



1 Sommaire

1	Sommaire	1
2	Objectifs.....	1
3	Avertissements.....	2
4	Gestion de version avec Git et GitHub	2
4.1	Rappels :.....	2
4.2	Consignes pour le TP :.....	2
5	Ressources.....	2
6	Node.js.....	3
7	Création d'un projet Node.js avec Git sous Visual Studio	3
8	Les bonnes pratiques de codage	7
8.1	Utilisation du mode strict	7
8.2	Utilisation d'un Linter	7
9	Réalisation	9
9.1	Créer le serveur Web en Node.js	9
9.2	Créer une WebSocket « echo ».....	10
9.3	Application texte chat	13
9.4	Jeux questions/réponses	14
9.4.1	Utilisation de fonctions.....	15
9.4.2	Question conversion base 2	16
9.4.3	Utilisation d'une classe	17
9.4.4	Transmission d'information sérialisée en JSON	19
9.4.5	Gestion des joueurs	20
9.4.6	Créer un module pour la classe CQr.....	22

2 Objectifs

- Initiation au langage JavaScript
- Programmation sur navigateur et serveur Node.js
- Programmation avec un gestionnaire de version Git

3 Avertissements

- Tous les fichiers sources seront soigneusement et intelligemment commentés.
- L'évaluation du TP sera basée sur la progression des validations (commits) dans le gestionnaire de version, par la consultation du .git présent dans le BACKUP de votre répertoire de TP et aussi par la consultation de votre dépôt GitHub.

4 Gestion de version avec Git et GitHub

4.1 Rappels :

Un logiciel de gestion de versions (version control system : **VCS**) permet de :

- sauvegarder les fichiers et leurs arborescences en gardant l'historique des modifications
- conserver le code source de différentes versions d'un logiciel

Vous utiliserez dans ce TP le gestionnaire de version décentralisée **Git**.

Le **dépôt ou référentiel (repository)** est un stockage organisé de données. Il peut être :

- **local** (sur votre PC)
- **distant** (pour Git : Azure DevOps, GitHub, ...)

La base de donnée du dépôt (**repository**) est dans le répertoire « .git ». Il contient la sauvegarde de toutes les données.

Les fichiers qui seront sauvegardés et suivis dans leurs versions seront ceux qui se trouveront dans le **répertoire de travail** et qui sont **indexés**.

Votre répertoire de travail est le répertoire de projet VS, vous y trouverez le répertoire « .git »

Pour sauvegarder les fichiers suivis, il faut faire une validation d'un instantané avec la commande **git commit**.

4.2 Consignes pour le TP :

Faire un commit lorsque c'est indiqué dans les questions «  Commit # » suivi d'un numéro.

Le nom du commit commencera par # suivi du numéro indiqué puis de quelques mots pertinents résumant ce qui a été fait.

 Note : Il est très important de bien nommer un commit.

Avant chaque commit, assurez-vous que les modifications sont correctes en effectuant un test (exécution du programme, fonctionnalité validée).

Vous pouvez faire des commits supplémentaires pour des corrections de code (indiquez l'erreur corrigée) ou faire du refactoring de code en précisez la modification (ajout de commentaires, tabulations, changement de nom de variable...).

 N'utiliser pas d'opérations Git sans en connaître les conséquences.

Vous utiliserez dans ce TP les outils Git de Visual Studio.

5 Ressources

- Visual Studio 2022 avec la fonctionnalité Node.js
- ⇒ A lire sur le langage javascript : <https://developer.mozilla.org/fr/Apprendre/JavaScript>

- ⇒ A lire sur Git : <https://git-scm.com/docs>
- ⇒ A lire sur Node.js : <https://nodejs.org/en/about>

6 Node.js

Moteur d'exécution JavaScript événementiel asynchrone.

- Utilise la **machine virtuelle V8**.
- Bien adapté à la création d'une bibliothèque Web ou d'un Framework.
- Comportement similaire à JavaScript du navigateur.
- Conçu pour créer des applications réseau évolutives.
- **N'utilise pas de thread** contrairement au modèle plus commun d'aujourd'hui.
- Pas de risque de blocage : pas de verrous de thread, pas d'accès directe des E/S.
- Plus simple à utiliser surtout pour le développement de systèmes évolutifs.
- Le modèle d'événement présente une **boucle d'événement** comme une construction d'exécution plutôt que comme une bibliothèque.

VS et Node.js : <https://visualstudio.microsoft.com/fr/vs/features/node-js/>

Exemple de code pour un réaliser un serveur web en Node.js (**ne pas faire**) :

