans sa console. Si le serveur était également en mesure d'envoyer des données au client, ce dernier les afficherait alors dans la console du navigateur Internet.

L'implémentation d'un serveur Websocket complet est très laborieuse. Elle nécessite de gérer les données qui sont transmises selon une ou plusieurs pages de données et de gérer les divergences qui existent entre l'implémentation du protocole par le différents navigateurs Internet (la forme des messages reçus et envoyés à l'aide du protocole peut varier selon les navigateurs Internet et leurs versions). C'est pourquoi nous allons nous intéresser à des extensions de Node JS qui facilitent le développement d'un serveur basé sur le protocole WebSocket.

2. Extensions

1. **L'extension WebSocket**
2. **L'extension Socket.io**

2.1. L'extension WebSocket

L'extension WebSocket de Node JS est disponible **ici** sur npm. Cette extension permet de créer un serveur WebSocket à partir d'un serveur HTTP existant. Elle s'utilise coté serveur comme n'importe quelle extension de Node JS.

Voici un exemple de serveur WebSocket utilisant l'extension WebSocket :

```JavaScript

/**
  Le Serveur HTTP.
  URL : http://[adresse IP/nom de domaine]:8888/

  Ce serveur produit une réponse HTTP contenant un document
  HTML suite à une requête HTTP provenant d'un client HTTP.
**/

// Chargement du module HTTP.
const http = require('http');

// Création du serveur HTTP.
var httpServer = http.createServer();

// Fonction qui produit la réponse HTTP.
var writeResponse = function writeHTTPResponse(HTTPResponse, responseCode, responseBody)
    HTTPResponse.writeHead(responseCode, {
      'Content-type':'text/html; charset=UTF-8',
      'Content-length':responseBody.length
    });
    HTTPResponse.write(responseBody, function(){
      HTTPResponse.end();
    });
};

// Fonction qui produit une réponse HTTP contenant un message d'erreur 404 si le document
var send404NotFound = function(HTTPResponse){
  writeResponse(HTTPResponse, 404, 'Page Not Found404: Page Not FoundThe requested URL c
};

/**
  Gestion de l'évènement 'request' : correspond à la gestion
  de la requête HTTP initiale permettant d'obtenir le fichier
  HTML contenant le code JavaScript utilisant l'API WebSocket.
**/
httpServer.on('request', function(HTTPRequest, HTTPResponse){
    console.log('Événement HTTP \'request\'.');
```

```
    var fs = require('fs');
    //le fichier HTML que nous utiliserons dans tous les cas.
    var filename = 'websocket-client.html';
    fs.access(filename, fs.R_OK, function(error){
      if(error){
        send404NotFound(HTTPResponse);
      }else{
        fs.readFile(filename, function(error, fileData){
            if(error){
              send404NotFound(HTTPResponse);
            }else{
              writeResponse(HTTPResponse, 200, fileData);
            }
        });
      }
    });
});

/**
  Le Serveur WebSocket associé au serveur HTTP.
  URL : ws://[adresse IP/nom de domaine]:8888/

  Ce serveur accepte une requête HTTP upgrade et établit
  une connexion persistante basée sur WebSocket.
**/

/**
  On installe et on utilise le package websocket.
  La documentation est ici : https://www.npmjs.com/package/websocket
**/
var webSocket = require('websocket');

// On récupére une référence à la classe WebSocketServer.
var WebSocketServer = webSocket.server;
/**
  La classe WebSocketServer est documentée ici :
  https://github.com/theturtle32/WebSocket-Node/blob/master/docs/WebSocketServer.md
**/

// On instancie la classe avec pour argument un référence à notre serveur HTTP.
var webSocketServer = new WebSocketServer({
  httpServer: httpServer
});

/**
  Gestion de l'évènement 'request' : correspond à la gestion
  d'une requête WebSocket provenant d'un client WebSocket.
**/
webSocketServer.on('request', function(webSocketRequest){
  console.log('Événement WebSocket \'request\'.');
```

```
  webSocketRequest;
  /**
    Objet de type WebSocketRequest documenté ici :
    https://github.com/theturtle32/WebSocket-Node/blob/master/docs/WebSocketRequest.md
  **/

  var acceptedProtocol = 'diwjs';
  var allowedOrigin = webSocketRequest.origin;

  /**
    La méthode .accept() prend en argument :
      - le nom du protocole autorisé pour les clients du serveur WebSocket ;
      - l'origine autorisée des requêtes.
    La méthode .accept() retourne un objet de type WebSocketConnection documenté ici : ht
  **/

  var webSocketConnection = webSocketRequest.accept(acceptedProtocol, allowedOrigin);
  /**
    A titre indicatif, coté client, l'utilisation de l'API WebSocket
    devra être utilisée de la façon suivante :
    var webSocketClient = new WebSocket('ws://[adresse IP/nom de domaine]:8888/', 'diwjs
  **/

  /**
    Gestion de l'évènement 'message' : correspond à la gestion des messages
    reçus par le serveur WebSocket en provenance du client WebSocket.
  **/
  webSocketConnection.on('message', function(message){
    /**
      La variable message est un objet de la forme :
        - Si le message est en UTF-8 : {type:'utf8', utf8Data:'le message reçu'}
        - Si le message est en binaire :  {type:'binary', utf8Data: bufferDeDonneesBinai
    **/

    // Affichage du message reçu dans la console.
    console.log(message);

    // Envoi d'un message au client WebSocket.
    webSocketConnection.sendUTF('Message bien reçu !');

  });

});

httpServer.listen(8888);
```

