

PLAN



LOGISTIQUE

- Horaires
- Déjeuner & pauses
- Autres questions ?







RAPPEL DES BONNES PRATIQUES JAVASCRIPT



PLAN



HISTORIQUE DU LANGAGE

Date	Éditeur	Évènement
Déc. 1995	Sun/Netscape	Annonce de JavaScript (ancien LiveScript)
Mars 1996	Netscape	JavaScript dans Netscape 2.0
Août 1996	Microsoft	Sortie de JScript dans Internet Explorer 3.0
Nov. 1996	Netscape	Standardisation de JavaScript à l'ECMA
Juin 1997	ECMA	Adoption de l'ECMAScript
1998	Adobe	ActionScript

- L'**ECMA** est un organisme privé européen de standardisation
- Il n'est pas spécialisé dans l'IT (plutôt l'électronique)



ECMASCRIPT

Les versions du standard **ECMAScript**

Ver	Date	Évolution
1	Juin 1997	Adoption de l'ECMAScript 1
2	Juin 1998	Réécriture de la norme, première version du JavaScript comme on le connaît
3	Déc. 1999	RegExp, Try/Catch, Erreur, ...
4	Abandonnée	
5	Déc. 2009	Clarifie beaucoup d'ambiguïtés de la V3 Version actuellement dans Node.js et également la plus répandu
6+	2015	Plusieurs nouveaux concepts Nom de code Harmony Désormais une version par an au mois de juin



INSTRUCTIONS ET POINTS-VIRGULES

- La structure des instructions est héritée du C
- Les instructions sont délimitées par des points-virgules ;
- Les points virgules sont **optionnels**
 - Il y a un système d'insertion automatique
 - <https://bclary.com/log/2004/11/07/#a-7.9>
 - Schématiquement, s'il n'y a pas d'ambiguïté et un retour chariot, un point-virgule est ajouté automatiquement
- Peut amener à des erreurs si on comprend mal ce principe
- L'usage des **linters** et des outils de build à réduits le besoin d'être explicite sur les points-virgules



TYPES DE DONNÉES

ECMAScript manipule différents types de données :

- `null`
- `undefined`
- `boolean` (`true` ou `false`)
- `number`

Décimal	Hexa.	Octal	Binaire
<code>100, 100.0, 1e2</code>	<code>0x64</code>	<code>0144, 0o144</code>	<code>0b1100100</code>

- `string` ("Hello", 'World')
- `objet` ({} , { clef: "valeur" })
- `symbol` (ES2015)



COERCITION

Coercion est un gros mot qui signifie que la VM JavaScript va tenter de changer automatiquement le type des données pour réaliser les opérations demandées

- Rend possible le mélange de données de types différents
- **Une des plus grandes sources d'erreurs en JavaScript**

```
[] == false //-> true  
null == false //-> false  
Number(true) //-> 1
```

- Pour gérer ce problème, JavaScript propose l'opérateur **====**
- Comme **==** mais avec une vérification de type en plus

```
| 1 === '1' //-> false
```



OBJETS STANDARDS

- Array, Boolean, Number, String, Date, RegExp, Math, Function

```
// Notation littéral :  
/* Array */ let a = [1, 2, 3]  
/* Boolean */ let b = true; let b = false;  
/* Number */ let n = 0xEA  
/* String */ let s = "Essai"  
  
// Instantiation d'objet obligatoire  
let d = new Date()  
  
// A voir en fonction des besoins  
let r = new RegExp('^.{3}$')  
r = /^.{3}$/  
  
// Math ne contient que des méthodes et propriétés statiques  
Math.PI; Math.random();  
  
// Les fonctions sont des objets un peu spéciaux  
let add = function fnName(a, b) {return a + b};  
add.name; //-> 'fnName'
```



STRUCTURATION DE CODE : IF

- La structure `if ... else`

```
| if /* test */ {  
| /* liste d'instructions */  
| } else {  
| /* autres instructions */  
| }
```

- Opérateur ternaire

```
| const a = /* test */ ? /* true */ : /* false */;
```

- Quelque soit le type retour du test il est évalué comme un booléen, les valeurs "falsy" sont: `false`, `null`, `undefined`, `0`, `""`, `NaN`

- Opérateurs booléen

```
| const a = /* Exp. truthy */ && /* Exp. falsy */ || /* Expression */
```

- ***Attention : Les opérateurs booléens retournent la valeur de la dernière expressions valide sans en changer le type.***



STRUCTURATION DE CODE : SWITCH

- le `switch / case`

```
switch /* valeur */ {  
    case /* valeur 1 */:  
        /* Liste d'instructions */  
        /* Attention au fall through */  
    case /* valeur 2 */:  
        /* Liste d'instructions */  
        break;  
    case /* valeur x */:  
        /* Liste d'instructions */  
        break;  
    default:  
        /* Liste d'instructions */  
}
```

- Il est possible de faire un `switch` sur les `String`



STRUCTURATION DE CODE : WHILE

- `while`

```
while /* test */ {  
/* liste d'instructions */  
}
```

- `do while`

```
do {  
/* liste d'instructions */  
} while /* test */;
```

- Toujours faire attention à l'évaluation de la valeur du test en booléen



STRUCTURATION DE CODE : FOR

- `for`

```
for (let i = 0; i < 5; i++) {  
    console.log('The number is', i);  
}
```

- `for ... in` !

- Permet d'itérer sur les propriétés d'un objet
- Permet d'itérer sur les clés d'un tableau

```
const object = { prop1: 1, prop2: 2 };  
  
for (const prop in object){  
    console.log('Property', prop, '=', object[prop]);  
}
```



STRUCTURATION DE CODE : RUPTURE DE FLUX

Il existe plusieurs solutions pour changer le flux du programme

- `continue` : continue la boucle avec le prochain élément
- `break` : sort de la boucle et continue le code
- `return` : sort de la fonction et continue le code
- `throw` : sort du traitement classique et remonte un problème de fonctionnement



LES VARIABLES

- Il n'y a pas de contrainte sur la longueur
- Un identifiant JavaScript doit commencer par une lettre, \$ ou _
- Les caractères qui suivent peuvent également être des chiffres
- Il est possible d'utiliser des lettres Unicode
Souvent déconseillé pour éviter des incompatibilité avec les outils qui transforment et manipule le code

```
// Ça va être compliqué de saisir ces noms de constantes
// avec un clavier francophone ou anglophone :
const π = Math.PI
const φ = Math.PHI = (1 + Math.sqrt(5)) / 2
```