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

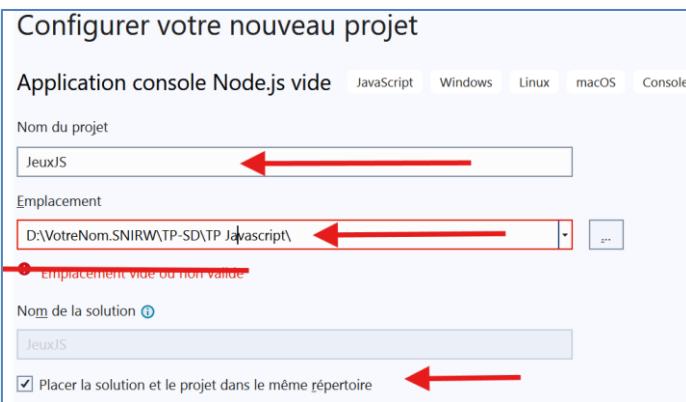
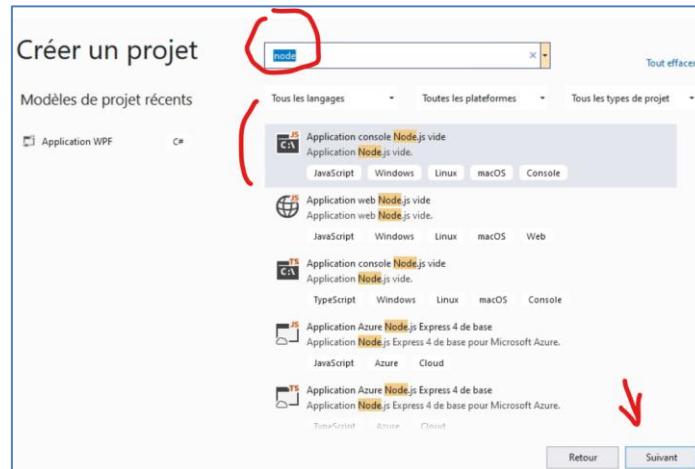
var server = http.createServer(function (req, res) {
    res.writeHead(200);
    res.end('Bonjour !');
});
server.listen(8090);
```

 **Note** : Le gestionnaire de paquets est : **npm** (Package Manager for Node.js modules)

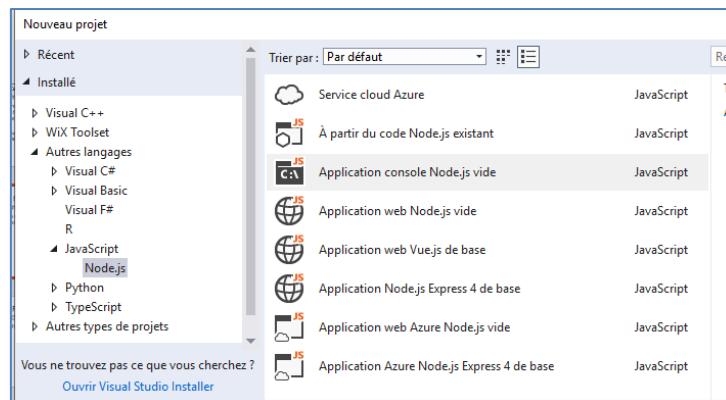
 **Note** : Sous linux pour lancer en demon, utiliser « forever »

7 Crédit d'un projet Node.js avec Git sous Visual Studio

- ⇒ Créer un nouveau projet sur VS dans le répertoire
D:\NomEtudiant.SNIRW\TP- SD\TP Javascript
 - Au lancement de VS :

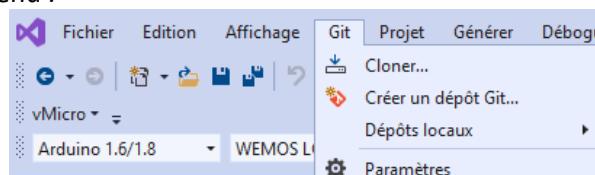


- Ou bien avec le menu nouveau projet :



- ⇒ Créer un dépôt git local :

⇒ Soit par le menu :



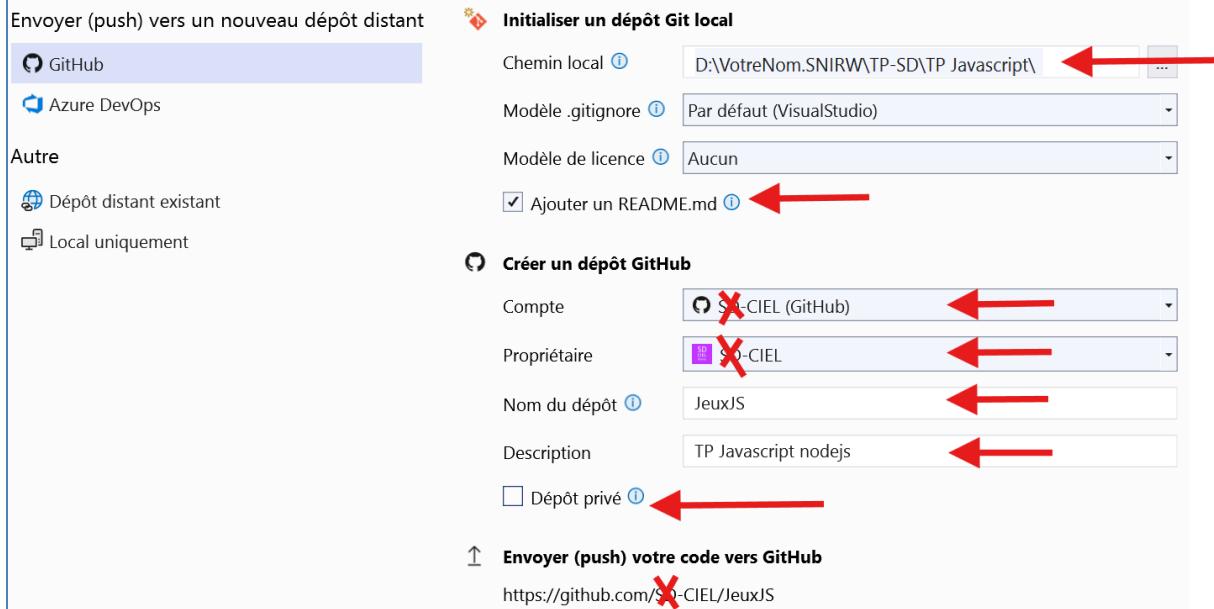
⇒ Soit en bas à droite : ajouter au contrôle de code source/GIT



- ⇒ Vous devez vous connecter sur votre compte GitHub sur le navigateur.

⇒ Configurer comme indiquer ci-dessous :

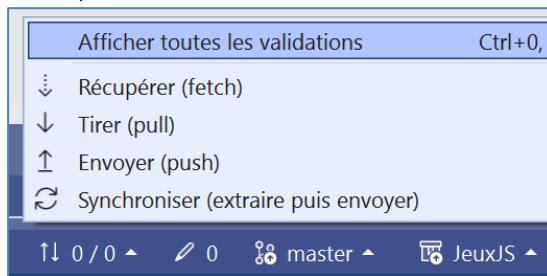
Créer un dépôt Git



- ⇒ Indiquer les éléments qui ont été ajouté dans le répertoire du projet suite à la création du dépôt et préciser leurs fonctions :

[Large empty box for notes]

- ⇒ Relever l'historique de la branche, sélectionner « Afficher toutes les validation » en bas à droite :



[Large empty box for notes]

- ⇒ Renseigner le README.md avec le nom du TP et la date.
 - ⇒ Tester que le script « Hello World » fonctionne.
- Lancer le programme à partir d'une console PowerShell : dans le répertoire du projet VS taper :

