
Learning

# Rappel sur node.js

# Installation REPL

Lorsque vous installez node.js, vous installez toute une chaîne d'outils, notamment le gestionnaire de paquets **npm** (node package manager). Mais vous installez également un autre outil, la console de node.js nommée **REPL**(read evaluate print loop). Cet outil est accessible via la commande 'node', si vous la rentrez au sain de votre terminal, voilà ce que vous obtenez:

```
# La commande reste en standby, en attente d'instructions.
node
>

# On peut alors lui envoyer n'importe quelle instruction JS.
> console.log('Hello world')
Hello World
>

# On peut utiliser la touche tab pour bénéficier de l'auto complétion
# Et ainsi explorer les différents objets JS avec aisance. Essayez
# de rentrer la mot "global." dans votre console vous devriez
# avoir le retour suivant:

> global
Object [global] {
  global: [Circular],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Function]
  },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Function]
  }
}
>
```

REPL possède également quelques commandes spéciales que vous pourrez trouver à l'adresse suivante: [Documentation REPL (https://nodejs.dev/learn/how-to-use-the-nodejs-repl)](https://nodejs.dev/learn/how-to-use-the-nodejs-repl).

# Hello World

L'environnement node.js, bien que différent d'un navigateur, est avant tout un environnement d'éxécution Javascript. Son moteur d'interprétation, V8, conçu par Google, se charge de réceptionner, d'interpréter, de compiler et d'éxécuter le code JS fourni. Voici l'exemple le plus simple possible, le fameux 'hello world'

```
// un fichier éxécuté par node js est un fichier
// javascript classique, il sera interprété par le moteur
// javascript de node.js, çàd V8.

// le fichier se nomme helloworld.js
const msg = "Hello World";
console.log(msg);
```

Comme on peut le constater, le fichier js est tout simple. Node.js propose le support des toutes dernières versions de Javascript, mais également un support des versions plus anciennes.

```
# Pour éxécuter le script, il suffit de taper la commande suivante
node helloworld.js
```

L'éxécution d'un script est donx extrêment simple. Ce qui distingue avant tout node.js d'un autre environnement est son API. En effet, le DOM n'est par exemple pas disponible nativement au sein de node.js car toute cette partie de Javascript fait référence au document HTML auquel notre script est lié. Ici, notre script s'éxécute en dehors de tout contexte web, nous n'avons donc pas d'accès au DOM car cela n'a pas de sens. Toutefois, certaines librairies proposent le support d'un DOM virtuel.

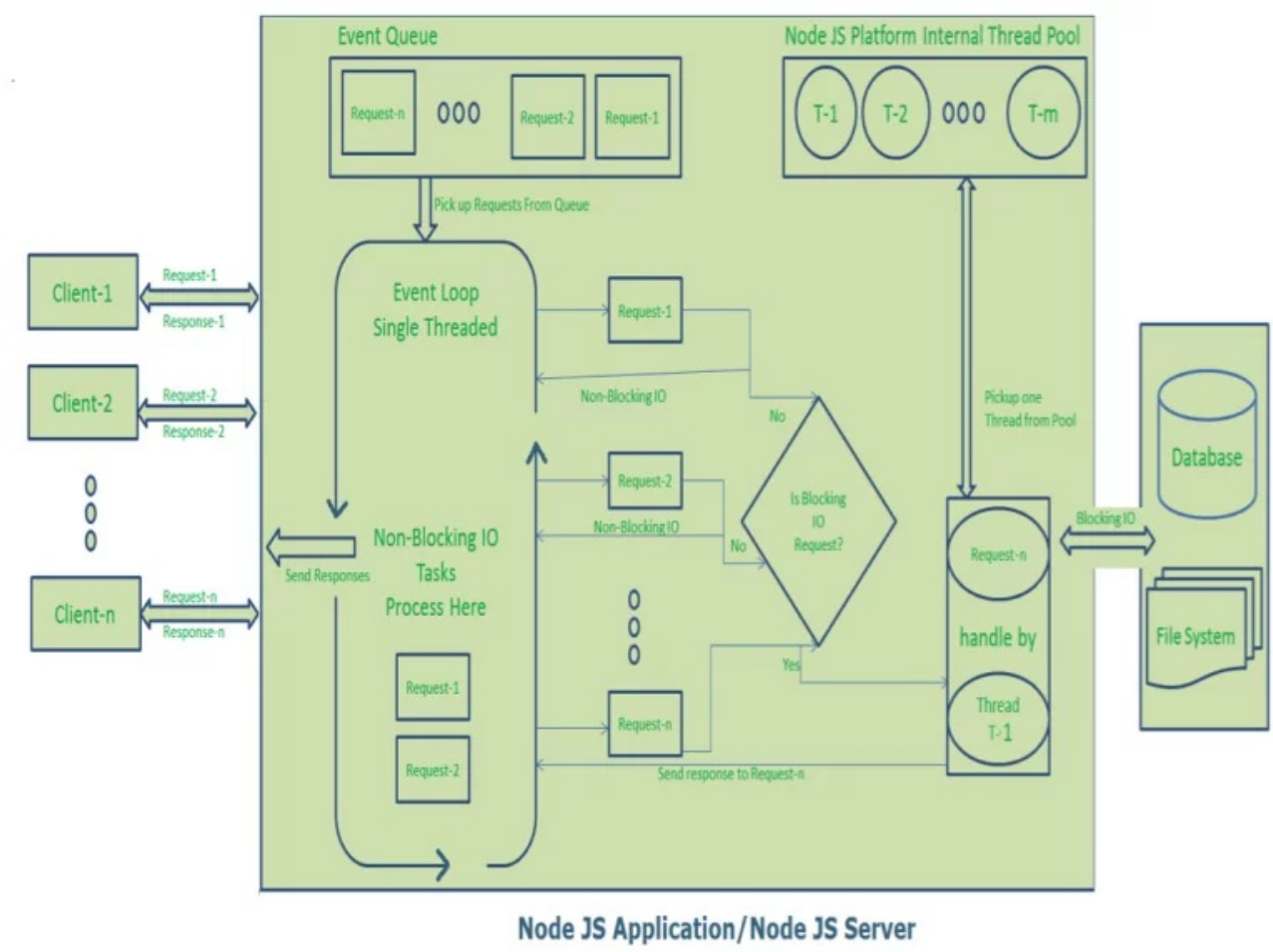
# Single Threaded Event Loop

L'environnement node.js, ne suit pas une architecture classique de type requête / réponse multi thread (comme apache). Càd qu'une requête ne donne pas naissance à un thread en particulier. Au lieu de ça, node.js fonctionne sur un modèle mono thread avec une **avec une boucle de rendu**. Cette boucle de rendu repose sur le modèle évènementiel de Javascript, basé sur des fonctions de **callback** (rappel). Voici la liste des étapes suivies par ce modèle:

- Le client envoie une requête au serveur
- Le serveur node.js maintient en interne un pool de threads, afin de fournir un service aux requêtes du client.
- Le serveur node.js reçoit les requêtes et les places au sein d'une file d'attente, nommée la **'Event Queue'**.
- Node.js possède en interne un composant nommé 'Event Loop', ce composant est chargé de placer les requêtes entrante dans la boucle et de traiter la prochaine, et ce indéfiniment.
- La 'Event Loop' est mono thread.
- La 'Event Loop' vérifie s'il y a des requêtes client en attente au sein de la 'Event Queue', s'il n'y en a pas, elle attend indéfiniment.
- Si oui, elle retire une requête de la file d'attente et:
- Démarre le traitement de cette demande client
- Si cette demande client ne nécessite aucune opération d'E / S de blocage, prépare la réponse et la renvoie au client.
- Si cette demande de client nécessite des opérations de blocage d'E / S telles que l'interaction avec la base de données, le système de fichiers, les services externes, elle suivra une approche différente.
- Vérifie la disponibilité des threads à partir du pool de threads interne
- Ramasse un thread et attribue cette demande client à ce thread.
- Ce thread est chargé de prendre cette demande, de la traiter, d'effectuer des opérations de blocage d'E / S, de préparer la réponse et de la renvoyer à la boucle d'événements.
- La boucle d'événement à son tour envoie cette réponse au client respectif.

# Single Threaded Event Loop

Voici un schéma vous résumant l'intégralité des étapes suivies par node.js



# Non Blocking API

Le blocage se produit lorsque l'exécution de JavaScript supplémentaire dans le processus Node.js doit attendre la fin d'une opération non initialisée par JavaScript. Cela se produit car la boucle d'événements ne peut pas continuer à exécuter JavaScript pendant qu'une opération de blocage est en cours.