- Il ne doit pas correspondre à un mot clef réservé
- Plusieurs syntaxes pour définir des variables : var, let et const

```
var maVariable1;
let maVariable2 = 'maValeur';
const maVariable3, maVariable4 = 'maValeur';
```



LES VARIABLES

- Les variables sont typées dynamiquement

```
let a = 5; // number  
a = 'essai'; // string  
a = { clef: 'valeur' }; // object
```

- Utiliser une variable qui n'a pas été définie avec un mot clé `var`, `let` ou `const`
 - En lecture, on obtient une exception
 - En écriture :
 - en mode laxiste : on définit une **variable globale**
 - en mode strict : on obtient une exception
 - L'utilisation des variables globales est à éviter

```
console.log(b); //-> Uncaught ReferenceError: b is not defined  
b = 5; // création de la variable globale b  
console.log(b); //-> 5
```



LES FONCTIONS : DÉFINITION

- Une fonction est un ensemble d'instructions
 - Une fonction **peut** être appelée avec des arguments
 - Une fonction retourne une valeur (par défaut **undefined**)
- Définition d'une fonction
 - le mot-clé **function**
 - un nom optionnel
 - une liste de paramètres entre parenthèses (peut être vide)
 - un corps entre accolades (peut être vide)

```
function nomDeLaFonction (parametre1, parametre2) {  
    /* instructions */  
}
```



LES FONCTIONS : DÉFINITION

- Nom de la fonction
 - Les noms de fonctions fonctionnent comme les noms de variables
 - Une fonction sans nom est dite anonyme
 - La propriété `name` d'une fonction donne son nom
- Les fonctions sont des objets
 - Il est possible de les affecter à des variables

```
const uneVariable = function nomDeLaFonction (parametre) {  
    /* instructions */  
};
```



LES FONCTIONS : LA SYNTAXE FLÉCHÉ

- ES2015 introduit une nouvelle syntaxe pour les fonctions
 - On utilise `=>` à la place de `function` (mais pas au même endroit)

```
const nomDeLaFonction = (parametre1, parametre2) => {  
    /* instructions */  
}
```

- Principal différence avec les fonctions classique
 - `this` est attaché statiquement au contexte de création de la fonction
 - `arguments` n'existe pas (on doit utiliser le **spread operator** à la place)
- On l'utilise surtout pour créer des fonctions anonymes à la volé :



LES FONCTIONS : EXEMPLES

- Déclarer une fonction nommée crée également une référence du même nom

```
// Fonction nommée "foo"  
function foo () { /**/ }
```

```
// Variable pointant sur une fonction existante  
const fooVar = foo;
```

```
// Variable pointant sur une nouvelle fonction nommée  
const barVar = function bar () { /**/ }
```

```
// Variable pointant sur une nouvelle fonction anonyme  
const bazVar = function () { /**/ }
```

```
// Noms des fonctions  
foo.name === 'foo'  
fooVar.name === 'foo'  
barVar.name === 'bar'  
bazVar.name === 'bazVar' // !?
```



LES FONCTIONS : ARGUMENTS

- Pour appeler une fonction il faut disposer d'une référence
- L'appel de la fonction se fait avec les parenthèses
- Il est possible de lui passer autant d'arguments que souhaité

```
let foo = function bar(arg1, arg2) {  
    console.log(arg1, arg2);  
}  
  
console.log(foo, bar); //-> ReferenceError: bar is not defined  
foo(); //-> undefined, undefined  
foo(1, 'exemple') //-> 1 exemple  
foo(1, 2, 3, 4) //-> 1 2  
  
// arguments est un mot clé retournant tous les arguments  
foo = function() { console.log(arguments); }  
  
foo(1, 2, 3, 4) //-> [1, 2, 3, 4]
```



LES FONCTIONS : PROGRAMMATION FONCTIONNELLE

- Il est possible de passer une variable contenant une fonction en argument à une autre fonction
- Cela permet de passer un comportement en paramètre
- C'est le point de départ de la **Programmation Fonctionnelle**

```
function pourChaque(tableau, action) {  
    for(index in tableau){  
        action(tableau[index]);  
    }  
}  
pourChaque([1,2,3,5,8], console.log);
```

- **⚠ La fonction est passée sans parenthèse !**
- Il s'agit de la référence sur la fonction qui est passée



VISIBILITÉ DES VARIABLES ET DES FONCTIONS



VISIBILITÉ DES VARIABLES ET DES FONCTIONS

```
function scope() {  
    // Forward reference  
    var valeur1 = foo();  
    function foo() {  
        return 42;  
    }  
    var valeur2 = foo();  
  
    // Création des variables  
    console.log('avant a', a); //-> Undefined  
    console.log('sans déclaration de b', b); //-> Reference Error  
    var a = 2;  
    console.log('apres a', a); //-> 2  
  
    // Manipulation des scopes  
    if (true) {  
        var banane = 'banane';  
        let orange = "orange"  
    }  
    console.log(banane);
```



LES CLOSURES

- Closure signifie **fermeture lexicale**
 - Le principe est de capturer un scope
 - Et le rendre disponible pour une autre fonction
 - Ce principe s'applique à la déclaration d'une fonction
 - Le scope courant est alors capturé

```
function foo() {  
  let bar = 'value';  
  
  return function log () {}  
    // bar a été capturé et est visible ici par "log"  
    console.log(bar)  
}  
  
const log = foo()  
log() // On appelle log hors du scope de "foo" mais "bar" est bien affiché.
```



LES CLOSURES : PIÈGES

- Les closures peuvent sembler très naturelles
- Elles sont très utilisées en JavaScript
- Mais attention aux pièges
- Il s'agit de la **référence** (pointeur) qui est capturée
- Les données ne sont pas copiées dans la nouvelle fonction

```
const a = ['elem1', 'elem2', 'elem3', 'elem4', 'elem5'];

for(var i = 0; i < 3; i++) {
  setTimeout(() => {
    console.log(a[i]);
  }, 1000);
}

//-> Après 1s, on obtient : elem4 elem4 elem4
```



LES OBJETS

- JavaScript est un langage **orienté objet** à **prototype**
 - http://fr.wikipedia.org/wiki/Programmation_orientée_prototype
 - Les objets ont une notion de prototype
 - Un prototype est **aussi** un objet (possibilité de chaînage)
 - Un objet accède de façon transparente à son prototype
- Les objets ont des **propriétés** de n'importe quel type
 - Les **propriétés** qui ont comme valeur une fonction sont généralement appelées **méthodes**
 - Le nom d'une propriété est appelé **clé**
- Depuis ES2015, on peut créer des objets avec une syntaxe de classes