```
node .\app.js
```

- ⇒ Changer le texte « Hello World » par « TP CIEL », tester.

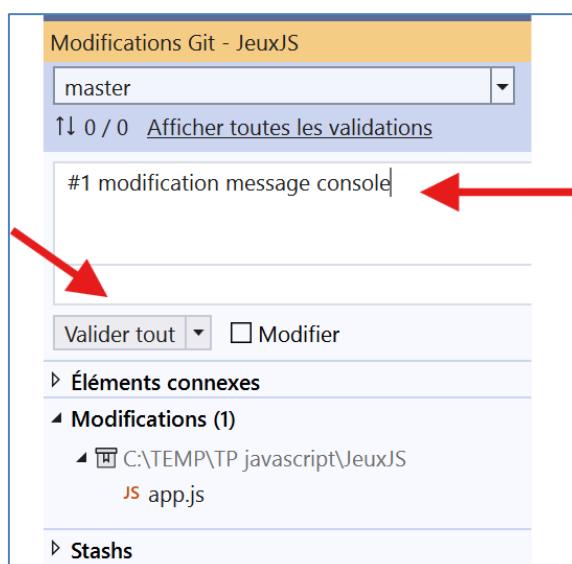
- ⇒ Réaliser votre premier commit :

Commit #1 :

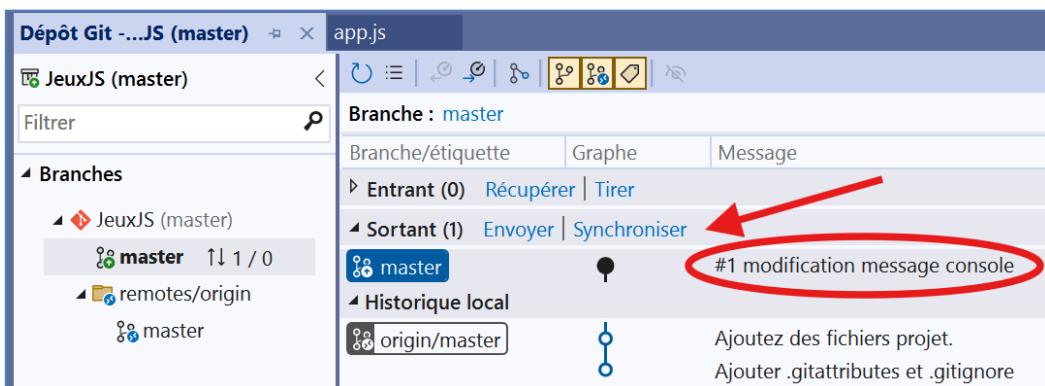
- Cliquer sur



- Dans la fenêtre qui s'ouvre « Modification Git », donner un nom à votre commit puis « Valider tout » :



- ⇒ Relever à nouveau l'historique de la branche et synchroniser avec le dépôt distant :



⇒ Vérifier la mise à jour du dépôt GitHub :

The screenshot shows a GitHub repository page for 'SD-CIEL / JeuxJS'. The 'Commits' section displays three recent commits from the 'master' branch:

- #1 modification message console (SD-CIEL committed 2 minutes ago)
- Ajoutez des fichiers projet. (SD-CIEL committed 38 minutes ago)
- Ajouter .gitattributes et .gitignore (SD-CIEL committed 38 minutes ago)



8 Les bonnes pratiques de codage

⇒ A lire pour coder proprement : [Callback hell](http://callbackhell.com/) : <http://callbackhell.com/>

8.1 Utilisation du mode strict

En javascript, il peut se produire des erreurs dites « silencieuses » qui ne produisent aucune exception mais vont simplement stopper l'exécution du code. Cela peut être évité en s'imposant des règles de codage.

On utilisera dans ce TP le mode strict. Pour invoquer le mode strict, écrire en début de fichier :

```
'use strict';
```

Voir : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Strict_mode

8.2 Utilisation d'un Linter

Des outils de contrôle de code (linter) permettent de vérifier avant exécution si des règles de codage sont respectées.

Vous utiliserez pour ce TP **ESLint** dans Visual Studio

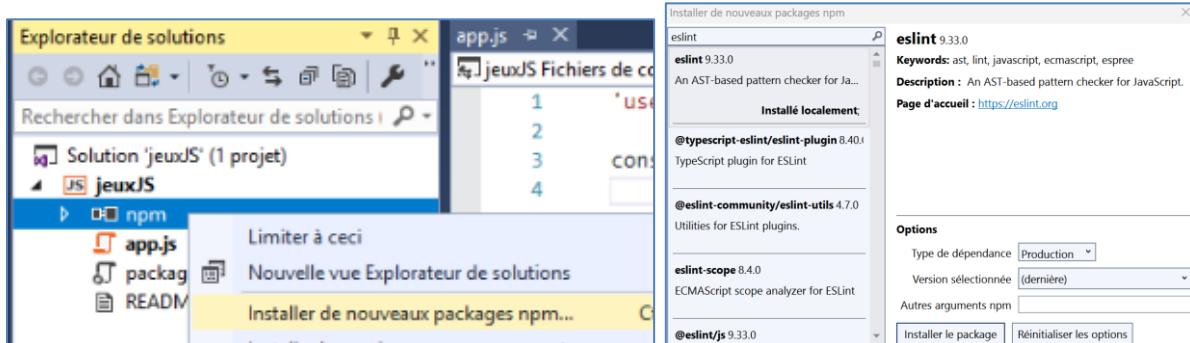
⇒ Vérifier que la vérification (linting) est activé (menu/Outils/Option et Editeur de texte/Javascript/Vérification(linting)) :

The screenshot shows the 'Options' dialog in Visual Studio. The left sidebar lists categories like HTML, JavaScript/TypeScript, and Validation. The right pane shows the 'Général' section with the 'Activer ESLint' option checked (True). Below it, under 'Langues lintables', several file types are listed with their respective linting settings.

Langue	Activer ESLint
Fichiers HTML	True
Fichiers JavaScript Lint	True
Fichiers JavaScript React Lint	True
Fichiers TypeScript Lint	True
Fichiers TypeScript React Lint	True
Fichiers Vue Lint	True

⚠ La suite est valable que pour les versions d'Eslint supérieur à 9

- ⇒ Dans l'explorateur de solutions, installer les packages suivant en gardant les options par défaut :
- Eslint
 - @eslint/js
 - Globals



- ⇒ Dans l'explorateur de solutions, ajouter un fichier `eslint.config.js` qui contient la configuration des règles que l'on désir respecter :