Dans Node.js, un code JavaScript présentant des performances médiocres en raison de son utilisation intensive en CPU n'est pas considéré comme un code bloquant. Le code bloquant est celui qui attend un retour synchrone d'une opération externe, comme l'accès à une base de données ou au système de fichiers. Les méthodes synchrones de la bibliothèque standard Node.js qui utilisent libuv sont les opérations de blocage les plus couramment utilisées . Les modules natifs peuvent également avoir des méthodes de blocage. Voici quelques exemples:

```
// Voici un exemple de code bloquant
// car il attends le retour du système de fichiers avant de poursuivre.
const fs = require('fs');
const data = fs.readFileSync('/file.md');

// Et voici un exemple asynchrone équivalent :
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

Le premier exemple semble plus simple que le second mais présente l'inconvénient que la deuxième ligne bloque l'exécution de tout JavaScript supplémentaire jusqu'à ce que l'intégralité du fichier soit lu. Notez que dans la version synchrone, si une erreur est générée, elle devra être interceptée ou le processus plantera. Dans la version asynchrone, c'est à l'auteur de décider si une erreur doit être générée comme indiqué.

L'exécution de JavaScript dans Node.js est mono thread, donc la concurrence fait référence à la capacité de la boucle d'événements à exécuter des fonctions de callback après avoir terminé d'autres travaux. Tout code censé s'exécuter de manière simultanée doit permettre à la boucle d'événements de continuer à s'exécuter au fur et à mesure que des opérations non JavaScript, telles que les E / S, se produisent.

Pour plus d'informations rendez-vous ici: [Blocking VS non blocking operations (https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/)](https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/).

# PRINCIPES DE BASE DU JS

# Les caractéristiques du Javascript

Le Javascript est un langage de haut niveau dérivé du C, à ce titre il possède plusieurs éléments de syntaxe proches de ce dernier tout en se distinguant sur d'autres. Il faut retenir avant tout les points suivants( liste non exhaustive ):

- Javascript permet d'interragir avec le document HTML et le style des élements.
- Il s'agit d'un langage éxécuté avant tout côté client.
- Il peut écouter et réagir aux évènements.
- C'est un langage orienté objet par prototypage.
- Les variables n'ont pas de type défini
- C'est un langage compilé à la volée, à l'aide d'un J.I.T ( Just In Time Compiler ).
- Les instructions sont séparées par des points-virgules, leur usage est optionnel.
- Les commentaires '//' peuvent commenter une seule ligne.
- Les commentaires '/* */' peuvent plusieurs lignes.
- Le mode non strict est le mode par défaut.
- Utiliser l'instruction 'use strict'; en début de fichier permet de passer en mode strict.

# Les différents types de données

Il existe différents types de données en Javascript ainsi qu'une multitude d'objets disponibles à travers certaines APIs ben spécifiques. Voici la liste des types de données de base:

- Nombres : int / float
- Chaînes de caractères (string) : encadrées de guillemets (simples ou doubles)
- Booléens (boolean) : false / true.
- Null (pas de valeur).
- undefined: absence de valeur définie.
- Function: les functions sont aussi des objets en Javascript.
- Objets : standard (Object)
- Tableau: (Array)
- Date (Date)
- Etc...

# Les outils

Javascript n'est pas un langage précompilé, cela veut dire que si une erreur doit se produire, elle se produira lors de l'éxécution du script, lorsque le J.I.T (Just In Time compiler) passera dans la portion de code "coupable".

Il existe cependant quelques outils qui peuvent faciliter la vie du développeur, en laissant la possibilité de debugger de manière efficace un script Javascript, en voici une liste non exhaustive:

- Console des différents navigateurs: ( les meilleures sont celles de Chrome et de Firefox )
- L'environnement Node.js: ... Met à notre disposition une multitude de modules pour nous aider à debugger sous Node.
- L'instruction 'debugger': à mettre dans un script, nous permet de poser un breakpoint dans le code, on peut donc vérifier à tout moment, en ouvrant le console, l'état de chaque variable du programme à un instant T.
- Les méta langages tels TypeScript, qui viennent avec une palette d'outils comme un compilateur et un transpiler TypeScript -> Javascript ES5 & ES6. Ces langages sont fournis, précis, outillés et nous permettent d'obtenir du Javascript propre en sortie.

# LE LANGAGE JAVASCRIPT

# Déclaration et portée des variables

Javascript permet de stocker des données à l'intérieur de structures appelées variables. Ces dernières peuvent contenir tout type de valeur et ne sont pas limitées dans leur changement, par exemple une variable contenant une chaîne de caractères pourra stocker plus tard une valeur numérique ( bien que ce ne soit pas une bonne pratique ).

Les variables ont une portée, çàd qu'elles sont accessibles dans l'espace dans lequel elles ont été définies ainsi que dans tout les autres sous-espaces ( ou espaces imbriqués de types closures etc ... ).

La syntaxe pour définir une variable est: '**var ma_variable**'. Un nom de variable valide comporte tout les caractères alphanumériques hormis les caractères spéciaux et doit impérativement commencer par une lettre.

```
// définit une variable contenant la chaîne hello world
var msg = "hello world";

function hello() {
    // ici la variable est accessible car elle a été définie
    // dans l'espace global qui est un espace parent de la fonction
    alert(msg);
}
```

# Les tableaux

Javascript nous mets à disposition des structures appelées **Arrays**, ou tableau. Ces tableaux permettent de stocker une suite de valeurs au lieu d'une seule. Il est possible de stocker des valeurs de type complètement différents les unes à la suite des autres. Pour déclarer un tableau, il existe deux types de syntaxes, la syntaxe objet et la syntaxe JSON.

Pour accéder à un élement de tableau, il suffit d'utiliser la syntaxe suivante: '** tab[index]**', ou index représente la position d'un élément à l'intérieur du tableau. Les tableaux sont** '0-based**', cela signifie que le 1er élement se trouve en position 0, le 2ème en position 1 etc...

```
// déclare un tableau vide ...
var tab1 = [];  // syntaxe JSON
var tab2 = new Array(); // syntaxe objet

// déclare un tableau pré-rempli ...
var tab3 = [0, 80, "hello world"];

// affiche "hello world"
alert(tab3[2]);

// pousse un élement en 5ème position
alert(tab3[4]);
```

# Opérateurs logiques et arithmétiques

Javascript introduit un certain nombre d'opérateur, un opérateur binaire nécéssite deux opérandes, un opérateur unaire n'en nécéssite qu'un seul, et un opérateur ternaire nécéssite trois opérandes.

- Les opérateurs d'affectation
- Les opérateurs de comparaison
- Les opérateurs arithmétiques
- Les opérateurs binaires
- Les opérateurs logiques
- Les opérateurs de chaînes
- Les opérateurs ternaire (conditionnel)
- L'opérateur virgule
- L'opérateur unaire
- L'opérateur relationnels

```
"Les opérateurs d'affectation",
("Les opérateurs de comparaison",
"Les opérateurs arithmétiques",
"Les opérateurs binaires",
"Les opérateurs logiques",
"Les opérateurs de chaînes",
"Les opérateurs ternaire (conditionnel)",
"L'opérateur virgule",
"L'opérateur unaire",
"L'opérateur relationnels")

//opérateurs arithmétiques:
alert(1 + 2); //addition
alert(1 * 10); // multiplication
alert(1 / 0.1); // division
alert(10 - 5); // soustraction
alert( 10 % 3 ); // modulo

// opérateurs logiques:

alert(10 == "10"); // égalité non stricte
alert( 10 === "10"); // égalité stricte
alert( 10 != "10" ); // non égalité non stricte
alert( 10 !== "10" ); // non égalité stricte
alert( (10==10) && (5==5)); // A est vrai et B est vrai
alert( (10==10) || (5==5)); // A est vrai ou B est vrai


//ternaire;
alert( (10==8) ? "yes" : "no");
```

# Les boucles

Les boucles permettent de répéter des actions simplement et rapidement, il y en a différents types mais elles se ressemblent toutes au sens où elles répètent une action un certain nombre de foi

```
var i = 0, prop = null, obj = {"name":"Dark Vador", "job":"Sith Lord" };
// boucle for
for (i = 0; i < 10; i++) {
    console.log(i);
}

// boucle while
i = 0;
while (i < 10) {
    console.log(i++);
}

// boucle do while
i = 0;
do {
    console.log(i++);
}
while (i < 10)

//boucle for in
for (prop in obj) {
    console.log(prop, obj[prop]);
}
```

# Déclaration et appel de fonctions

Une fonction est une suite d'instructions que le développeur pourra utiliser ( on dit invoquer ) quand il en aura besoin. Elle peut avoir un nom, recevoir des paramètres, et retourner une valeur. Javascript apporte son propre lot de fonctions mais il est possible de coder les siennes !

```
// cette fonction nommée saySomething affiche une boîte de dialogue
// contenant le message contenu dans le paramètre p_msg, elle fait appel
// à une fonction Javascript nommée alert.

function saySomething(p_msg) {
    alert(p_msg);
}

// cette fonction ne porte pas de nom, on dit qu'elle est anonyme.
// ici, elle est stockée dans une variable nommée myfunc
var myfunc = function () {
    alert("anonymous function");
};

/// invoque la fonction saySomething et affiche une boite de dialogue
saySomething("Hello World");

// invoque la fonction anonyme à l'aide de la variable myfunc
myfunc();
```

# Gestion des erreurs et des exceptions

En Javascript, il est fréquent que des erreurs puissent survenir au sein du code, parfois elles n'ont (presque) aucune conséquence et le programme continue de fonctionner tant bien que mal. D'autres fois en revanche, il est tout à fait possible que le code plante complètement, soit à cause d'une étourderie du développeur ( c'est plus fréquent qu'on ne le pense ), soit parce qu'une erreur a été jeté volontairement.

Il est en effet possible d'interrompre volontairement le code en jetant un objet de type Error suite à un comportement prévu mais non voulu, pour cela il faut utiliser l'opérateur **throw**. Puisqu'il est possible de jeter des erreurs, il est possible de les attraper à l'aide du duo d'opérateurs **try ... catch**

```
// on essaie d'éxécuter le code contenu dans le bloc try ...
try {
    // ... code potentiellement à risque.
}
catch (error) {
    //  mais une erreur peut se produire et être jetée dans ce cas on l'attrape avec catch
    console.log(error);
}