LES OBJETS : MODIFICATION DES PROPRIÉTÉS

- Il est possible d'assigner des valeurs aux propriétés
- Mais aussi d'en ajouter et d'en supprimer

```
const o = {}

const x = {
  prop0: 'initial'
}

o.nom = 'essai'

x.prop1 = function () {
  console.log('hello')
}

delete x.prop0

console.log(o) //-> { nom: 'essai' }
console.log(x) //-> { prop1: function }
```



LES OBJETS : THIS

- Au sein d'une **fonction**, on dispose du mot clé **this**
- Le comportement est dépendant de la manière dont la fonction est appelée
 - Hors contexte (appel simple) :
 - En mode laxiste : **this** = window (browser) ou global (Node)
 - En mode strict : **this** = **undefined**
 - En contexte (constructeur, objet avec syntaxe à point) : **this** = l'objet
- Attention à ce piège :

```
const dandy = {  
  name: 'Georges Abitbol',  
  say: function () {  
    console.log("L'homme le plus classe du monde est " + this.name)  
  }  
}
```



LES OBJETS : THIS

- Il est possible de forcer le contexte (le `this`)
 - En utilisant `call` ou `apply`

```
| fn.call(ctx, arg1, arg2);
| fn.apply(ctx, [arg1, arg2]);
```
 - En créant une nouvelle fonction proxy :

```
| var newFn = fn.bind(ctx);
```
- Au sein d'une fonction fléché, `this` correspond toujours au contexte de création de la fonction, comme si on avait utilisé `bind(this)`.



LES OBJETS : THIS

```
global.test = 'global';

function log() {
  console.log(this.test);
}

const objet = {
  test: 'objet',
  log: log
}

log(); //-> global
objet.log(); //-> objet

log.call(objet); //-> objet
log.apply(objet); //-> objet

const proxy = log.bind(objet);

log(); //-> global
proxy(); //-> objet
```



LES OBJETS : CONSTRUCTEURS

- Pour définir un objet (**singleton**)
 - Utiliser la syntaxe littérale { }

```
| let monObjet = { property: "field1" };
```
- Utiliser une fonction comme constructeur avec le mot clé **new**
 - Dans le constructeur, **this** représente alors l'objet
 - Toute fonction peut être utilisée comme constructeur
- (ES2015) Utiliser la syntaxe des classes



LES OBJETS : CONSTRUCTEURS

```
function Contact () {  
    this.nom = '';  
    this.prenom = '';  
}  
  
Contact.prototype.toString = function () {  
    return this.prenom + ' ' + this.nom;  
}  
  
let c = new Contact();  
  
class Contact {  
    constructor () {  
        this.nom = '';  
        this.prenom = '';  
    }  
  
    toString () {  
        return this.prenom + ' ' + this.nom;  
    }  
}  
  
let c = new Contact();
```



LES OBJETS : HÉRITAGE

```
class Parent {  
    toString () {  
        return this.prenom + ' ' + this.nom;  
    }  
}  
  
class Contact extends Parent {  
    constructor () {  
        super();  
        this.nom = 'nom';  
        this.prenom = 'prenom';  
    }  
}  
  
let contact = new Contact();  
console.log(contact.toString());
```





LODASH

L
O


- Librairie qui sert à la fois côté client et côté serveur
- Elle propose des fonctions pratiques au niveau JavaScript
 - Des **helpers** pour des opérations récurrentes
 - Des outils de **programmation fonctionnelle**
- On l'affecte d'habitude à la variable `_`



LODASH : LES UTILITAIRES

- Pour les objets

- `keys`, `values` : retourne les clés ou les valeurs

```
| _.keys({one: 1, two: 2, three: 3});  
| //-> ["one", "two", "three"]
```

- `clone (objet)` : effectue une copie superficielle

```
| _.clone({nom: 'Doe'});  
| //-> {nom: 'Doe'};
```

- `assignIn (destination, sources)` :
copie les propriétés des sources vers la destination

```
| _.assignIn({prenom: 'John', nom: 'moe'}, {nom: 'Doe', age: 50});  
| //-> {prenom: "John", nom: "Doe", age: 50}
```



LODASH : LES UTILITAIRES

- Les tests de type `is*`
 - Certains tests sont plus compliqués qu'il n'y paraît
 - Comparaison en profondeur
 - `isEqual(obj1, obj2)`
 - Cas particuliers du JavaScript
 - `isUndefined`, `isNull`, `isNaN`
 - Les types de base
 - `isArray`, `isObject`, `isFunction`, `isString`,
`isNumber`, `isBoolean`, `isDate`, `isRegExp`



LODASH : LA PROGRAMMATION FONCTIONNELLE

- Qu'est ce que la programmation fonctionnelle ?
 - Privilégie l'évaluation de fonctions
 - Évite le changement d'état et la mutation des données
- Intéressant pour le traitement des collections
- Utilisation intensive des fonctions en arguments
- Vocabulaire
 - **predicate** : fonction qui répond `true` ou `false`
 - **iterator** : fonction exécutée sur chaque élément d'une collection



LODASH : PREDICATES

- `find(liste, predat)` :
retourne le premier élément correspondant au prédicat
- `find(liste, objet)` :
retourne l'élément qui contient les mêmes propriétés
- `filter(liste, predat)` :
retourne tous les éléments correspondant au prédicat
- `reject` : inverse du filter
- `every, some` :
retourne vrai si un / tous les éléments sont conformes
- `partition` : retourne deux listes

Et bien d'autres...