```
// eslint.config.cjs
const globals = require("globals");
const pluginJs = require("@eslint/js");

module.exports = [
  {
    languageOptions: {
      globals: globals.browser,
    },
    rules: {
      "indent": ["warn", 4],           // avertir une indentation de 4 espaces
      "linebreak-style": ["error", "windows"], // impose les fins de ligne Windows (\r\n)
      "quotes": ["error", "single"],   // impose l'usage de guillemets simples (' ')
      "semi": ["error", "always"]     // impose le point-virgule obligatoire
    },
    pluginJs.configs.recommended,
  ];
];
```

Il faut relancer VS pour prendre en compte une modification des règles

- ⇒ Voir la liste des règles : <https://eslint.org/docs/rules/>

Code des états d'activations des règles :

- "off" or 0 - turn the rule off
- "warn" or 1 - turn the rule on as a warning (doesn't affect exit code)
- "error" or 2 - turn the rule on as an error (exit code will be 1)

Résultat dans l'éditeur de code :



Dans les options de correction, on peut corriger les erreurs automatiquement.

On peut faire aussi la vérification en ligne de commande dans le dossier du projet, taper : `npx eslint`



9 Réalisation

9.1 Créer le serveur Web en Node.js

⇒ Installer le package « **express** » et garder les options par défaut.

Si le proxy pose problème pour l'accès aux dépôts npm alors copier les packages dans le dossier du projet, ils apparaissent ensuite dans l'explorateur de solutions.

⇒ Importer le module (bibliothèque en javascript) :

```
/* ***** Serveur Web ***** */
// var express = require('express');
```

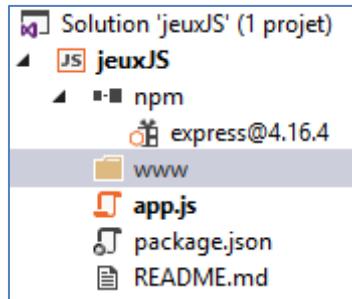
⇒ Créer une instance de l'application express :

```
var exp = express();
```

⇒ Paramétrer le répertoire racine du site avec la méthode use :

```
exp.use(express.static(__dirname + '/www'));
```

⇒ Créer le répertoire www dans l'explorateur VS :



⇒ Paramétrer la réponse à une requête GET avec une url sur la racine et sans nom de fichier :

```
exp.get('/', function (req, res) {
    console.log('Reponse a un client');
    res.sendFile(__dirname + '/www/index.html');
});
```

⇒ Ajouter/créer la page index.html à partir de l'explorateur VS.

⇒ Dans index.html, ajouter un titre et dans le corps un header avec un titre, une section et un footer.

⇒ Ajouter le traitement en cas d'erreur sur le serveur.

```
exp.use(function (err, req, res, next) {
    console.error(err.stack);
    res.status(500).send('Erreur serveur express');
});
```

⇒ Mettre le serveur web en écoute sur le port 80 :

```
exp.listen(80, function () {
    console.log('Serveur en ecoute');
});
```

- ⇒ Ajouter une variable « port » en début de programme pour le port.
- ⇒ Tester sur un navigateur.



Autoévaluation

❖ Commit #2

9.2 Créez une WebSocket « echo »

A lire sur les WebSocket : https://developer.mozilla.org/fr/docs/Web/API/WebSockets_API

- ⇒ Quel est l'avantage principal d'une WebSocket comparé à la technique AJAX ?

Le principal avantage d'une WebSocket est de permettre une communication bidirectionnelle et en temps réel entre client et serveur, contrairement à AJAX qui nécessite des requêtes ponctuelles du client.

Afin de créer un serveur WebSocket qui puisse fonctionner sur le même port que le serveur web, on utilisera un module extension d'express.

- ⇒ Installer le packet « **express-ws** »
- ⇒ A la suite du script, importer le module et paramétrer le chemin d'une WebSocket (/echo), la websoket aura pour url : adresselpServeur/echo :

```
/* ***** serveur WebSocket express ***** */
// var expressWs = require('express-ws')(exp);

// Connexion des clients à la WebSocket /echo et evenements associés
exp.ws('/echo', function (ws, req) {

    console.log('Connection WebSocket %s sur le port %s',
        req.connection.remoteAddress, req.connection.remotePort);

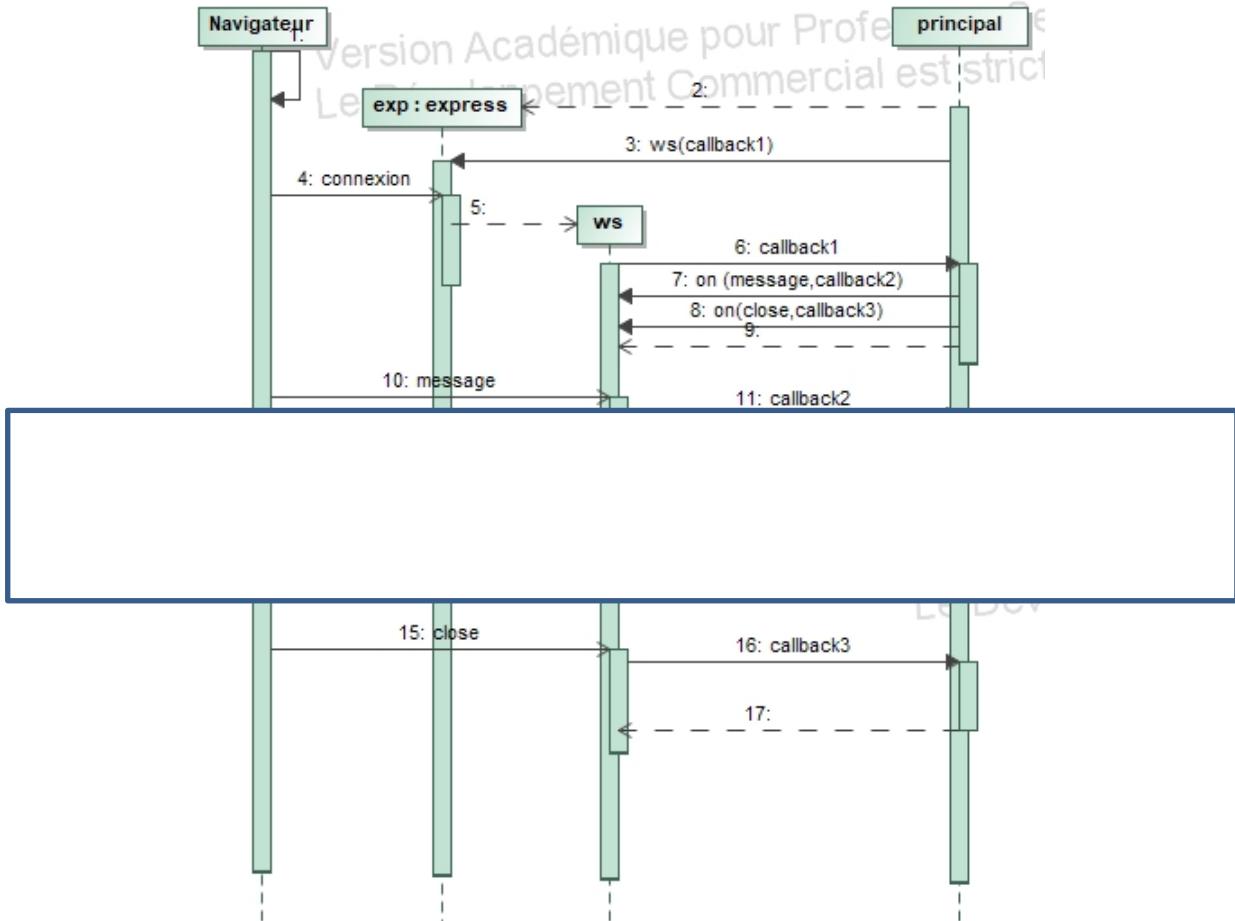
    ws.on('message', function (message) {
        console.log('De %s %s, message :%s', req.connection.remoteAddress,
            req.connection.remotePort, message);
        ws.send(message);
    });