// on peut également jeter nos propres erreurs à l'aide de l'opérateur throw
throw (new Error("This is an error, don't do that again !"));
```

# LA PROGRAMMATION EVENEMENTIELLE

# Javascript, un langage évènementiel

Javascript fait partie de la famille des langages évènementiels comme Actionscript, Lingo, Qt en C++ (bibliothèque), cela signifie qu'il nous procure, de manière native ( ou à l'aide de bibliothèques comme jQuery par exemple ), des mécanismes pour programmer de façon évènementielle.

En informatique, la programmation événementielle s'oppose à la programmation séquentielle. Le programme sera principalement défini par ses réactions aux différents événements qui peuvent se produire, çàd des changements d'état de variable, l'incrémentation d'une liste, un mouvement de souris ou de clavier...

En Javascript, elle est surtout utilisée avec les objets du DOM, mais d'autres objets peuvent être utilisés en programmation évènementielle. Pour cela, il faut utiliser conjointement les méthodes suivantes des objets implémentant l'interface IEventDispatcher:

- **addEventListener**: Qui nous permet d'ajouter un écouteur d'évenements ( une fonction de rappel ) qui se déclenchera en réaction à un évènement.
- **removeEventListener**: Qui nous permet d'enlever un écouteur d'évènements ajouté auparavant.
- **dispatchEvent**: Qui nous permet de lancer un évènement à la main, s'utilise avec les objets de type Event
- **Objet Event**: Cet objet est utilisé pour créer des évènements déjà connus ou personnalisés, attention les évènements utilisateurs tels que les clicks de souris, les frappes au clavier ne peuvent pas être simulés parfois, pour des raisons de sécurité.

# Les évènements de chargement de page

En Javascript, il est courant d'attendre que la page soit chargée avant de déclencher un script quelconque. pour cela, il existe plusieurs méthodes, notamment une qui nous dicte d'inclure le script en tant que dernier enfant du noeud body.

Parfois, cela peut être une bonne méthode, mais le plus souvent, il s'agit d'une mauvaise pratique. En effet, il existe deux évènements spécifiques au chargement d'une page, l'un envoyé par l'objet window, et l'autre par l'objet document:

- **load event**: Envoyé par l'objet window lorsque l'ensemble des noeuds HTML de la page et toutes les ressources qui y sont associées (images, vidéos, sons etc...) sont parsés, téléchargés, prêts à être manipulés.
- **DOMContentLoaded event**: Envoyé par l'objet window.document lorsque l'ensemble du HTML de la page est parsé et prêt à être manipulé. On ne tient pas compte des autres ressources.

```
function onPageFullyLoaded(event){
    console.log("HTML & other ressources fully loaded");
}

function onHTMLFullyLoaded(event){
    console.log("HTML fully loaded");
}

window.addEventListener("load", onPageFullyLoaded, false );
window.document.addEventListener("DOMContentLoaded", onHTMLFullyLoaded, false);
```

# Les évènements utilisateur

Il est possible de détecter des évènements déclenchés par une action de l'utilisateur comme par exemple:

- Le click souris: à l'aide de l'évènement click'
- Passer par dessus un élément: à l'aide de l'évènement mouseover
- Appuyer sur une touche du clavier: à l'aide de l'évènement keydown

```
function keyDownHandler(event){
    console.log("appui sur une touche");
}
window.addEventListener("keydown", keyDownHandler, true);


function clickHandler(event){
    console.log("click souris");
}

window.addEventListener("click", clickHandler, true);
```

Il en existe beaucoup d'autres que vous pouvez découvrir sur la documentation officielle Javascript de Mozilla Developer Network: https://developer.mozilla.org/fr/

# Différences entre les navigateurs (non modernes)

Il existe des différences d'API entre les navigateurs les plus anciens en ce qui concerne la programmation évènementielle. Par exemple, au lieu de la méthode standardisée **addEventListener** le navigateur Internet Explorer a très longtemps utilisé la méthode **attachEvent** ainsi que les méthodes ** detachEvent et fireEvent**.

Mais il n'y a pas qu'Internet Explorer qui pouvait poser problème, à l'époque chaque navigateur avait ses propres subtilités et il était assez compliqué de programmer efficacement de façon évènementielle. C'est en partie pour répondre à ce besoin qu'a été créée la bibliothèque **jQuery**.

# ES6 et au delà: Le javascript moderne

# Bonnes pratiques Ecmascript 5

Javascript est un langage qui évolue en permanence, ceci dit, dans sa version Ecmascript 5, les bonnes pratiques restent globalement inchangées. Les voici en versions condensées, et bien entendu cette liste n'est en rien exhaustive.

- Eviter les variables globales
- Déclarer les variables locales à l'aide du mot clé "var"
- Le mode strict doit être utilisé
- Eviter la fonction eval()
- Organiser son code en module
- Une expression doit toujours se terminer par un point-virgule
- Le code doit être découpé en plusieurs fichiers

... Et bien d'autres que vous pourrez retrouver par exemple à l'adresse suivante : https://maxlab.fr/javascript/bonnes-pratiques-javascript-pour-lentreprise

# ES7/ES6/ES2015, présentation générale.

ECMAScript est un ensemble de normes concernant les langages de programmation de type script et standardisées par Ecma International dans le cadre de la spécification ECMA-262. Il s'agit donc d'un standard, dont les spécifications sont mises en œuvre dans différents langages de script, comme JavaScript ou ActionScript, ainsi qu'en C++ (norme 2011). C'est un langage de programmation orienté prototype.

Pour le langage Javascript, la norme ECMASCRIPT 5 est aujourd'hui la plus répandue, toutefois certains navigateurs, comme Chrome (Google Inc.), mettent déjà à disposition des développeurs une implémentation de Javascript respectant la norme ECMA 6 (ES2015).

Nommée ES2015, la dernière version d'ECMAScript a été publiée en juin 20156. Son support par les navigateurs évolue progressivement, mais il est possible d'utiliser un transcompilateur (tel que Babel.js8) vers ES5 pour développer dès aujourd'hui en ES6.

Première version publiée après le changement du processus de normalisation, l'ES7, ou ES2016, apporte peu de changements au langage. Le nouveau processus prévoit en effet de publier chaque année une nouvelle norme avec les ajouts qui ont eu lieu dans l'intervalle, afin d'éviter de publier des changements énormes comme ce fut le cas en 20159.

# ECMASCRIPT 6 et 7, les nouveautés

Les normes ECMASCRIPT 6 et 7 arrivent avec leurs lots de nouvelles fonctionnalités notamment les suivants ( mais pas que ):

- Les mot-clés let et const
- Les templates et chaînes de caractères
- Les paramètres par défaut
- Les promesses
- Les fonctions lambda ( arrow functions )
- Les classes et l'héritage

Nous allons détailler un peu plus chacune de ces fonctionnalités.

## Les mot-clés let et const

Le mot clé **let** permet de déclarer une variable limitée à la portée d'un bloc, c'est-à-dire qu'elle ne peut être utilisée que dans le bloc où elle a été déclarée, ce qui n'est pas le cas avec var.

Le mot-clé **const** sert à définir une valeur immuable, accessible uniquement en lecture seule.

```
function whoIsTheSithLord() {
    // la valeur ne peut plus être modifiée
    const MOVIE = "STAR WARS";

    if (true) {
        // existe en dehors du bloc conditionnel
        var sith_lord = "Palpatine";
        //existe uniquement dans le bloc conditionnel
        let padawan = "Dark Vador";
    }

    console.log(sith_lord); // displays Palpatine
    console.log(padawan); // Uncaught Reference Error
    conole.log(MOVIE);
}

whoIsTheSithLord();
```

# Templates Strings

Les "templates strings" permettent d'insérer plus facilement des valeurs de variables au sein d'une chaîne de caractère, ces dernières peuvent également être définies sur plusieurs lignes.

```
let jedi = {surname:"Obiwan", name:"Kenobi"};
let msg = `${jedi.name} ${jedi.surname} is the Jedi Master`;
console.log(msg);
```

## Paramètres par défaut

Il est dorénavant possible de donner une valeur par défaut aux paramètres, cela évite les lignes de codes superflues et facilite la lecture du code.

```
function whoIsTheSithLord(p_name='Dark Vador')
{
    console.log("The Sith lord is: ", p_name);
}

whoIsTheSithLord();
```

## Les fonctions fléchées/lambdas

Il s'agit d'une nouvelle façon d'écrire les fonctions anonymes, plus compacte et permettant de lier définitivement le contexte d'éxécution de la fonction (this).

```
function launchFunc(p_function) {
    p_function();
}

// ecma 5 style anonymous function
launchFunc(
    function () {
        console.log("I am The Sith Lord")
    }
);

// ecma6 arrow function
launchFunc(
    () => { console.log("I am The Sith Lord") }
);
```

# Les promesses

L'objet Promise (pour « promesse ») est utilisé pour réaliser des traitements de façon asynchrone. Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais.

L'interface Promise représente un intermédiaire (proxy) vers une valeur qui n'est pas nécessairement connue au moment de sa création. Cela permet d'associer des gestionnaires au succès éventuel d'une action asynchrone et à la raison d'une erreur. Ainsi, des méthodes asynchrones renvoient des valeurs comme les méthodes synchrones, la seule différence est que la valeur retournée par la méthode asynchrone est une promesse (d'avoir une valeur plus tard).

```
new Promise((resolve, reject) => {
    const xhr = new XmlHttpRequest();
    xhr.open("GET", url);
    xhr.onload = () => resolve(xhr.responseText);
    xhr.onerror = () => reject(xhr.statusText);
});

function resolve(p_msg){
    console.log("resolve: ", p_msg);
}

function reject(p_msg){
    console.log("reject: ", p_msg);
}
```

## Classes et Héritages

Ecma 6 introduit des mécanismes Orienté Objet par classe grâce aux mots-clés **class** et **extends**, malheureusement ces derniers n'introduisent aucun changement dans la manière dont les objets sont réellements agencés et conçus au sein de la VM Javascript, il s'agit uniquement de sucre syntaxique.