LODASH : LES ITERATORS

- `each(liste, iterator)` :
exécute l'itérateur pour chaque élément
- `map(liste, iterator)` :
exécute l'itérateur pour chaque élément
retourne la liste des résultats de l'itérateur
- `reduce(liste, iterator, memo)` :
réduit une liste pour ne retourner qu'une seule valeur en utilisant memo
pour stocker le résultat du reduce précédent

```
let sum = _.reduce([1, 2, 3], function(memo, num){  
  return memo + num;  
}, 0); //-> 6
```

Le reduce est une notion complexe mais fondamentale de la programmation fonctionnelle



LODASH : LE CHAÎNAGE

- Le chaînage n'est pas possible par défaut dans Lodash
- Il est possible de l'activer explicitement

```
let stooges = [
  {name: 'curly', age: 25},
  {name: 'moe', age: 21},
  {name: 'larry', age: 23}
];

let youngest = _.chain(stooges)
  .sortBy(function(stooge){ return stooge.age; })
  .map(function(stooge){
    return stooge.name + ' is ' + stooge.age;
  })
  .first()
  .value();

//-> "moe is 21"
```



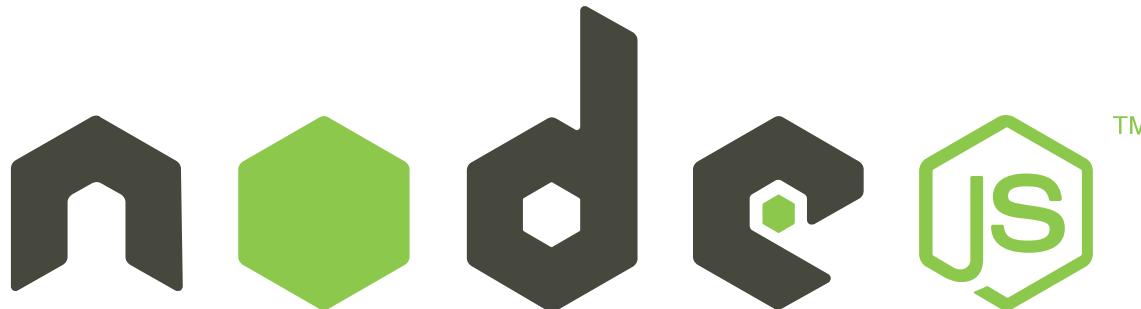




Lab 1



INTRODUCTION À NODE.JS



PLAN



NODE.JS : LES ORIGINES

- Node.js est une plateforme basée sur un moteur JavaScript
- Il a été créé par **Ryan Dahl**
- L'idée initiale était de créer un site Web avec une possibilité de faire du push
- Le serveur est écrit en C, il voulait ajouter un langage de script
- Il lui fallait un système d'I/O asynchrone
- Il a choisi JavaScript car il n'avait aucune API d'I/O
- Il a développé sa propre API non bloquante



NODE.JS : LES ORIGINES

- Le projet a été lancé en 2009
 - La première version a été publiée par **Dahl** en 2011
 - Il est rapidement sponsorisé par la société **Joyent**
- Aujourd'hui
 - Le projet est sur GitHub à l'adresse :
<https://github.com/nodejs/node>
 - L'équipe projet est composée d'un groupe d'une dizaine de personnes et de plus de 500 committers
 - La version stable actuelle est la 14
 - La version mature actuelle est la 12
 - Le projet est sous licence X11



NODE.JS : VERSIONING

- Node.js utilise un versioning sur 3 nombres
 - MAJOR.MINOR.PATCH
 - La numérotation suit la norme **semver** depuis la 4.0.0
- La norme **semver** détaille quand changer de versions majeure, mineure, et patch
 - La version majeure est incrémentée dès que la rétrocompatibilité de l'API publique est cassée
 - La version mineure est incrémentée lors de l'ajout d'une nouvelle fonctionnalité et que celle-ci conserve la rétrocompatibilité de l'API publique
 - La version de patch est incrémentée lors de la correction d'un bug





- V8 est une machine virtuelle JavaScript Open-Source
- Elle est développée en C++ par Google
- C'est le moteur JavaScript de **Chrome**
- Lancée en 2008 avec la première version de Chrome



V8

- V8 est compilée pour différents OS et architectures processeur
- Elle est publiée sous licence BSD
- Le code JavaScript est compilé et optimisé à l'exécution
- Tout est fait pour améliorer les performances
 - Le moteur compile le JavaScript de manière native pour la plate-forme cible (IA-32, x86-64, ARM, ou MIPS ISAs)
 - Elle intègre une gestion de l'allocation mémoire, un Garbage Collector générationnel et du inline caching
- Elle permet d'exécuter du code C++ extérieur



V8 : JAVASCRIPT

- V8 prend en charge ES2019
- Elle implémente 100% des fonctionnalités d'ES2019 dans la version 12
- L'utilisation de V8 pour Node.js
 - Assure l'utilisation d'un moteur très performant
 - Une version avancée du langage JavaScript
 - Un seul moteur d'interprétation
- Des problèmes de compatibilité peuvent tout de même exister
 - Avec les différentes versions de Node.js
 - Avec le système sur lequel il tourne



ASYNCHRONISME

- La particularité fondatrice de Node.js est l'**asynchronisme**
 - Dans Node.js toute opération faisant accès à un périphérique est **non-bloquante**
 - Non-bloquante signifie que l'exécution du code continue sans attendre le résultat
 - Un mécanisme de callback permet de réaliser des traitements lorsque les données sont prêtes
 - Le programme n'est jamais en **attente** d'une entrée/sortie
- Les **I/O** non bloquantes existent dans d'autres langages mais ne sont jamais aussi centrales



ASYNCHRONISME : DÉVELOPPEMENT

- La programmation asynchrone diffère de la programmation classique (séquentielle)
 - Il est impossible de programmer séquentiellement
 - Il faut découper le programme en plusieurs fonctions
 - Les traitements sont réalisés suite à des évènements
- Les évènements sont gérés par le cœur de Node.js
- Exemples d'évènements : fin de lecture d'un fichier, retour d'une requête HTTP, arrivée d'une requête HTTP...
- On passe un **callback** en argument pour qu'il soit appelé à la réception du résultat de notre traitement



NODE.JS : LES USAGES

La souplesse de Node.js permet de l'utiliser pour de nombreux usages

- Script : Type commande Shell
 - npm, Gulp, Eslint, Karma, Mocha...
- Serveur Web : un classique
- Robotique (Node-RED, NodeBots, NodeCopter, Rosnodejs)
- NodeOS
- tessel.io
- JS.everywhere()
- Et vous ?



ECOSYSTÈME

- La puissance de Node.js est aussi dûe à son écosystème
 - ***Intégralement*** Open-Source
 - Extrêmement complet et varié
 - Très réactif
- Le repository officiel : npmjs.org
 - Affiche des notes de popularité, de maintenance et de qualité
 - Peut faire des suggestions en fonction de thème (CLI, Web server, etc.)
- npms.io propose une UI différente
- Il existe plus de 1 300 000 packages
- Bien sûr, parmi tous ces packages, la qualité diffère !