Et le fichier HTML contenant le code JavaScript du client WebSocket (il s'agit du même fichier HTML que celui présenté précédemment) :

```Markup
<doctype html!>
<html>
  <head>
    <title>HTML5/JS & Websockets</title>
  </head>
  <script type="text/javascript">
    (function(window){

      //Au chargement du document
      window.addEventListener('DOMContentLoaded',function(){

          // Établissement d'une nouvelle connexion WebSocket vers le serveur WebSocket.
          var webSocketClient = new WebSocket('ws://[adresse IP/nom de domaine]:8888/',
          /**
            Le premier argument est l'URL du serveur WebSocket.
            Le second argument est le "protocole". Cette chaîne de caractère peut
            être utilisée coté serveur pour déterminer le traitement à effectuer.
          **/

          /**
            On attache un gestionnaire d'évènement à l'évènement 'open' qui correspond
            à la fin de l'échange request handshake / response handshake.
          **/
          webSocketClient.addEventListener('open', function(event){

            var HTMLpElement = window.document.getElementById('clickable-element');
            HTMLpElement.addEventListener('click', function(event){

              /**
                A chaque clic de souris sur l'élément HTML considéré
                on envoi un message à travers la connexion WebSocket.
              **/
              webSocketClient.send('Hello !');

            });

            /**
              On attache un gestionnaire d'évènement à l'évènement 'message' qui corresp
              l'événement déclenché lorsqu'un message à été reçu en provenance du serveu
            **/
            webSocketClient.addEventListener('message', function(event){

              /**
                A chaque message reçu, on affiche les données
                obtenues dans la console du navigateur Internet.
              **/
              console.log(event.data);
```

```
            });

          });

      });

    })(window);
  </script>
  <body>
    <h1>HTML5/JS & WebSocket</h1>
    <p id="clickable-element">Envoyer un message à l'aide de WebSocket</p>
  </body>
</html>
```

2.2. L'extension Socket.io

L'extension Socket.io de Node JS est disponible **ici** sur npm. La documentation officielle du module est accessible **ici**. Cette extension peut être utilisée pour :

- Créer un serveur WebSocket à partir d'un serveur HTTP;
- Créer un serveur WebSocket à partir d'un serveur HTTP basé sur l'extension Express JS;
- Créer un serveur WebSocket pur.

Des exemples d'utilisation de Socket.io pour créer ces 3 types de serveurs sont disponibles **ici**.

La particularité de Socket.io réside dans le fait que ce « framework » propose une couche d'abstraction pour le serveur sous la forme d'un module Node JS et d'une couche d'abstraction pour le client sous la forme d'un fichier .js à utiliser avec son script JavaScript coté client. En d'autres termes, Socket.io se compose d'un :

- Module coté serveur à utiliser avec Node JS documenté **ici**;
- Framework coté client à utiliser avec son script JavaScript au sein du navigateur Internet documenté **ici**.

Voici un exemple de serveur WebSocket utilisant l'extension Socket.io :