    ws.on('close', function (reasonCode, description) {
        console.log('Deconnexion WebSocket %s sur le port %s',
            req.connection.remoteAddress, req.connection.remotePort);
    });
});
```

La méthode **exp.ws** permet de définir la fonction à appeler (**callback**) lorsqu'un client se connecte (**événement**), une WebSocket est alors créée. Dans cette fonction, on définit les fonctions à appeler lors des événements de réception d'un message du client (**événement message**) et de la fermeture de la WebSocket par le client (**événement close**).

Lorsqu'un message arrive du client, il est affiché sur la console du serveur et il est renvoyé au client (echo).

⇒ Compléter le diagramme de séquence :



⇒ L'ouverture du serveur doit se faire après la configuration du serveur web et WebSocket, déplacer à la fin du script :

```

/* ***** Serveur web et WebSocket en écoute sur le port 80 ***** */
// 
var portServ = 80;
exp.listen(portServ, function () {
  console.log('Serveur en écoute');
});
  
```

⇒ Dans la page web ajouter deux champs input et un bouton :

```

<div>
  <label>Message à envoyer : </label>
  <input type="text" id="messageEnvoi">
  <input type="button" id="Envoyer" value="Envoyer">
</div>
<div>
  <label>Message reçu : </label>
  <input type="text" id="messageReçu" readonly>
</div>
  
```

Ouvrir une connexion WebSocket sur le client (navigateur) en JavaScript :

- ⇒ Créer un nouveau dossier « js » dans le dossier « www » en utilisant l'explorateur de solution VS.
- ⇒ Ajouter un fichier JavaScript « echo.js » **dans** ce dossier « js ».
- ⇒ Associer le javascript à la page index.html en ajoutant dans le head :

```
<script type="text/javascript" src="js/echo.js"></script>
```

- ⇒ Dans le script echo.js ajouter les variables pour la WebSocket en mettant en dur votre adresse IP de poste :

```
var ipServeur = '172.17.50.xxx';      // Adresse ip du serveur
var ws;                                // Variable pour l'instance de la WebSocket.
```

- ⇒ Après le chargement de la page (événement) vérifier si le navigateur est compatible avec les WebSockets, ce qui est le cas maintenant pour la plupart des navigateurs :

```
window.onload = function () {
    if (TesterLaCompatibilite()) {
        ConnexionAuServeurWebsocket();
    }
    ControleIHM();
}

function TesterLaCompatibilite() {
    let estCompatible = true;
    if (!('WebSocket' in window)) {
        window.alert('WebSocket non supporté par le navigateur');
        estCompatible = false;
    }
    return estCompatible;
}
```

- ⇒ La connexion au serveur WebSocket est réalisée lors de linstanciation de la WebSocket, ensuite on traite les événements associés :

```
/* ***** Connexion au serveur WebSocket ***** */
// 
function ConnexionAuServeurWebsocket() {
    ws = new WebSocket('ws://' + ipServeur + '/echo');

    ws.onclose = function (evt) {
        window.alert('WebSocket close');
    };

    ws.onopen = function () {
        console.log('WebSocket open');
    };

    ws.onmessage = function (evt) {
        document.getElementById('messageReçu').value = evt.data;
    };
}

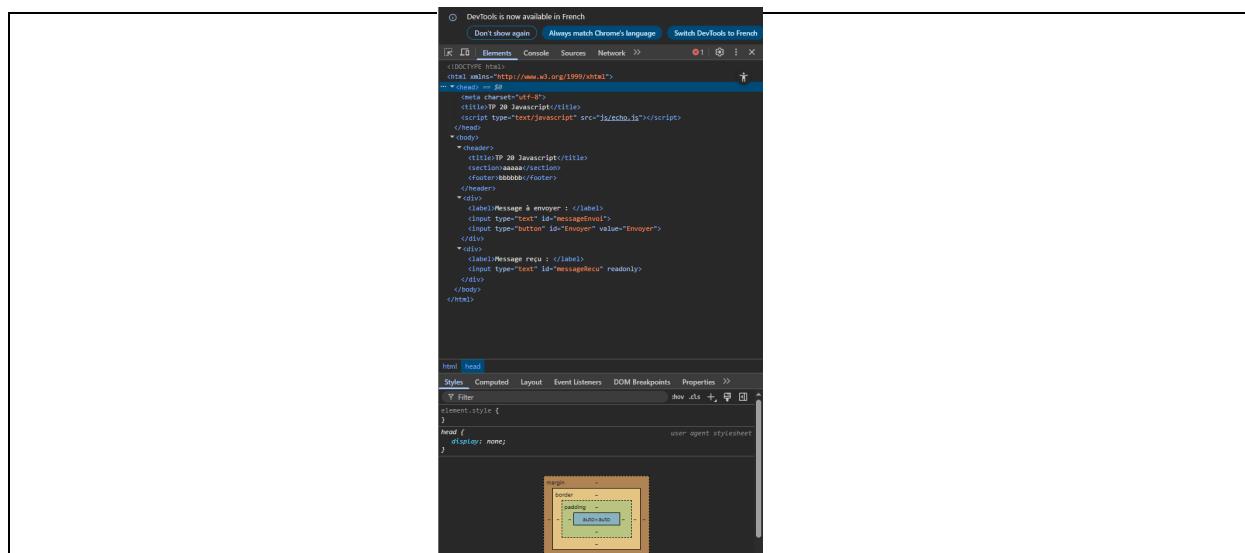
function ControleIHM(){
    document.getElementById('Envoyer').onclick = BPEnvoyer;
}