ECOSYSTÈME

- Utilitaires

- Lodash
- Winston
- Date-fns
- Q

- Web

- Axios
- Express
- NestJS
- Hapi

- Test

- Jest
- Mocha
- Chai
- Cucumber

- Build

- Webpack
- Babel
- Nodemon
- Gulp



INSTALLATION

- Installation standard
 - Aller sur le site nodejs.org
 - Cliquer sur **INSTALL**
 - Le téléchargement se lance en fonction de la plateforme
 - Windows : `msi`, Mac : `pkg`, Linux : `tar.gz`
 - Suivre la procédure en fonction de la plateforme

```
$ node -v  
v12.17.0  
$ node  
> console.log('hello world');  
hello world  
> .exit  
$
```



INSTALLATION

- Installation depuis les binaires
 - Cliquer sur **Downloads** plutôt qu'**Install**
 - Décompresser l'archive
 - Ajouter le répertoire `/bin` dans le `PATH`
- Installation depuis les sources
 - `git clone https://github.com/nodejs/node.git`
 - `./configure, make, sudo make install`
- Installation depuis les systèmes de paquets
 - Linux : **Installing from package managers**
 - Mac : `brew install node`



INSTALLATION

- Installation de plusieurs version de Node.js
 - Mac & Linux : [NVM](#)
 - Windows : [NVM for Windows](#)
 - Si vous travaillez à plusieurs sur un projet dans des environnements hétérogènes : [docker-node](#)
- Ce sont les solutions idéales si vous devez travailler avec plusieurs versions de Node.js, Ce sont également les meilleures solutions pour gérer la montée en version de votre environnement Node.js tout en préservant votre environnement de travail.



LE PREMIER SCRIPT

- Créer un fichier `helloworld.js`

```
| console.log('hello world');
```

- Lancer le programme avec `node`

```
| $ node helloworld.js  
hello world
```

- Le rendre auto exécutable

- Ajouter au fichier

```
| #!/usr/bin/env node
```

- Lancer les commandes

```
| $ chmod +x helloworld.js  
$ ./helloworld.js  
hello world
```







Lab 2



ARCHITECTURE DE NODE.JS





PLAN

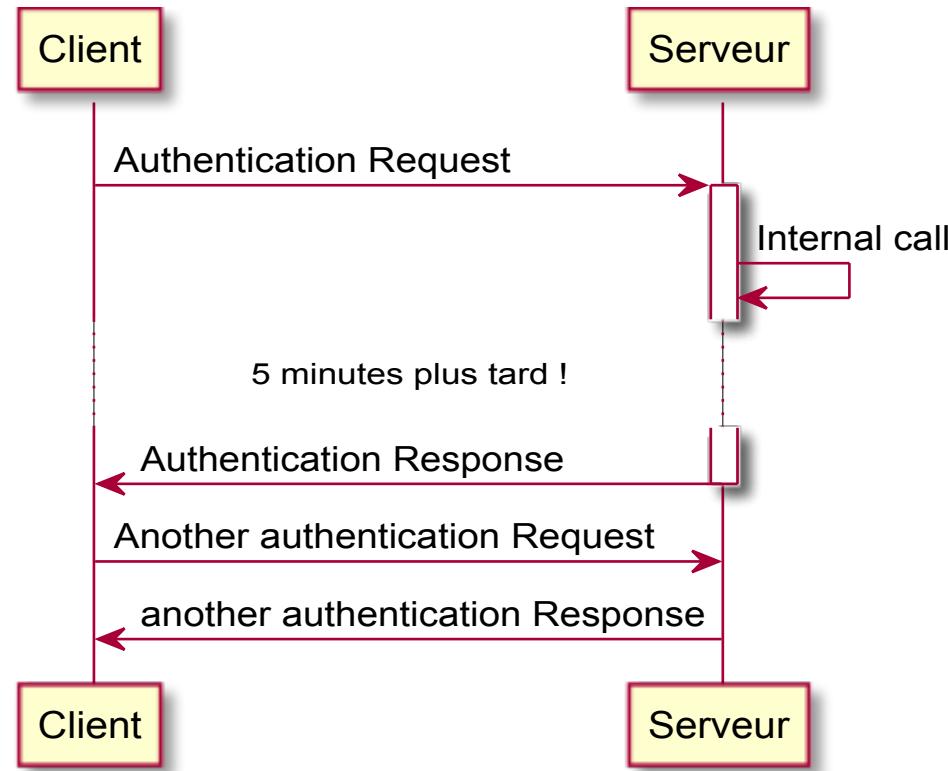


ASYNCHRONISME

- La majorité des programmes sont écrits de manière séquentielle
- **La plupart** des autres langages / plateformes / frameworks
 - Incitent à programmer séquentiellement
 - Multiplient les threads pour augmenter les performances
 - Ce qui crée des problèmes d'accès concurrents
 - Peuvent utiliser des I/O asynchrones mais le font rarement