```
class Personnage {
    constructor() {
        this.name = "";
    }

    sayMyName() {
        console.log(this.name);
    }
}

class JediKnight extends Personnage {
    constructor(p_name) {
        this.power = 250;
        this.name = p_name;
    }
}

let obiwan = new JediKnight("Obiwan Kenobi");
obiwan.sayMyName();
```

# Les Observables

Un Observable est un producteur de données qui peut être observé. On le mettra sous observation avec la méthode **subscribe** et cette observation sera exécutée par un objet de type Observer. Les Observables se sont imposés dans le monde de la programmation réactive ( reactive programming ) par le biais de la librairie Reactive X, abrégé RxJs pour sa version Javascript. Un Observable peut également produire des données de façon asynchrone et il est possible de fusionner plusieurs Observables entre eux.

Il faut bien noter que les Observables sont souvent associés aux promesses, voire décrits comme des super-promesses, ce qui n'est pas tout à fait exact, un article entier a d'ailleurs été rédigé à l'attention des développeurs à l'adresse suivante: https://medium.com/@benlesh/learning-observable-by-building-observable-d5da57405d87 et explique un peu plus en détail ce que sont les Observables.

```
Rx.Observable.create(function(observer) {
    setTimeout(() => observer.next("valeur A"), 700);
    setTimeout(() => observer.next("valeur B"), 400);
})
 .subscribe(e => console.log(e));

 // affiche "valeur A", puis "valeur B"  dans la console
```

# Reactive Extension For Javascript (RxJS)

Reactive Extensions (Rx) est une bilbiothèque permettant de développer des applications "orientées évènements" et utilisant des requêtes asynchrones en liant entre elles des séquences d'objets de type Observables.

Les séquences de données peuvent prendre plusieurs formes, telle qu'un flux de données issue d'un webservice, des requêtes vers des webservices, des notifications systèmes, ou bien encore une séquence d'évènements utilisateurs etc...

Rx represente toutes ces données sous la forme d'une séquence d'Observables, une application peut souscrire à ce flux de données et être notifiée chaque fois qu'une donnée est ajoutée au flux.

La comparaison avec les promesses est alors tentante et il est possible de se faire une idée plus précise de ce que sont les Observables et dans quels domaine les préférer par rapport aux Promises à l'adresse suivante: https://xgrommx.github.io/rx-book/index.html

# Les outils

L'une des choses les plus difficiles à déterminer lorsque l'on commence un développement en Angular est la stack complète de développement. Fort heureusement, une multitude d'outils tous plus performants les uns que les autres sont venus renforcer l'écocsystème Javascript ces derniers temps, et ce, grâce à **node.js** et son gestionnaire de dépendances spécialisé front-end **npm**. L'on peut donc compter au rang des outils quasiment indispensables:

- Node.Js et NPM (node package manager)
- Typescript ( langage poussé par Google pour le développement Angular )
- Un bon éditeur Typescript (Micorsoft Visual Code)
- Un outil de scaffolding ( génération de structure de projets et codes ) comme @angular/cli
- Un module loader, comme Webpack
- Un bon navigateur et/ou environnement d'éxécution ( Google Chrome, Ionic ... )

# Le langage Typescript

TypeScript est un langage de programmation libre et open source développé par Microsoft qui a pour but d'améliorer et de sécuriser la production de code JavaScript. C'est un sur-ensemble de JavaScript (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript). Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.

TypeScript permet un typage statique optionnel des variables et des fonctions, la création de classes et d'interfaces, l'import de modules, tout en conservant l'approche non-contraignante de JavaScript. Il supporte la spécification ECMAScript 6.

- Typage statique
- Typage générique
- Interfaces
- Classe, classe abstraite, expressions de classe
- Modules
- Mixin
- Enumérations
- Paramètres optionnels
- Symboles

... Et bien d'autres, d'ailleurs depuis la version 1.6, le format JSX est supporté.

# ES6 Module Pattern

La norme ES6 prévoit de founir au développeur la possibilité de regrouper et empaqueter des objets, classes, fonctionnalités et valeurs au sein de **modules**, dans la continuité des pattern modulaires déjà bien ancrés au sein de l'écosystème Javascript (AMD par exemple).

L'instruction export est utilisée pour permettre d'exporter des fonctions et objets ou des valeurs primitives à partir d'un fichier (ou module) donné. Ces fonctions et objets peuvent ensuite être utilisées dans d'autres fichiers grâce à import.

Cette fonctionnalité n'est pas encore implémentée nativement par la plupart des navigateurs, en revanche, elle est gérée par de nombreux transpileurs comme **Traceur, Babel ou Rollup.**

```
function whoIsTheSithLord() {
    console.log("Palpatine is the Sith Lord !");
}

// on exporte une fonction déclarée plus haut
export { whoIsTheSithLord };

// on exporte une constante
export const DARK_VADOR = "Dark Vador";
```

# ES7 Async functions

La déclaration async function définit une fonction asynchrone qui renvoie un objet AsyncFunction. On peut également définir des fonctions asynchrones grâce au constructeur AsyncFunction et via une expression de fonction asynchrone.

Lorsqu'une fonction asynchrone est appelée, elle renvoie une promesse. Lorsque la fonction asynchrone renvoie une valeur, la promesse est résolue avec la valeur renvoyée. Lorsque la fonction asynchrone lève une exception, la promesse est rompue avec la valeur de l'exception.

Une fonction asynchrone peut contenir une expression await qui permet d'interrompre l'exécution de la fonction asynchrone en attendant la résolution d'une promesse passée à l'expression. L'exécution de la fonction asynchrone reprend lorsque la promesse est résolue.

L'objectif des fonctions asynchrones avec await est de simplifier le comportement des promesses lors d'opérations synchrones et d'opérer sur des groupes de promesses. Si les promesses sont en quelque sorte des callbacks organisés, async/await permet de combiner les générateurs et les promesses. On peut trouver des exemples à l'adresse suivante: https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/async_function

# Module Loader & Webpack

Un module loader est un outil qui fournit une solution permettant de gérer des modules Javascript, en respectant la norme des ES6 Modules. Il s'agit en règle générale d'une suite logicielle composée d'un utilitaire en ligne de commande et de fichiers à configurer.

L'un des plus célèbres Module Loader de l'écosystème Javascript est Webpack, ce dernier permet, comme son nom l'indique, de créer des paquetages à partir de fichier Javascript ( mais pas que ) et de les mettre à disposition sous forme de modules au sein d'une application front-end.

Pour plus de renseignements sur le fonctionnement de Webpack, rendez-vous sur le site officiel du produit à l'adresse suivante: https://webpack.github.io/docs/

# Ecrire un serveur avec node.js

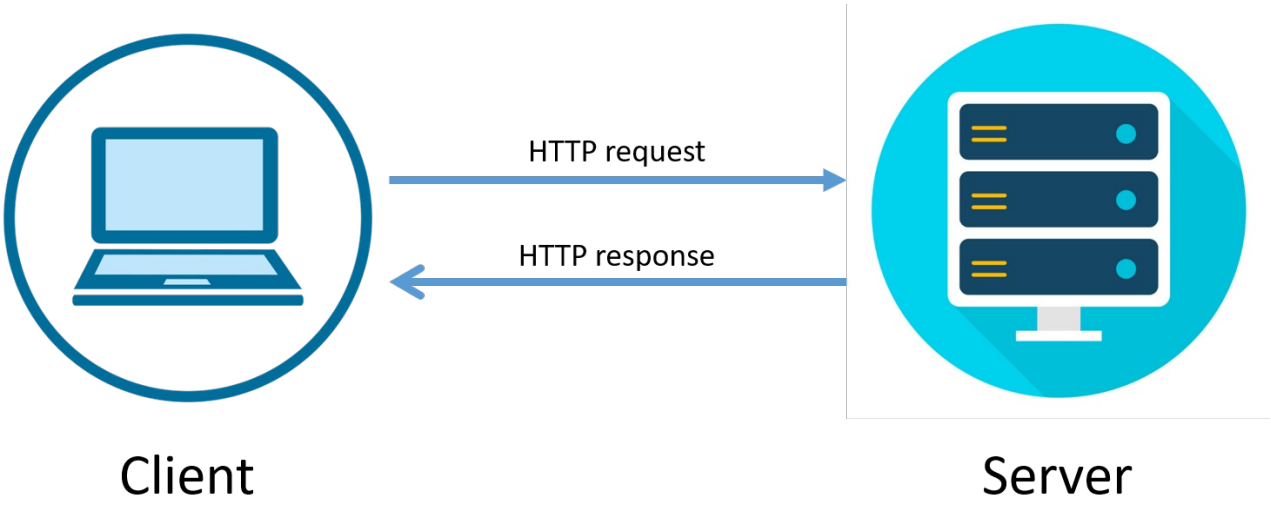
# Ecrire un serveur web avec node.js

Si node.js nous permet du code autre part qu'au sein d'un navigateur, c'est bien pour que l'on puisse écrire du code côté serveur. Il existe pas mal de façons de coder un serveur en node.js, nous allons donc nous intéresser à la façon de coder un serveur web comprenant la norme **http**.

Pour cela, il existe un module natif, nommé 'http', qui nous permet de créer facilement un serveur. Ce dernier va alors:

- Ecouter sur un port particulier
- Réceptionner les requêtes http entrantes
- Les interpréter
- Fournir au développeur un objet contenant les informations de la requête
- Fournir au développeur un objet permettant d'écrire une réponse au client
- Envoyer la réponse au client lorsque le développeur le demandera
- Fermer la requête

On peut donc constater qu'il s'agit d'un module fournissant des fonctionnalités de haut niveau, à aucun moment le développeur ne devra lui-même s'occuper d'interpréter / formater les requêtes entrantes et les réponses, le module s'en occupera pour nous. Notons aussi que ce module nous permet de créer un client http.



# Un serveur http basique