JavaScript

```
/**
  Le Serveur HTTP.
  URL : http://[adresse IP/nom de domaine]:8888/

  Ce serveur produit une réponse HTTP contenant un document
  HTML suite à une requête HTTP provenant d'un client HTTP.
**/

// Chargement du module HTTP.
const http = require('http');

// Création du serveur HTTP.
var httpServer = http.createServer();
```

```
// Fonction qui produit la réponse HTTP.
var writeResponse = function writeHTTPResponse(HTTPResponse, responseCode, responseBody)
    HTTPResponse.writeHead(responseCode, {
      'Content-type':'text/html; charset=UTF-8',
      'Content-length':responseBody.length
    });
    HTTPResponse.write(responseBody, function(){
      HTTPResponse.end();
    });
};

// Fonction qui produit une réponse HTTP contenant un message d'erreur 404 si le document
var send404NotFound = function(HTTPResponse){
  writeResponse(HTTPResponse, 404, 'Page Not Found404: Page Not FoundThe requested URL c
};

/**
  Gestion de l'évènement 'request' : correspond à la gestion
  de la requête HTTP initiale permettant d'obtenir le fichier
  HTML contenant le code JavaScript utilisant l'API WebSocket.
**/
httpServer.on('request', function(HTTPRequest, HTTPResponse){
    console.log('Événement HTTP \'request\'.');
    var fs = require('fs');
    //le fichier HTML que nous utiliserons dans tous les cas.
    var filename = 'websocket-client.html';
    fs.access(filename, fs.R_OK, function(error){
      if(error){
        send404NotFound(HTTPResponse);
      }else{
        fs.readFile(filename, function(error, fileData){
            if(error){
              send404NotFound(HTTPResponse);
            }else{
              writeResponse(HTTPResponse, 200, fileData);
            }
        });
      }
    });
});

/**
  Le Serveur WebSocket associé au serveur HTTP.
  URL : ws://[adresse IP/nom de domaine]:8888/

  Ce serveur accepte une requête HTTP upgrade et établit
  une connexion persistante basée sur WebSocket.
**/

/**
```

```
  On installe et on utilise le package socket.io.
  La documentation est ici :
  - https://www.npmjs.com/package/socket.io
  - https://github.com/socketio/socket.io
  - http://socket.io/
**/
var socketIO = require('socket.io');

//  On utilise utilise la fonction obtenue avec notre serveur HTTP.
var socketIOWebSocketServer = socketIO(httpServer);

/**
  Gestion de l'évènement 'connection' : correspond à la gestion
  d'une requête WebSocket provenant d'un client WebSocket.
**/
socketIOWebSocketServer.on('connection', function (socket) {

  // socket : Est un objet qui représente la connexion WebSocket établie entre le client

  /**
    On attache un gestionnaire d'évènement à un évènement personnalisé 'unEvenement'
    qui correspond à un événement déclaré coté client qui est déclenché lorsqu'un message
    a été reçu en provenance du client WebSocket.
  **/
  socket.on('unEvenement', function (message) {

    // Affichage du message reçu dans la console.
    console.log(message);

    // Envoi d'un message au client WebSocket.
    socket.emit('unAutreEvenement', {texte: 'Message bien reçu !'});
    /**
      On déclare un évènement personnalisé 'unAutreEvenement'
      dont la réception sera gérée coté client.
    **/

  });

});

httpServer.listen(8888);
```

Et le fichier HTML contenant le code JavaScript du client WebSocket utilisant le « framework » socket.io :

```Markup
<doctype html!>
<html>
```

```
<head>
  <title>HTML5/JS & Websockets</title>
</head>
<!-- Chargement du "framework" client socket.io -->
<script type="text/javascript" src="/socket.io/socket.io.js"></script>
<!--
  Coté serveur, si on utilise le module socket.io, le serveur se voit automatiquement a
  gestionnaire d'événement pour la route : /socket.io/socket.io.js ( le fichier étant r
  situé coté serveur dans le dossier : \node_modules\socket.io\node_modules\socket.io-c
-->
<script type="text/javascript">
  (function(window, io){

    //Au chargement du document
    window.addEventListener('DOMContentLoaded',function(){

      /**
        Établissement d'une nouvelle connexion WebSocket vers le serveur
        WebSocket à l'aide de la fonction io fournie par le "framework"
        client socket.io.
      **/
      var socket = io('http://127.0.0.1:8888/');

      // socket : Est un objet qui représente la connexion WebSocket établie entre le c

      var HTMLpElement = window.document.getElementById('clickable-element');
      HTMLpElement.addEventListener('click', function(event){

        /**
          A chaque clic de souris sur l'élément HTML considéré
          on envoi un message à travers la connexion WebSocket.
        **/
        socket.emit('unEvenement', { texte: 'Hello !' });
        /**
          On déclare un évènement personnalisé 'unEvenement' dont
          la réception sera gérée coté serveur.
        **/

      });

      /**
        On attache un gestionnaire d'évènement à un évènement personnalisé 'unAutreEver
        qui correspond à un événement déclaré coté serveur qui est déclenché lorsqu'un
        a été reçu en provenance du serveur WebSocket.
      **/
      socket.on('unAutreEvenement', function (data) {

        /**
          A chaque message reçu, on affiche les données
          obtenues dans la console du navigateur Internet.
```

```
            **/
            console.log(data);
          });

        });

    })(window, io);
  </script>
  <body>
    <h1>HTML5/JS & WebSocket</h1>
    <p id="clickable-element">Envoyer un message à l'aide de WebSocket</p>
  </body>
</html>
```