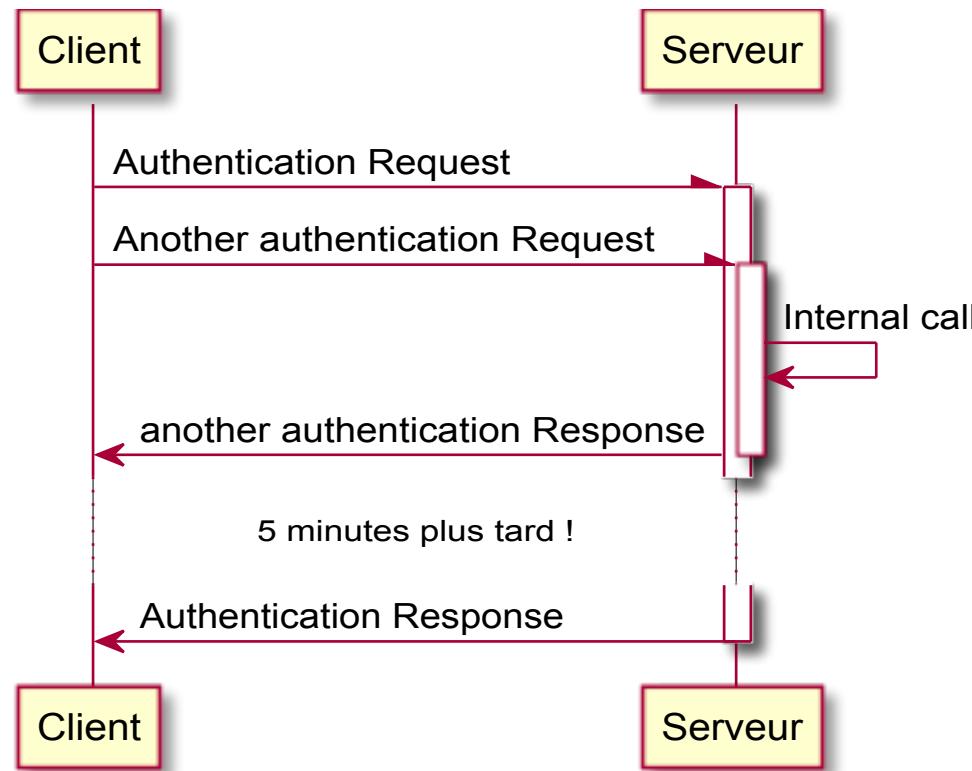


ASYNCHRONISME : EXEMPLE



ASYNCHRONISME : EXEMPLE

- Deux appels traités de façon **asynchrone** :



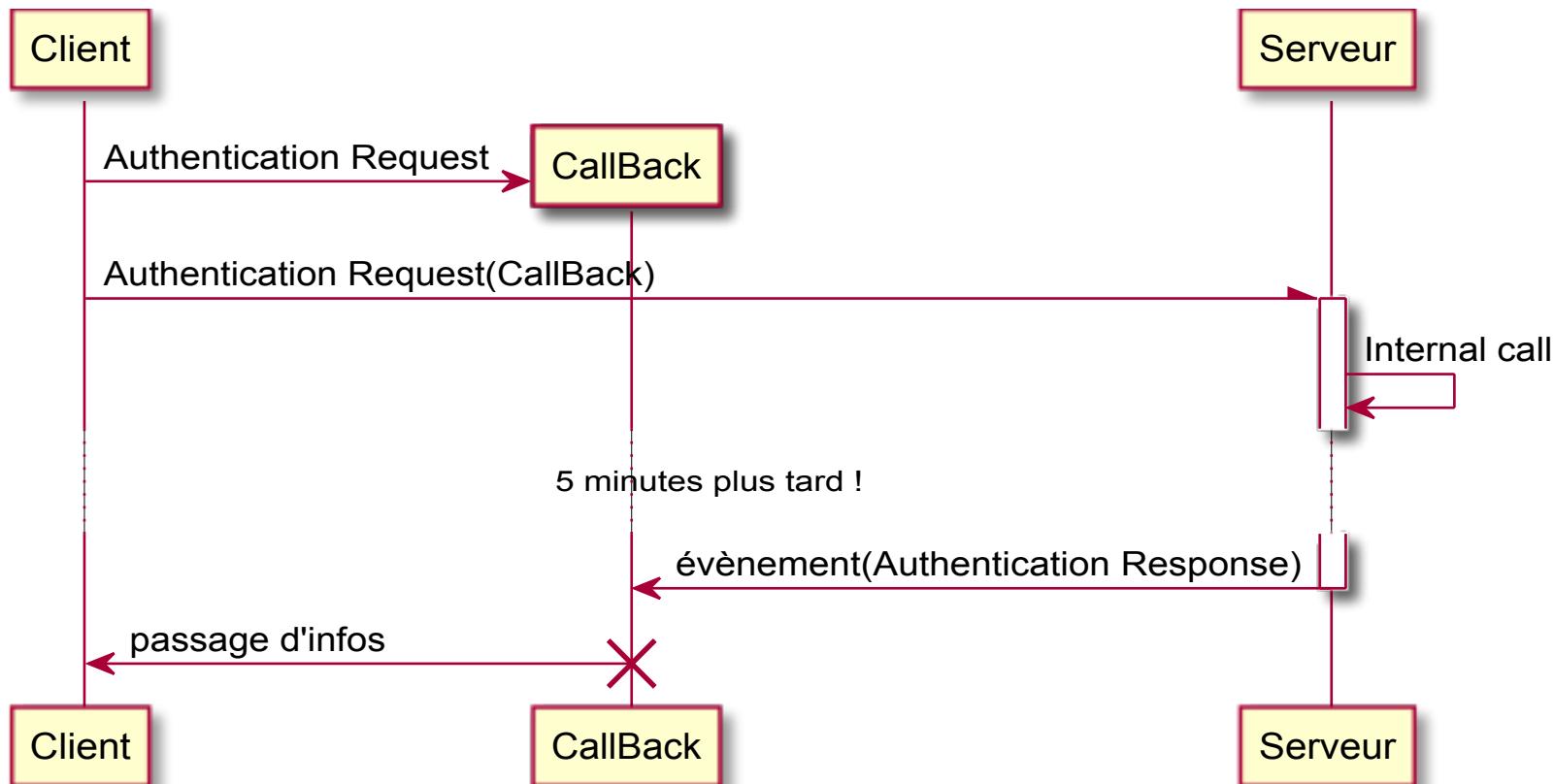
ASYNCHRONISME : COMPLEXITÉ

- La programmation asynchrone est très impactante
- Les traitements ne peuvent pas *retourner la réponse*
- Il faut réaliser les enchaînements dans des **callbacks**
- Ces callbacks seront exécutées *plus tard*
- Le contexte de l'application aura alors peut-être changé
- Heureusement, la nature **fonctionnelle** du JavaScript aide beaucoup pour l'écriture du code



ASYNCHRONISME : COMPLEXITÉ

- La fonction utilisée comme callback a son propre contexte



ASYNCHRONISME : EXEMPLE

- De manière synchrone (à éviter absolument !)

```
const file = fs.readFileSync('essai.json')

console.log(file)
//-> '[ { "nom": "Doe", "prenom": "John" } ]'
```

- De manière asynchrone

```
const file = fs.readFile('essai.json', (err, data) => {
  console.log('async', data)
})

console.log('sync', file)
//-> sync undefined
//-> async '[ { "nom": "Doe", "prenom": "John" } ]'
```



CALLBACKS : DÉFINITION

- Fonction qui sera déclenchée sur un évènement
- Il peut s'agir de n'importe quelle référence à une fonction
- Des paramètres lui seront passés pour traiter l'évènement
- Par convention dans Node.js
 - Le premier argument correspond à une éventuelle erreur
 - Les autres arguments sont les données de l'évènement

```
const callback = (err, result) => {
  if (err) {
    return console.error(err.message)
  }

  console.log(result)
}
```