function BPEnvoyer(){
    ws.send(document.getElementById('messageEnvoi').value);
}
```

- ⇒ Tester et **commenter** le code.
- ⇒ Recopier le contenu de la console pour une connexion et un échange de message par webSocket :

Debugger attached.
TP CIEL
Serveur en écoute

- ⇒ Donner une capture d'écran de l'échange de message en WebSocket dans le DevTools du navigateur (F12) :



Autoévaluation

👉 Commit #3

- ⇒ Le serveur web et le serveur WebSocket ont la même adresse IP, affecter à ipServeur la valeur de location.hostname.

9.3 Application texte chat

Objectif : envoyer les messages des clients à tous les clients connectés, c'est-à-dire réaliser une diffusion (**Broadcasting**).

- ⇒ Ajouter une méthode de broadcast **sur le serveur** :

```
/* ***** Broadcast clients WebSocket ***** */
var awss = expressWs.getWss('/echo');
```

```

var WebSocket = require('ws');
aWss.broadcast = function broadcast(data) {
    console.log("Broadcast aux clients navigateur : %s", data);
    aWss.clients.forEach(function each(client) {
        if (client.readyState == WebSocket.OPEN) {
            client.send(data, function ack(error) {
                console.log("    - %s-%s", client._socket.remoteAddress,
client._socket.remotePort);
                if (error) {
                    console.log('ERREUR websocket broadcast : %s', error.toString());
                }
            });
        }
    });
};

```

Cette méthode **aWss.broadcast** enverra à tous les clients la chaîne passée en paramètre.

- ⇒ Appeler cette méthode à chaque réception d'un message.
- ⇒ Ajouter au message à diffuser l'adresse IP et port :

```
message = ws._socket._peername.address + ws._socket._peername.port + ' : '+message;
```

- ⇒ Changer le nom de la page html : index.html -> textchat.html
- ⇒ Tester : avec un client, plusieurs clients et en réalisant des déconnexions et connexions de clients.



Autoévaluation

❖ Commit #4

9.4 Jeux questions/réponses

Objectif : le serveur envoie une question à tous les clients connectés. Le premier client qui renvoie la bonne réponse a gagné. Une nouvelle question est ensuite posée.

- ⇒ Créer une nouvelle page html « qr.html » avec la même structure que précédemment. Avec un champ pour la question et un autre avec la réponse et un bouton validé.
- ⇒ Créer un nouveau fichier javascript « qr.js » et y recopier le script de « echo.js ». Faire référence à ce script dans le html. Changer l'url du serveur WebSocket ws://Ipserveur/qr
- ⇒ Tester la page web (la WebSocket n'est pas encore fonctionnelle).



Autoévaluation

❖ Commit #5

9.4.1 Utilisation de fonctions

⇒ Créer une nouvelle WebSocket qr avec le code suivant :

```
var question = '?';
var bonneReponse = 0;

// Connexion des clients à la WebSocket /qr et événements associés
// Questions/reponses
exp.ws('/qr', function (ws, req) {
    console.log('Connection WebSocket %s sur le port %s',
    req.connection.remoteAddress, req.connection.remotePort);
    NouvelleQuestion();

    ws.on('message', TraiterReponse );

    ws.on('close', function (reasonCode, description) {
        console.log('Déconnexion WebSocket %s sur le port %s',
        req.connection.remoteAddress, req.connection.remotePort);
    });

    function TraiterReponse (message) {
        console.log('De %s %s, message :%s', req.connection.remoteAddress,
        req.connection.remotePort, message);
        if (message == bonneReponse) {
            NouvelleQuestion();
        }
    }

    function NouvelleQuestion(){
        var x = GetRandomInt(11);
        var y = GetRandomInt(11);
        question = x + '*' + y + ' = ?';
        bonneReponse = x * y;
        awss.broadcast(question);
    }