```
// on importe le module natif http
const http = require('http');

// puis on crée un serveur http à l'aide de la méthode dédiée
const server = http.createServer();

// On crée notre fonction de callback, celle qui va gérer nos
// requête entrante. Dans notre exemple, toutes nos requêtes
// renverront le message hello world au format html.
function requestHandler(request, response)
{
    // on écrit une réponse en précisant le type MIME (html)
    response.writeHead(200, {'Content-Type': 'text/html'});

    // on écrit le notre réponse
    response.write('<h1>hello world</h1>');

    // puis on l'envoie
    response.end();
}

// On définit un écouteur d'évènement à l'aide de la méthode 'on'
// ici, la fonction de callback s'éxécutera à chaque fois qu'une requête
// est envoyée au serveur.
server.on('request', requestHandler );

// Et on écoute sur le port 3000
server.listen(3000);
```

# Ecrire un serveur avec express

# Introduction à Express

Express est un framework web Node.js minimal et flexible qui fournit un ensemble robuste de fonctionnalités pour développer des applications Web et mobiles. Il facilite le développement rapide d'applications Web basées sur Node. Nous l'utilisons donc dans le but de créer un serveur http et lui envoyer des requêtes.

Express fournit une foule de méthodes utilitaires HTTP et de middlewares capables de rendre la création d'une API robuste, simple et rapide.

Il apporte également une couche fine de fonctionnalités d'application Web fondamentales, sans masquer les fonctionnalités de Node.js que vous connaissez et appréciez.

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

# Utiliser express generator

Il est possible de se faire générer son serveur express avec l'outil express generator.

```
# install express generator globally on your system
npm install express-generator -g

# then display all the options
express -h

#then generate another project named "myapp"
express --view=pug myapp

#go to myapp folder & install dependencies
cd myapp
npm install

# then define an env variable named DEBUG to "myapp"
# on windows:
set DEBUG=myapp:* & npm start

#on linux:
DEBUG=myapp:* npm start

#then, you can open your browser and go to http://localhost:3000
# your app structure should be like
.
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug

7 directories, 9 files
```

# Le routing avec express

Grâce au puissant et flexible système de routing d'express, vous pouvez définir finement la façon de répondre à n'importe quelle requête utilisateur, que ce soit en fonction du verbe http utilisé, de l'uri envoyé, ou encore des paramètres passés.

```
// une route GET
// Ce chemin de routage fera correspondre des demandes à la route racine, /.
app.get("/", function (req, res) {
  res.send("GET request to the homepage");
});

// une route POST
// Ce chemin de routage fera correspondre des demandes à la route racine, /.
app.post("/", function (req, res) {
  res.send("POST request to the homepage");
});

// Ce chemin de routage fera correspondre des demandes à /about.
app.get("/about", function (req, res) {
  res.send("about");
});

// Il s’agit d’exemples de chemins de routage basés sur des masques de chaîne.
// Ce chemin de routage fait correspondre acd et abcd.
app.get("/ab?cd", function (req, res) {
  res.send("ab?cd");
});
```

# Introduction à HAPI

# Introduction à HAPI

Hapi est un framework riche pour la création d'applications et de services qui permet aux développeurs de se concentrer sur l'écriture d'une logique d'application réutilisable au lieu de passer du temps à créer une infrastructure. Le vaste système de plug-ins de Hapi nous permet de créer, d'étendre et de composer rapidement des fonctionnalités spécifiques à la marque en plus de son architecture solide comme le roc.


```
'use strict';
import Hapi from 'hapi ';
// Create a server with a host and port
const server = new Hapi.Server();
server.connection({
  host: 'localhost',
  port: 8000
});

// Add the route
server.route({
  method: 'GET',
  path:'/hello',
  handler: (request, reply) => reply('hello world')
});

// Start the server
server.start(() => {
  console.log('Server running at:', server.info.uri);
});
```

# HAPI - les options

On peut démarrer HAPI avec plusieurs options:

```
// load one plugin
server.register(
    {
      register: require('myplugin'),
      options: {
        key: 'value'
      },
    },
    (err) => {
      if (err) {
      console.error('Failed to load plugin:', err);
      }
    }
  );

  // load multiple plugins
  server.register(
    [
      {
        register: require('myplugin'),
        options: {}
      },
      {
        register: require('yourplugin')
      }
    ],
    (err) => {
      if (err) {
        console.error('Failed to load a plugin:', err);
      }
    }
  );
```

# HAPI - routing & controllers 1/2

On peut définir des routes pour chaque verbe HTTP très facilement avec HAPI.

```
'use strict';

import Assets from './assets.controller';

module.exports = (server) => {
  server.route({
    method: 'GET',
    path: '/assets',
    handler: (request, reply, next) => {
      Assets.getAssetsByAttributes(request, reply, next);
    }
  });

  server.route({
    method: 'POST',
    path: '/assets',
    handler: (request, reply, next) => {
      Assets.create(request, reply, next);
    }
  });

  server.route({
    method: 'PUT',
    path: '/assets/{key}',
    handler: (request, reply, next) => {
      Assets.modify(request, reply, next);
    }
  });

  server.route({
    method: 'DELETE',
    path: '/assets/{key}',
    handler: (request, reply, next) => {
      Assets.remove(request, reply, next);
    }
  });
}
```

# HAPI - routing & controllers 2/2

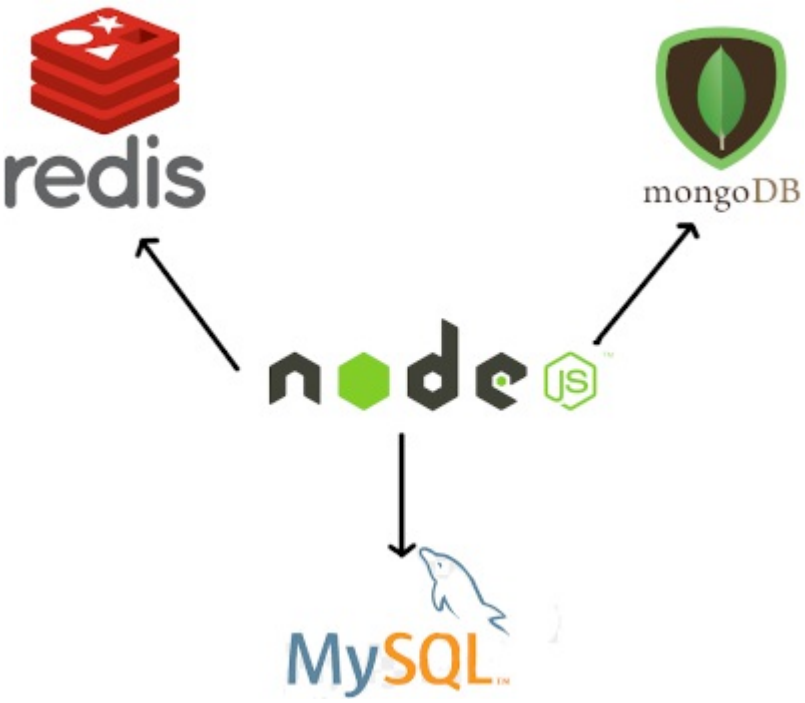
On peut définir des routes pour chaque verbe HTTP très facilement avec HAPI.

```
'use strict';
export const getAssetsByAttributes = (req, res, next) => res([]).code(200);
export const create = (req, res, next) => res({}).code(201);
export const modify = (req, res, next) => res({}).code(200);
export const remove = (req, res, next) => res({}).code(200);
```

# Les librairies d'accès aux bases de données

# Accéder à une base de données avec Node.js

En tant qu'environnement indépendant, node.js nous propose également l'accès à des logiciels tiers hébergé sur la même machine. Ce qui nous permet donc d'accéder à une base de données. Il n'existe cependant pas de module natif permettant d'accéder à une base de données en particulier, en effet, il va nous falloir installer des modules supplémentaires à l'aide de npm (node package manager).




La plupart du temps, les modules en question sont proposés et maintenus par les équipes en charge de la production et du maintien du SGBD choisi (Mysql, MongoDB, Redis ...). En ce qui nous concerne, nous allons nous pencher sur les solutions Redis et MongoDB.

# Le NoSQL avec MongoDB

Le système de gestion de base de données NoSQL le plus répandu est sans nul doute MongoDB. Après l'avoir installé sur votre serveur, vous devez ajouter les drivers par le biais de la commande **npm install mongodb**. Une fois les drivers installés, vous pouvez accéder à votre base de données et requêter des collections.