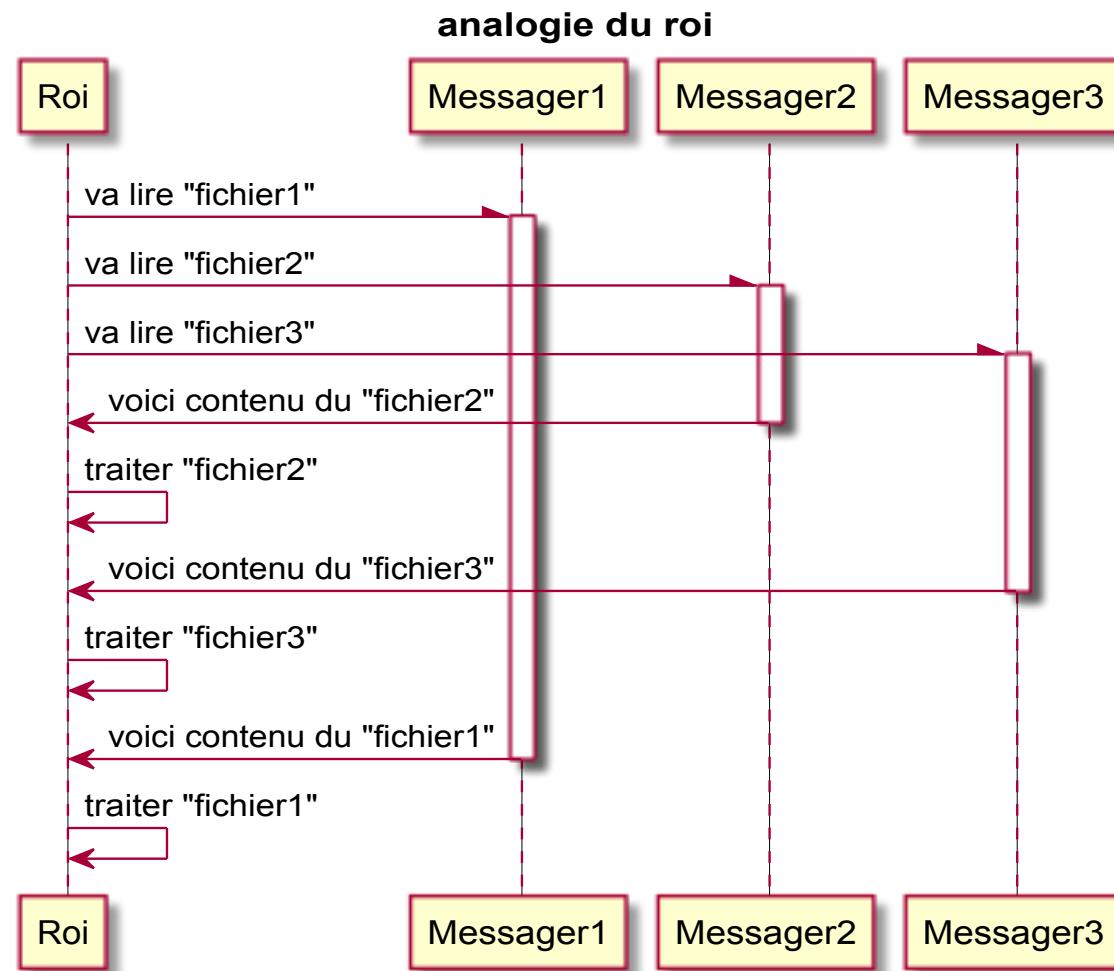


CALLBACKS

- Les APIs de Node.js ont toujours cette structure
 - | `module.action(/*[args*]*/, callback)`
- Le plus souvent, la callback est définie sous forme de fonction anonyme dans l'appel du module
 - | `module.action(/*[args*]*/, (error, result) => {
 /* contenu du callback */
})`
- Attention à cette syntaxe, car elle ne détermine pas du tout l'ordre d'exécution du code
- Le code situé sous l'appel à l'action peut être exécuté tout aussi bien **avant** qu'après le contenu du callback