    function GetRandomInt(max) {
        return Math.floor(Math.random() * Math.floor(max));
    }
});
```

⇒ Tester et **commenter** le code.



Autoévaluation

❖ Commit #6

- ⇒ Pourquoi les variables « question » et « bonneReponse » ne sont pas déclarées dans les fonctions de callback (« nouvelleQuestion » et « TraiterReponse ») ?

- ⇒ Expliquer ce qu'implique sur les variables l'imbrication des fonctions en javascript. En comparaison, rappeler ce qui est ou non possible en C++ avec les fonctions.

- ⇒ Renvoyer un message qui indique si la réponse est fausse ou juste, mais uniquement à celui qui envoi la réponse.

Ce message doit s'afficher dans le texte de la question pendant 3 secondes, ensuite une nouvelle question réapparaît à nouveau. Utiliser la fonction **setTimeout**.

Donner le code dans le cadre ci-dessous.



Autoévaluation

- ❖ Commit #7

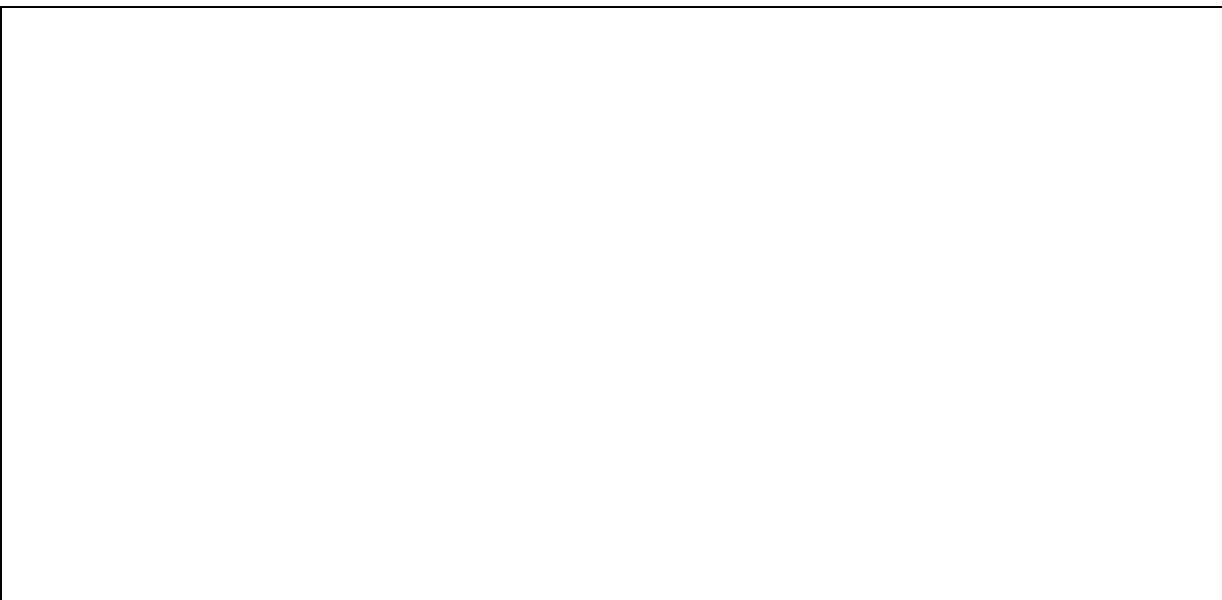
9.4.2 Question conversion base 2



Supplément

- ⇒ Proposer une question de conversion d'un nombre de base 2 vers la base 10.

Par exemple, pour un nombre de 8bits, tirer au hasard la valeur d'un entier entre 0 et 255, puis dans une boucle ajouter dans la chaîne du message les 0 et 1 correspondant à la valeur en base 2.



Autoévaluation

❖ Commit #8

9.4.3 Utilisation d'une classe

Objectif : structurer et organiser le code

Créer une classe CQr **en début de script** (on utilisera un fichier dédié plus loin dans le TP) :

 **Note** : Contrairement au C, les déclarations de fonctions sont remontées dans le code (**hoisting**), En revanche, ce n'est pas le cas pour les déclarations de classes. Ainsi, il est nécessaire de déclarer la classe avant de l'instancier. Dans le cas contraire, on obtient une ReferenceError.

⇒ Les attributs sont à déclarer et initialiser dans le constructeur :

```
class CQr {  
    constructor() {  
        this.question = '?';  
    }  
}
```

⇒ Ajouter l'attribut bonneReponse

⇒ Ajouter les méthodes : GetRandomInt, NouvelleQuestion, TraiterReponse

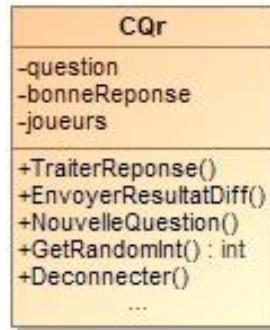
 **Note** : Une méthode n'utilise pas le mot clé « function ». Les méthodes sont implémentées dans le corps de la classe.

 **Note** : L'appel d'une méthode dans une méthode de la même classe, doit utiliser this.

Reprendre le code des fonctions précédentes :

TraiterReponse, prendra un paramètre supplémentaire : wsClient, qui permettra de référencer la WebSocket du client qui envoie un message :
TraiterReponse(wsClient, message) ...

Diagramme de la classe Cqr complet :



⇒ Instancier un objet de cette classe juste après sa déclaration :

```
var jeuxQr = new Cqr;
```

⇒ Le code du serveur webSocket pour le jeux qr utilisant la classe devient :

```
/* ***** serveur WebSocket express /qr ***** */
//
exp.ws('/qr', function (ws, req) {
    console.log('Connection WebSocket %s sur le port %s', req.connection.remoteAddress,
req.connection.remotePort);
    jeuxQr.NouvelleQuestion();

    ws.on('message', TMessage);
    function TMessage(message) {
        jeuxQr.TraiterReponse(ws, message);
    }

    ws.on('close', function (reasonCode, description) {
        console.log('Deconnexion WebSocket %s sur le port %s',
req.connection.remoteAddress, req.connection.remotePort);
    });
});
```

Dans la fonction de réception ws.on, on utilise une fonction intermédiaire TMessage pour le callback, car ws.on('message', jeuxQr.TraiterReponse(ws, message)) ne fonctionne pas.

Lorsque l'on passe une méthode d'un objet à un callback, le this de la méthode n'est plus celui de l'objet. On peut aussi utiliser la fonction bind pour lier this à l'objet :

```
ws.on('message', jeuxQr.TraiterReponse .bind(jeuxQr));
```

⇒ Tester sans et avec bind.



Autoévaluation

❖ Commit #9

9.4.4 Transmission d'information sérialisée en JSON

Objectif : Afin de pouvoir échanger des informations structurées entre le client et le navigateur, on établit un protocole simple qui utilisera un format JSON.

Les champs d'informations du client vers le serveur :

- le nom du joueur : nom
- la réponse : reponse

Les champs d'informations du serveur vers les clients (broadcast) :

- la liste des joueurs : joueurs
 - o nom
 - o score
- la question ou résultat : question
- la réponse : reponse

⇒ Ajouter un champ « nom » sur la page html.

⇒ Modifier de l'action sur le bouton pour envoyer les informations structurées en utilisant **JSON.stringify** :

```
ws.send(JSON.stringify({  
    nom: document.getElementById('nom').value,  
    reponse: document.getElementById('messageEnvoi').value  
}));
```

Stringify permet de **sérialiser** un objet qui est ici anonyme.

⇒ Donner la définition de sérialisation :

A la réception sur le serveur, modifier la fonction TraiterReponse pour prendre en compte la réponse et afficher le nom du joueur pour un résultat FAUX.

Pour **parser** le texte JSON : `var mess = JSON.parse(message);`

⇒ Tester avec le message en texte (pas en JSON)



Autoévaluation

❖ Commit #10

⇒ Pour l'envoi des résultats en JSON aux clients, créer une nouvelle méthode EnvoyerResultatDiff dans la classe CQr. Cette dernière composera la réponse du serveur aux clients et l'enverra. Déplacer l'envoi en Broadcast, de la méthode NouvelleQuestion

dans EnvoyerResultatDiff, puis créer le texte du message (on ne traite pour l'instant que l'information de la question) :