```
const { MongoClient } = require("mongodb");
const uri = "mongodb://localhost";
const client = new MongoClient(uri);
async function run() {
    try {
        await client.connect();
        const database = client.db('demo');

        // on récupère la collection
        const characters = database.collection('characters');

        // on va chercher l'ensemble des données de la collection
        const results = await characters.find();
        results.forEach( console.log );

        // on fait une requête spécifique en fonction du nom
        const query = { name: 'jon snow' };
        const character = await characters.findOne(query);
        console.log(character);

        // on insère une donnée
        await characters.insertOne({name: "Jean neige"});

        // supprimer une donnée
        await characters.deleteOne({name: "Jean neige"});

        // supprimer toutes les données
        await characters.deleteMany({name: "Jean neige"});

        // update un élément
        await characters.updateOne({name: "jon snow"}, {$set: {name: "John Snow"}});
    }
    finally {
        await client.close();
    }
}
run().catch(console.dir);
```

# Le NoSQL haute performance avec Redis

Redis est un gestionnaire de données open source (licence BSD) qui a pour particularité de tourner uniquement dans la RAM. Il est utilisé comme base de données, système de cache et 'message broker'. Redis fournit des structures de données telles que:

- Des chaînes de caractères
- Des hachages
- Des listes
- Des Sets
- Des SortedSets triés avec plages de requête
- Des bitmaps
- Des hyperloglogs
- Des index géospatiaux et des flux.

Redis intègre la réplication, les scripts Lua, l'expulsion LRU, les transactions et différents niveaux de persistance sur disque, et offre une haute disponibilité via Redis Sentinel et un partitionnement automatique avec Redis Cluster.

# La modélisation des bases NoSQL

De manière générale, les bases de données NoSQL s'articulent autour de quelques concepts fondamentaux:

**La paie clé/valeur**: Au sein d'une base NoSQL, les valeurs sont stockées et associées à ce que l'on appelle communément une clé. Bien qu'il s'agisse en règle générale d'une chaîne de caractère, il est tout à fait possible de créer des clés sous forme de nombre, ou, à contrario, utiliser des structures complexes comme clé.

**Le document**: Une valeur atomique (un entier, une chaîne de caractères) est un document; une paire clé-valeur est un document; un tableau de valeurs est un document; un agrégat de paires clé-valeur est un document; et de manière générale, toute composition des possibilités précédentes (un tableau d'agrégats de paires clé-valeur par exemple) est un document.

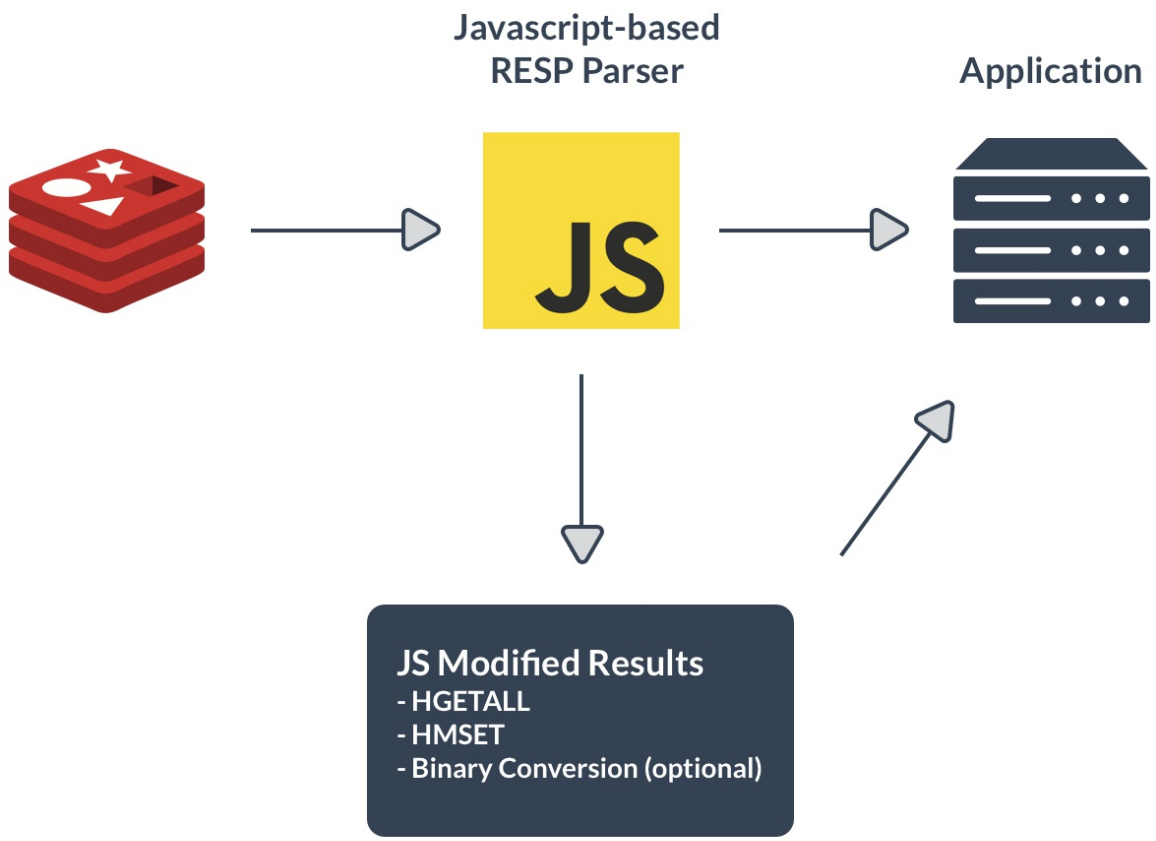
**La collection**: Une collection regroupe un ensemble de documents, ces derniers sont accessibles au sein de la collection et n'ont pas forcément besoin d'être stockés dans un ordre précis (à moins que la structure de la collection en elle-même ne le propose comme une fonctionnalité).

Il est à noter que bien que cela puisse apporter quelques problèmes, les différents documents d'une même collection n'ont aucunement l'obligation d'être semblable, aussi bien dans leur structure que dans l'ordre dans lequel ils déclarent leurs paires clés/valeurs. On appelle cela une organisation **Schema less**, ce qui signifie que:

- Les bases NoSQL se fondent sans schéma logique défini a priori.
- L'équivalent du CREATE TABLE en SQL n'est soit pas nécessaire, soit même pas possible ; on peut directement faire l'équivalent de INSERT INTO.
- Cela apporte de la souplesse et de la rapidité, mais se paye avec moins de contrôle et donc de cohérence des données.
- Toutefois le mouvement NoSQL tend à réintégrer des fonctions de schématisation a priori, à l'instar de ce qui se fait en XML : le schéma est optionnel, mais conseillé en contexte de contrôle de cohérence. **En gros, on ne stocke pas les choux et les patates dans la même collection.**

# Utiliser Redis sous Node.js

```
const { promisify } = require("util");
const redis = require("redis");
const client = redis.createClient();
const setAsync = promisify(client.set).bind(client);
const getAsync = promisify(client.get).bind(client);

async function run() {
  // Il est possible d'utiliser le module redis-json pour stocker des objets complexes
  // au sein de redis. En effet, ce dernier ne supporte à la base, que les structures
  // de données simples s'apparentant  à des chaînes de caractères.
  const data = { "name": "John Snow" };
  const stringified = JSON.stringify(data);

  // Par défaut, la librairie ne gère pas les promesses, mais nous pouvons
  // utiliser l'utilitaire promisify.
  await setAsync("characters:1", stringified);
  const results = await getAsync("characters:1");
  redis.print(results);
  client.end(true);
}

client.on("error", console.error );
run();
```


Javascript-based
RESP Parser
Application
JS
JS Modified Results
- HGETALL
- HMSET
- Binary Conversion (optional)

# Communication en temps réel

# Problématiques et définition

La communication en temps réel permet à deux process d'un même programme ( ou de programmes différents ) de communiquer par le biais de messages, et ce, sans faire appel à des opérations bloquantes. Les messages sont envoyés, directement au destinataire ou à un programme tiers qui fait alors office de gestionnaire de message.

Les messages sont alors consommés par le destinataire, et celui-ci peut choisir d'y répondre ou non sans limite de délai. Bien entendu une telle façon de communiquer doit être accompagnée d'une façon de programmer bien précise. Le javascript, de part sa nature asynchrone, répond très à ce défi, notamment grâce à sa gestion de l'asynchronicité et à son modèle évenementiel.

Il existe plusieurs technologies, natives ou non, nous permettant de mettre en place une communication en temps réel inter processus en javascript. Redis, RabbitMQ, Socket.IO ou encore les websockets seules sont autant de solutions nous permettant de répondre à nos besoins en la matière.

# Websockets et Socket.IO

Socket.IO est une bibliothèque JavaScript pour les applications Web en temps réel. Il permet une communication bidirectionnelle en temps réel entre les clients Web et les serveurs. Il comporte deux parties: un côté client bibliothèque qui fonctionne dans le navigateur , et un côté serveur bibliothèque pour Node.js . Les deux composants ont une API presque identique . Comme Node.js , il est piloté par les événements.

Socket.IO utilise principalement le protocole WebSocket avec l'interrogation comme option de secours, tout en fournissant la même interface. Bien qu'il puisse être utilisé comme un simple wrapper pour WebSocket, il fournit de nombreuses autres fonctionnalités, notamment la diffusion vers plusieurs sockets, le stockage des données associées à chaque client et les E / S asynchrones. Il peut être installé avec l' outil npm.

socketio client

```
const io = require("socket.io-client");
const socket = io("ws://localhost:3000");

function start(){
    socket.send("classic client")
    socket.emit("clientEvent", "custom client");
}

socket.on("connect", start);
socket.on("message", console.log);
socket.on("serverEvent", console.log);
```

socketio server

```
const io = require("socket.io")(3000);

function init(socket){
    socket.on("message", console.log );
    socket.send("classic server");

    socket.on("clientEvent", console.log );
    socket.emit("serverEvent", "custom server");
}

io.on("connection", init);
```

# Pub/Sub sous Redis

Subscriber.js

```
const redis = require("redis");

const subscriber = redis.createClient();
subscriber.on("subscribe", () => {
    console.log("waiting for messages from publisher");
});