L'ANALOGIE DU ROI

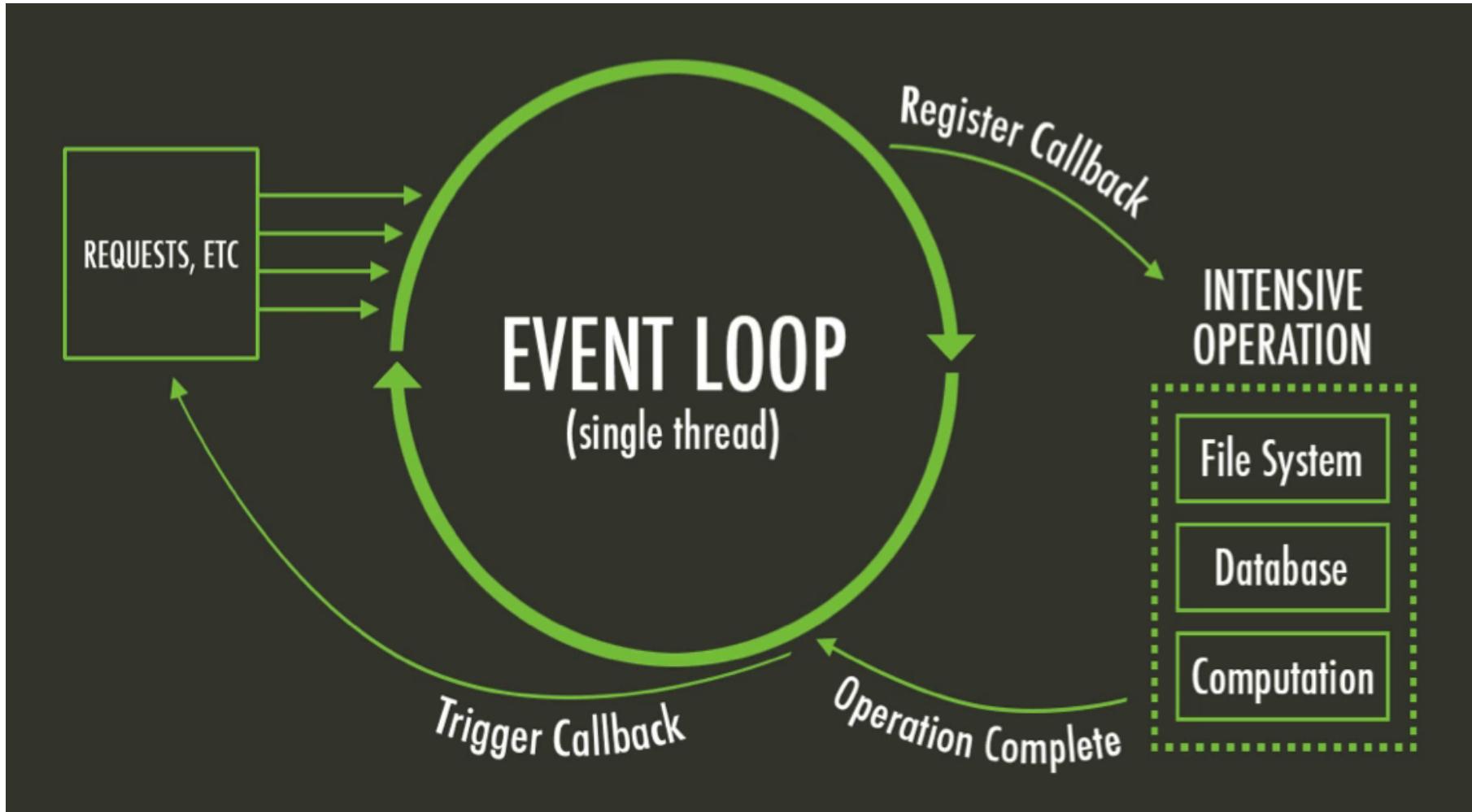


L'EVENT LOOP

- Le moteur JavaScript n'utilise qu'**un seul thread**
- Il n'est donc jamais utile de s'inquiéter des accès concurrents
- Par contre la gestion des I/O en interne utilise des threads
 - Une pile FIFO des callbacks à traiter
 - Le Single Thread Event Loop qui dépile les callbacks
 - Traitements : réalisation des demandes d'I/O
 - Les traitements terminés ajoutent un élément dans la pile



L'EVENT LOOP







MODULES ET GESTION DES DÉPENDANCES



PLAN



LES MODULES

- Chaque fichier `.js` dans Node.js est un module
- Node.js utilise l'API require/exports
- On distinguera
 - Les modules internes qui sont les fichiers `.js` qui composent l'application
 - Les modules externes qui sont des librairies téléchargées
- Mais au final, un module externe est un fichier `.js` également



LES MODULES: ECMASCIRIPT

- Le langage JavaScript (ES2015) apporte la notion de module (expérimentale)
- Le chargement des modules ES est asynchrone (synchrone supporté)
- L'extension des fichiers change vers `.mjs`
- L'API `import/export` peut importer des modules CJS (`.js`)
- Un module CJS peut importer un module ES en utilisant `await import()`



LES MODULES : RÈGLES

- L'écosystème des modules Node.js s'est construit sur des principes forts qu'il faut connaître et respecter
 - **Zéro passe plat** : les fonctions doivent toujours réaliser quelque chose, sinon, il faut donner un accès direct à la donnée
 - **KISS (*Keep It Simple, Stupid* ou *Keep It Stupid Simple*)** : chaque module doit réaliser une fonctionnalité identifiée, quitte à créer des modules extrêmement simples
 - **Toute action doit être explicite** : le code ne doit pas prendre d'initiative, tout comportement doit être piloté explicitement



NPM

- Node.js inclut un système de gestion des paquets : **npm**
- Il existe pratiquement depuis la création de Node.js
- C'est un canal important pour la diffusion des modules





Lab 3

NPM INSTALL

- `npm` est un outil en ligne de commande (écrit avec Node.js)
- Il permet de télécharger les modules disponibles sur [npmjs.org](https://www.npmjs.org)
- Les commandes les plus courantes :
 - `install` : télécharge le module et le place dans le répertoire courant dans `./node_modules`
 - `install -g` : installation globale, le module est placé dans le répertoire d'installation de Node.js
Permet de rendre accessible des commandes
 - `update` : met à jour un module déjà installé
 - `remove` : supprime le module du projet



NPM INIT

- `npm` gère également la description du projet
- Un module Node.js est un (ou plusieurs) script(s)
- Le fichier de configuration se nomme `package.json`
- `npm` permet également de manipuler le module courant
 - `init` : initialise un fichier `package.json`
 - `init --scope=@my-username` : permet de créer un projet scopé
- `docs <moduleName>` : ouvre la documentation du module
- `install <moduleName>` suivi de `--save` ou `--save-dev` :
 - Comme install mais référence automatiquement la dépendance dans le `package.json`



PACKAGE.JSON

- npm se base sur un fichier descripteur du projet
- package.json décrit précisément le module
- On y trouve différents types d'informations
 - Identification
 - name : l'identifiant du module (unique, url safe)
 - version : doit respecter node-semver
 - Description : description, authors, ...
 - Dépendances : dependencies, devDependencies, ...
 - Cycle de vie : scripts main, test, ...



PACKAGE.JSON : DESCRIPTION

- `description` : la description détaillée
- `author` : les détails sur l'auteur du module
 - Une personne est définie avec `name`, `email`, `url`
- `contributors` : la liste des contributeurs
- `homepage` : la page web du projet
- `repository` : la référence vers le système de version
 - `type`, `url`, par défaut **Git** et même **GitHub**
 - `loginGitHub/nomDuRepo` peut suffire
- <https://docs.npmjs.com/files/package.json>



PACKAGE.JSON : DÉPENDANCES

- `dependencies`

La liste des dépendances nécessaires à l'exécution

- `devDependencies`

Les dépendances pour les développements (build, test...)

- `peerDependencies`

Les dépendances nécessaires au bon fonctionnement du module (liste de compatibilité), mais pas installées lors d'un `npm install`

- `optionalDependencies` (**rare**)

Des dépendances qui ne sont pas indispensables à l'utilisation du module, prend en compte que la récupération peut échouer

- `bundledDependencies` (**rare**)

Des dépendances qui sont publiées et livrées avec le module



PACKAGE.JSON : VERSIONS

- Les modules doivent suivre la norme **semver**
 - Structure : `MAJOR.MINOR.PATCH`
 - `MAJOR` : Changements d'API incompatibles
 - `MINOR` : Ajout de fonctionnalités rétrocompatibles
 - `PATCH` : Corrections de bugs
- Pour spécifier la version d'une dépendance
 - `version` : doit être exactement cette version
 - `~, ^` : approximativement, compatible
 - `major.minor.x` : `x` fait office de joker
 - **Et bien d'autres** : `>`, `<`, `>=`, `min-max...`