```
EnvoyerResultatDiff() {  
    var messagePourLesClients = {  
        question: this.question  
    };  
    awss.broadcast(JSON.stringify(messagePourLesClients));  
}
```

⇒ Faire de même pour les autres envois aux clients :

```
TraiterReponse(wsClient, message) {  
    var mess = JSON.parse(message);  
    if (mess.reponse == this.bonneReponse) {  
        this.question="Bonne reponse de "+mess.nom;  
    }  
    else {  
        this.question ="Mauvaise reponse de " + mess.nom;  
    }  
    this.EnvoyerResultatDiff();  
    setTimeout(() => { //affichage de la question 3s après  
        this.NouvelleQuestion();  
    }, 3000);  
}
```

⇒ Modifier ensuite la réception dans le script du client pour récupérer l'information « question ». Vérifier que cela fonctionne comme avant.



Autoévaluation

◆ Commit #11

9.4.5 Gestion des joueurs

Pour gérer les joueurs, on va créer un tableau de joueur dans la classe CQr. Un joueur aura les attributs suivants : un nom, un score et une référence à la WebSocket utilisée.

⇒ Créer le tableau de joueur en attribut de la classe :

```
this.joueurs = new Array();
```

Pour gérer les joueurs et leurs informations, l'algorithme (en pseudocode) peut être le suivant :

A L'EVENTEMENT d'un message reçu par le serveur :

SI le nom du joueur est renseigné **ALORS**

SI le joueur n'est pas dans la liste de joueurs **ALORS**

Ajouter un nouveau joueur à la liste et envoyer à tous les joueurs

SINON

SI c'est la bonne réponse **ALORS**

Incrémenter le score du joueur, créer une nouvelle question et envoyer à tous les joueurs

SINON

Envoyer au joueur FAUX et la question 3 secondes après.

⇒ Ecrire le code correspondant au pseudocode

En utilisant :

- Pour trouver l'index d'un joueur connaissant son nom

```
var indexjoueur = this.joueurs.findIndex(function (j) {  
    return j.nom === mess.nom;  
});
```

Si pas trouvé alors indexjoueur vaut -1

- Pour ajouter un joueur à la liste :

```
this.joueurs.push({  
    nom: mess.nom,  
    score: 0,  
    ws: wsClient  
});
```

⇒ Modifier la méthode EnvoyerResultatDiff :

```
// Envoyer a tous les joueurs un message comportant les resultats du jeu  
EnvoyerResultatDiff() {  
    // Recopie des joueurs dans un autre tableau joueursSimple sans l'objet  
WebSocket dans ws  
    var joueursSimple = new Array;  
    this.joueurs.forEach(function each(joueur) {  
        joueursSimple.push({  
            nom: joueur.nom,  
            score: joueur.score,  
        });  
    });  
    // Composition du message a envoyer  
    var messagePourLesClients = {  
        joueurs: joueursSimple,  
        question: this.question  
    };  
    // Diffusion (Broadcast) aux joueurs connectés;  
    this.joueurs.forEach(function each(joueur) {  
        if (joueur.ws != undefined) {  
            joueur.ws.send(JSON.stringify(messagePourLesClients), function  
ack(error) {  
                console.log(' - %s-%s',  
joueur.ws._socket._peername.address, joueur.ws._socket._peername.port);  
                if (error) {  
                    console.log('ERREUR websocket broadcast : %s',  
error.toString());  
                }  
            });  
        }  
    });  
};
```

⇒ Afficher les résultats de tous les joueurs sur la page html.

Peut-être affiché simplement dans une div de nom « résultats » avec le script :

```
document.getElementById('resultats').textContent = JSON.stringify(mess.joueurs);
```

⇒ Tester le fonctionnement nominal.



Autoévaluation

❖ Commit #12

9.4.6 Créer un module pour la classe CQr

- ⇒ Ajouter un nouveau fichier javascript (coté serveur) : jeuxqr.js
- ⇒ Déplacer la classe CQr dans ce fichier
- ⇒ Ajouter à la fin du fichier :

```
module.exports = CQr;
```

- ⇒ A la place de la classe dans le fichier app.js, ajouter :

```
var Cjeuxqr = require('./jeuxqr.js');
// instantiation du jeux QR
var jeuxQr = new Cjeuxqr;
```

- ⇒ La référence à awss ne peut plus être utilisée dans la classe, pour la diffusion changer la méthode EnvoyerResultatDiff pour envoyer à toutes les WebSockets qui seront mémorisées dans le tableau de joueurs :

```
EnvoyerResultatDiff() {
    // recopie des joueurs dans un autre tableau joueursSimple sans ws
    var joueursSimple= new Array;
    this.joueurs.forEach(function each(joueur) {
        joueursSimple.push({
            nom: joueur.nom,
            score: joueur.score
        });
    });

    var messagePourLesClients = {
        joueurs: joueursSimple,
        question: this.question
    };

    // broadcast aux joueurs connectés;
    this.joueurs.forEach(function each(joueur) {
        if (joueur.ws != undefined) {
            joueur.ws.send(JSON.stringify(messagePourLesClients), function
ack(error) {
                console.log(' - %s-%s', joueur.ws._socket._peername.address,
joueur.ws._socket._peername.port);
                if (error) {
                    console.log('ERREUR websocket broadcast : %s',
error.toString());
                }
            });
        }
    });
}
```

- ⇒ Il est nécessaire de prendre en compte la déconnexion et la reconnexion de joueur, ajouter la méthode :

```
Deconnecter(ws) {  
    var indexjoueur = this.joueurs.findIndex(function (j) {  
        return j.ws === ws;  
    });  
    if (indexjoueur != -1) {  
        this.joueurs[indexjoueur].ws = undefined;  
    }  
}
```

- ⇒ Appeler cette méthode lors de l'évènement de déconnexion (close) de la WebSocket.

```
ws.on('close', function (reasonCode, description) {  
    //console.log('Déconnexion WebSocket %s sur le port %s',  
    req.connection.remoteAddress, req.connection.remotePort);  
    jeuxQr.Deconnecter(ws);  
});
```

- ⇒ Tester.



Autoévaluation

❖ Commit #13

- ⇒ Transmettre l'information connecté/déconnecté dans l'attribut ws des joueurs transmis dans le message du serveur.

- ⇒ Que se passe-t-il si un joueur se connecte avec le même nom deux fois ?



Autoévaluation

❖ Commit #14