// log channel and msg
subscriber.on(
    "message",
    (channel, msg) => {
        switch(msg){
            case "quit":
                subscriber.unsubscribe();
                subscriber.quit();
                break;

            default: console.log(channel, ":", msg);
        }
    }
);
subscriber.subscribe("private_channel");
```

Publisher.js

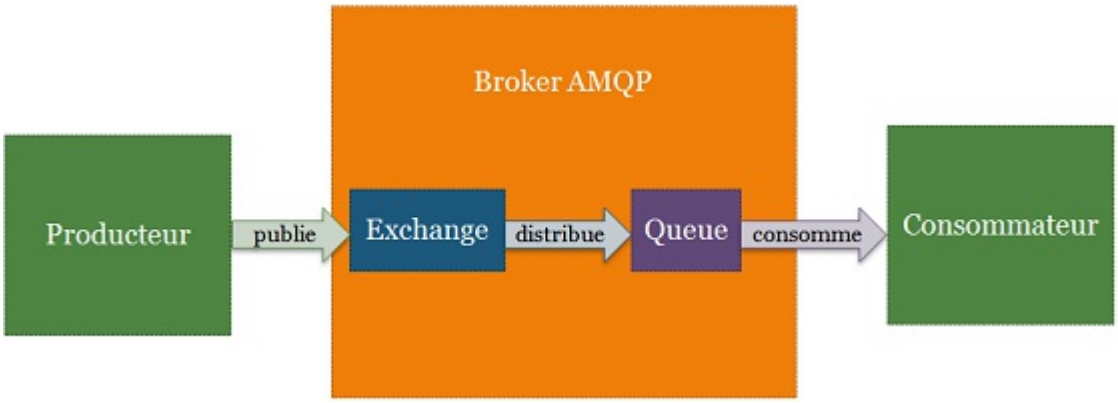
```
const redis = require("redis");

const publisher = redis.createClient();
publisher.publish("private_channel", "Hello I am the publisher");
publisher.publish("wrong_channel", "quit");
publisher.publish("private_channel", "quit");
publisher.quit();
```

# AMQP & Node.js

Il est impératif que les services puissent communiquer entre eux pour échanger des informations et transmettre des données. Contrairement à ce que l'on pourrait penser, c'est plus facile à dire qu'à faire, car les échanges entre différentes applications doivent surmonter plusieurs difficultés : le domaine informatique est lui aussi confronté aux barrières linguistiques (entre les différents langages de programmation) ce qui rend indispensable l'utilisation d'un protocole pour éviter que la communication ne se transforme en anarchie.

Une solution est offerte avec l'Advanced Message Queuing Protocol, abrégé en AMQP : un protocole global permettant de transporter les informations via un médiateur. AMQP résout plusieurs problèmes en même temps :

- Ce protocole garantit d'une part une transmission des données fiable (à l'aide d'un message broker).
- l'AMQP permet de stocker des messages dans des files d'attente, permettant ainsi une communication asynchrone : l'émetteur et le destinataire n'ont pas à agir au même rythme.
- Le destinataire (consommateur) du message n'est pas obligé d'accepter directement l'information, de la traiter et d'en accuser réception auprès de l'émetteur (producteur). À la place, il va récupérer le message dans la file d'attente lorsqu'il en a la capacité.
- Le producteur peut ainsi continuer à travailler sans créer de période d'inactivité.



# RabbitMQ & ZeroMQ

RabbitMQ est un message broker très complet et robuste, il repose sur le protocole AMQP. RabbitMQ est un message broker, son rôle est de transporter et router les messages depuis les publishers vers les consumers. Le broker utilise les exchanges et bindings pour savoir si il doit délivrer, ou non, le message dans la queue.

**Le message** est comme une requête HTTP, il contient des attributs ainsi qu'un payload. Parmi les attributs du protocol vous pouvez y ajouter des headers depuis votre publisher.

**Les bindings**, ce sont les règles que les exchanges utilisent pour déterminer à quelle queue il faut délivrer le message. Les différentes configurations peuvent utiliser la routing key (direct/topic exchanges) ainsi que les headers(header exchanges).

**Un exchange** est un routeur de message. Il existe différents types de routages définis par le type d'exchange.

**Une queue** est l'endroit où sont stockés les messages. Il existe des options de configuration afin de modifier leurs comportements.

**Le rôle du consumer** est d'exécuter un traitement après avoir récupéré un ou plusieurs messages. Pour ce faire il va réserver (prefetching) un ou plusieurs messages depuis la queue, avant d'exécuter un traitement. Généralement si le traitement s'est correctement déroulé le consumer va acquitter le message avec succès (basic.ack). En cas d'erreur le consumer peut également acquitter négativement le message (basic.nack). Si le message n'est pas acquitté, il restera à sa place dans la queue et sera re fetch un peu plus tard.

Voici le fonctionnement global du broker :

- Le publisher va envoyer un message dans un exchange.
- En fonction du binding, la couche exchange qui va router le message vers la ou les queues
- Ensuite un consumer va consommer les messages au sein des queues qui l'intéresse.

**ZeroMQ** est une bibliothèque de messagerie asynchrone hautes performances, destinée à être utilisée dans des applications distribuées ou simultanées. Il fournit une file d'attente de messages, mais contrairement au middleware orienté message, un système ZeroMQ peut fonctionner sans message broker dédié.

# Rabbit MQ example

sender.js

```
const amqp = require('amqplib/callback_api');

amqp.connect('amqp://localhost', function (error0, connection) {
    if (error0)
        throw error0;

    connection.createChannel(function (error1, channel) {
        if (error1)
            throw error1;

        channel.assertQueue('customQueue', { durable: false });
        channel.sendToQueue('customQueue', Buffer.from('Hello World!'));
    });
    setTimeout(
        () => {
            connection.close();
            process.exit(0);
        },
        500
    );
});
```

consumer.js

```
const amqp = require('amqplib/callback_api');
amqp.connect('amqp://localhost', function(error0, connection) {
    if (error0)
        throw error0;

    connection.createChannel(function(error1, channel) {
        if (error1)
            throw error1;

        const queue = 'customQueue';
        channel.assertQueue(queue, {durable: false});
        channel.consume(
            queue,
            (msg) => console.log(msg.content.toString()),
            {noAck: true}
        );
    });
});
```

# Les tests avec nodejs

# Les tests unitaires avec Mocha

Les tests font partie intégrante du développement de logiciels. Il est courant pour les programmeurs d'exécuter un code qui teste leur application lorsqu'ils apportent des modifications, afin de confirmer qu'elle se comporte comme ils le souhaitent. Avec la bonne configuration de test, ce processus peut même être automatisé, ce qui permet de gagner beaucoup de temps. L'exécution régulière de tests après l'écriture d'un nouveau code permet de s'assurer que les modifications ne cassent pas les fonctionnalités préexistantes. Cela permet d'accroître la confiance qu'ont les développeurs dans leur base de code, surtout lorsqu'elle est déployée en production pour que les utilisateurs puissent interagir avec elle.

Un framework de test structure la manière dont nous créons les cas de test. Mocha est un framework de test JavaScript populaire qui organise nos cas de test et les exécute pour nous. Cependant, Mocha ne vérifie pas le comportement de notre code. Pour comparer les valeurs dans un test, nous pouvons utiliser le module Node.js assert.

```
#installe les modules request et mocha
npm i request  mocha
```

increment.js

```
function increment(num){
    return num + 1;
}

module.exports = increment;
```

increment.spec.js

```
const increment = require('./increment');
const assert = require('assert').strict;
describe(
    "my test suite",
    () => {
        it(
            "should increment the number properly",
            () => {
                let num = 0;
                let result = increment(num);
                assert.strictEqual(result, 1);
            }
        )
    }
);
```

# Tests fonctionnels avec navigateur Headless

les tests fonctionnels permettent de vérifier le bon comportement d'une fonctionnalité. Ils permettent également de documenter le projet, une documentation qui sera toujours à jour, sinon les tests ne passent plus. Les tests fonctionnels sont la pour vérifier que le code répond bien aux spécifications du produit. Ces tests testent notre application comme une "boîte noire". Techniquement ils pourraient être développés dans un autre langage que le code à tester.

Avec javascript, il existe pléthore de bibliohtèques qui permettent de gérer ce type de tests. Toutefois il faudra les démarrer au sein d'un navigateur, le mieux est encore de les démarrer dans ce que l'on appelle un headless browser. Il existe une librairie très utile qui combine la possibilité de lancer un tel navigateur ainsi que la possibilité de le contrôler, son nom est **puppeteer.**

Puppeteer nous permet d'installer et de lancer une instance de chromium. Une fois lancée, une API nous permet de piloter cette instance, de sorte à ce que l'on puisse naviguer sur internet, récupérer le contenu des documents et nous avons même la possibilité de remplir des champs et de simuler les clicks utilisateurs. Parfait pour les tests fonctionnels ! Nous en profiterons au passage une librairie nous permettant d'opérer des assertions, comme **chai**. Voici la commande:

```
npm i puppeteer mocha chai
```

Et nous en profiterons pour rajouter le code suivant dans notre fichier package.json

```
{
  "scripts": {
    "test": "mocha --recursive './spec/*.spec.js' ",
    "serve": "http-server server"
  }
}
```

# Tests fonctionnels avec navigateur Headless

```
const { assert } = require("chai");
const puppeteer = require("puppeteer");

describe(
    'suite',

    () => {

        let browser = null;
        let page = null;

        it(
            'should contains google',
            async () => {
                const browser = await puppeteer.launch();
                const page = await browser.newPage();
                await page.goto('http://localhost:8080',  { waitUntil: 'networkidle2' } );
                const results = await page.evaluate("document.querySelector('h1').innerText");
                // Notre titre doit contenir la chaîne de caractères "Google"
                assert.include(results, "Google");
                await page.close();
                await browser.close();
            }
        )
    }
);
```

# Travaux pratiques

# TP: Server Side Events & Websockets

Le but de ce TP est de créer un tchat en temps réel à l'aide de Websockets. Le tchat devra comporter les fonctionnalités suivantes:

- Il doit y avoir un salon principal, où tout le monde peut s'exprimer.
- Il doit y avoir la possibilité de créer un salon privé entre deux utilisateurs.
- Il doit y avoir la possibilité de fermer un salon privé entre deux utilisateurs.
- BONUS: Il doit y avoir la possibilité d'envoyer des emojis

L'utilisation de socket.io est recommandée quoique non obligatoire. La durée maximale du TP est de 1 heure.

# Gestion de la performance avec javascript et node.js

# Bonnes pratiques Node.js

Javascript a bien grandi ces dernières années, à tel point qu'il est **l'un des langages les plus répandus sur github**. Le fait qu'il soit répandu est une chance, on peut facilement trouver de la documentation et de l'aide, mais le soucis c'est que l'on peut aussi trouver de tout et surtout du n'importe quoi. Il existe cependant de bonnes pratiques à garder en test lorsque l'on code en JS et principalement lorsqu'on est sous Node.js. Voici une liste de précieux conseils à toujours garder en tête:

- **Utiliser au maximum la parallélisation des tâches**. Node.js permet de lancer plusieurs processus en parralèle, et pour certaines tâches très lourdes, il est en effet préférable de les lancer dans un processus à part.
- **Utiliser le plus possible l'asynchronicité**. Si Node.js est aussi performant, c'est parce qu'il se base sur les principes de programmation asynchrone et événementielle. Son API non bloquante permet d'éviter les goulots d'étranglement et a pour conséquence de pouvoir traiter un plus grand nombre de demandes dans un laps de temps donné.
- **Si vous devez desservir des fichiers, préférez une compression gzip**.
- **Conservez le code le plus simple et le plus léger possible.**
- **N'utilisez pas Node.js pour desservir des fichiers statiques**: C'est une tâche que les serveurs webs plus classiques font sans problème, comme Apache. Ce dernier possède d'ailleurs nativement un système de cache paramétrable et sa couche C++ sera toujours plus rapide que votre couche écrite en Javascript si vous vous contentez simplement d'ouvrir et lire des fichiers afin d'en relayer le contenu.
- **Utilisez de préférence une solution de rendu client**: En effet, un serveur Node.js, comme n'importe quel serveur excelle dans la transmission / réception d'informations. Lui demander d'opérer le rendu d'un template est une charge bien souvent inutile à lui confier, en plus de rendre le fond dépendant de la forme. Bref, pensez web API.
- **Utilisez des outils de monitoring tel que Monitis ou Clinic.js**: Ces outils sont indispensables à tout projet conçu pour durer au moins sur le moyen terme et avec des exigences de performances accrues.

# Le Worker Pool et les tâches lourdes

Node.js utilise deux types de threads: un thread principal géré par une boucle d'événements et plusieurs threads auxiliaires dans le Worker Pool

La Event Loop est le mécanisme qui prend les fonctions de callback et les enregistre pour être exécutés à un moment donné dans le futur. Il fonctionne dans le même thread que le code JavaScript approprié. Lorsqu'une opération JavaScript bloque le thread, la boucle d'événements est également bloquée.

Le Worker Pool est un modèle d'exécution qui génère et gère des threads séparés, qui exécutent ensuite la tâche de manière synchrone et renvoit le résultat à la Event Loop. Cette dernière exécute alors le callback fourni avec ledit résultat.

En bref, il prend en charge les opérations d'E / S asynchrones, principalement les interactions avec le disque et le réseau du système. Il est principalement utilisé par des modules tels que fs (I / O-heavy) ou crypto (CPU-heavy). Le Worker Pool est implémenté dans libuv, ce qui entraîne un léger retard chaque fois que Node a besoin de communiquer en interne entre JavaScript et C ++, mais cela est à peine perceptible.

master.js

```
const { Worker, workerData } = require('worker_threads');
const worker = new Worker('./worker.js', { workerData: 0 });

worker.on('error', console.error);
worker.on('message', console.log);
worker.on('exit', (code) => process.exit(code));

worker.postMessage({type: "multiply", payload: 10});
worker.postMessage({type: "exit", payload: null});
```

worker.js

```
const { parentPort, workerData } = require('worker_threads');
parentPort.postMessage(workerData + 1);
parentPort.on(
   "message",
   (action) => {
      switch( action.type ){
         case "multiply" : parentPort.postMessage(action.payload * 2);  break;
         case "exit"     : process.exit(0);
      }
   }
);
```

# Les clusters

Les application Node.js sont par nature mono-threadées, or les serveurs, de nos jours, sont presque* toujours multi-core. Pour exploiter l'ensemble des capacités de ces serveurs, il est nécessaire de pouvoir exploiter tous les cores.Pour cela, on peut lancer une application Node.js en mode cluster

Dans l'idéal, il faut lancer autant d'instances qu'il y a de cores sur la machine. Cela permet de partager au mieux la puissance de la machine entre les différentes instances sans pour autant dégrader les performances en partageant les cores entre plusieurs instances.

Le module cluster, bien qu'ayant été pendant longtemps expérimental, est aujourd'hui utilisé de façon très large. Son principe est simple, lorsqu'une application est lancée en mode cluster, un premier process est démarré en mode master. Le process master n'a pas pour rôle de traiter les requêtes entrantes à proprement parler, mais plutôt de les dispatcher aux process forkés qui eux sont dédiés au traitement des requêtes.

Ils sont lancés dans le mode worker. La responsabilité de l'instanciation de forks en mode worker est du ressort du process master, et les règles de fork sont laissées à la responsabilité du développeur. Tout au long de la vie du cluster, des événements sont générés aussi bien par le master que par les workers. Il est important de s'y abonner pour être capable de réagir à des changements dans le cluster (Crash d'un worker, par exemple).

# Utiliser tous les coeurs de votre CPU avec les clusters

```
const cluster = require("cluster");
const http = require("http");
const numCPUs = require("os").cpus().length;

if (cluster.isMaster) {
    for( let i = 0; i < numCPUs; i++ ) {
        const proc = cluster.fork();
        proc.send(i);
    }

    cluster.on("exit", function (worker, code, signal) {
        console.log("worker " + worker.process.pid + " died");
    });
}
else {
    var answer = "Hello World from random process";
    process.on(
        'message',
        (msg) => {answer = "Hello world from process number" + msg;}
    );

    http.createServer(
        (req, res) => {
            res.writeHead(200);
            res.end(answer);
            // on déclenche une panne exprès
            process.exit(1);
        }
    ).listen(666);
}
```

# Gestion de la mémoire : la pile et la mémoire totale, comment les gérer

# Gestion de la mémoire / éviter les fuites mémoires

Dans toute application, la gestion de la mémoire est au centre des préoccupations des développeurs, et ça n'est pas pour rien! En effet, un code instable aura tôt fait d'engendrer bon nombre de problèmes et se dirigera doucement vers le plantage le plus lamentable. On peut penser par exemple à un dépassement de la taille maximale de la pile d'appel. La pile d'appel, c'est cette partie de la mémoire réservée à l'appel de vos fonctions, car oui à chaque fois que vous appelez une fonction, de la mémoire est consommée, ne serait-ce que pour mémoriser l'endroit où vous vous trouviez au sein du code source au moment où vous avez appelé ladite fonction. Il y a tout un tas de choses à mémoriser, et l'espace dédié à cela est la pile d'appel (plus exactement en fonction des opérations, l'espace dédié est une pile ou une file mais nous ne rentrerons pas dans les détails ici).

Il est donc assez facile de dépasser cette mémoire allouée, qui n'a jamais eu cette erreur: 'maximum call stack exceeded' ? Il s'agit tout simplement d'une erreur qui arrive lorsque vous gérez mal votre récursivité! Càd que vous appellez trop de fonctions dans des fonctions dans des fonctions dans des fonctions dans des ... Pour éviter cela, vous pouvez tout simplement **mieux penser vos algorithmes, paramétrer une profondeur de récursivité maximale ou encore envisager une version itérative de votre algorithme** (ce qui est souvent bien plus rapide car in fine on économise les opérations sur la pile).

Mais il existe un autre type de problème de mémoire bien plus fréquent, j'ai nommé **la fuite mémoire**. La fuite mémoire est vicieuse en cela qu'elle ne fera pas planter votre directement, mais au bout d'un certain temps. Elle est donc parfois compliquée à traquer. Sa cause est pourtant toute simple, **vous avez oublié de libérer de la mémoire**. Imaginez un garage, celui-ci possède un espace limité, vous employez une personne afin qu'elle range vos affaires dans ce garage et qu'elle le range tous les jours. Afin de distinguer quelles sont les choses qu'elle peut enlever du garage, vous indiquez à la personne d'enlever uniquement les objets portant une étiquette. Et bien la fuite mémoire, c'est vous qui oubliez de poser des étiquettes sur vos vieux débris inutiles, et votre employé c'est le garbage collector. Ne sachant que faire de ces vieux débris, il ne peut pas les supprimer et du coup ils continuent de prendre de la place dans le garage.

Afin d'éviter cela, il y a une technique très simple: **Dès que vous ne vous servez plus d'une valeur, assignez la valeur 'null' à la variable qui la contient, ainsi qu'à l'ensemble des variables pointant vers cette valeur**. J'insiste sur ce dernier point car c'est souvent là que ça coince !

# Outils de profiling Node.js

Il est possible de traquer les fuites mémoires, la consommation CPU, les bugs avec des outils dédiés, au sein d'un script Node.js. Il suffit pour cela de démarrer notre script avec l'option **node --inspect-brk myscript.js** et d'ouvrir le panneau de debug de **chrome dev tools** (nouel onglet + chrome:inspect). Il est également possible de lancer le profiler intégré à visual studio code avec l'option create debug terminal. Et pour finir, il existe des outils dédiés, parfois payants, comme clinic.js qui permettent d'avoir une représentation graphique de la mémoire et des objets, ainsi que de leur évolution au cours du temps.