



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js



Logistique

- Horaires
- Déjeuner & pauses
- Autres questions ?



Rappel des bonnes pratiques JavaScript



Plan

- *Rappel des bonnes pratiques JavaScript*
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js

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	Décembre 1999	RegExp, Try/Catch, Erreur, ... Version la plus répandue
4	Abandonnée	
5	Décembre 2009	Clarifie beaucoup d'ambiguïtés de la V3 Version actuellement dans Node.js
6	2015	Plusieurs nouveaux concepts Nom de code Harmony

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
 - <http://bclary.com/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
- La plupart des règles de bonne conduite demandent à toujours mettre le point-virgule

Types de données

ECMAScript manipule différents types de données :

- Booléen : `true` ou `false`
- Nombre
 - Entier décimal : `0`, `5`, `-50`, `77`
 - Entier octal ou hexa. : `077 => 63`, `0xA => 10`
 - Réel : `1000.566`, `.034`, `2.75e-2`
- Chaîne : `""`, `"Hello"`, `'World'`
- Tableau : `[]`, `[1, 2, 3]`, `[true, 'hello', 1, {}]`
- Map/Objet : `{ }`, `{ clef: "valeur" }`,
`{ "clef": {}, 'autre clef': [] }`

Coercition

Coercition 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 sans coercition automatique

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

Objets standards

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

```
let a = new Array(1, 3, 2); a.sort().reverse();

let b = new Boolean(); b.valueOf();

let n = new Number('5'); n.toLocaleString();

let d = new Date(); d.getTime();

let s = new String('Essai'); s.toUpperCase();

let r = new RegExp('^.{3}$'); /*ou*/ r = /^.{3}$/;
r.test('abcd'); r.exec('abc');

Math.PI; Math.random();

let add = function fnName(a, b) {return a + b};
add.name; //-> 'fnName'
```



Structuration de code : if

- Le `if`

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

- Opérateur ternaire

```
var a = /* test */ ? /* true */ : /* false */;
```

- Attention, la valeur du test est **convertie** en booléen

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 la conversion 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 object`

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

- **Attention au `for in array`**

- Préférer :

```
let values = [0, 1, 2, 3, 5, 8, 13, 21, 34, 55];  
values.forEach(function(value, keyIndex, array){  
  console.log('Index', keyIndex, ', Value', value);  
});
```

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 dans les règles de bonne conduite
- Il ne doit pas correspondre à un mot clef réservé
- Plusieurs syntaxes pour définir des variables : `var`, `let` et `const`

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

Les variables

- Les variables ont un typage dynamique

```
let a = 5;  
a = 'essai';  
a = { clef: 'valeur' };
```

- 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, on définit une **variable globale**
 - 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
- 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

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

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
let fooVar = foo;
// Variable pointant sur une nouvelle fonction nommée
let barVar = function bar() { /**/ }
// Variable pointant sur une nouvelle fonction anonyme
let bazVar = function() { /**/ }

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

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

- La limite dans laquelle une référence est visible est un **scope**
- Visibilité des symboles au sein d'un scope
 - **Fonctions nommées** : utilisable partout dans le scope même avant la déclaration (**forward-reference**)
 - **Variables locales** : le symbole existe partout dans le scope sa valeur est `undefined` jusqu'à l'initialisation
 - **Variables globales** : le symbole devient une propriété du scope par défaut (`window` dans un navigateur, `global` pour node.js)
- Avec `var`, les scopes sont délimités par les corps des fonctions
 - Les blocs `if` / `for` / `while` ne créent pas de scope
 - **⚠ Différent de la plupart des langages**
 - Préférer `let` et `const`

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);  
    console.log(orange); //-> Reference Error  
}
```

Les closures

- Closure signifie fermeture
 - 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

```
let bar = 'value';
function foo() {
  // bar a été capturé et est visible ici par closure
  console.log(bar)
}
```

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(function() {
        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 ES6, il propose une écriture des objets sous forme de classes
 - La rétrocompatibilité est assurée

Les objets : modification des propriétés

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

```
let o = new Object();

var 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
 - Appel simple : `this` = window (browser) ou global (Node)
 - Propriété d'un objet : `this` = l'objet
 - Constructeur (`new`) : `this` = l'objet créé

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);
```

Les objets : this

```
global.test = 'global';

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

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

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

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

let proxy = log.bind(objet);

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



Les objets : constructeurs

- Pour définir un objet
 - 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
- (ES6) Utiliser la syntaxe des classes

Les objets : constructeurs

```
function Contact() {  
    this.nom = '';  
    this.premon = '';  
    this.toString = function() {  
        return this.premon + ' ' +this.nom;  
    }  
}  
  
let c = new Contact();
```

```
class Contact {  
  
    constructor() {  
        this.nom = '';  
        this.premon = '';  
    }  
  
    toString() {  
        return this.premon + ' ' +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



- 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, predicat)` :
retourne le premier élément correspondant au prédicat
 - `find(liste, objet)` :
retourne l'élément qui contient les mêmes propriétés
 - `filter(liste, predicat)` :
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

- Rappel des bonnes pratiques JavaScript
- *Introduction à Node.js*
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js

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 7
 - La version mature actuelle est la 6
 - Le projet est sous licence MIT

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 ECMAScript 5
- Elle implémente 99% des fonctionnalités d'ECMAScript 6
- 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 (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
- nodejsmodules.org ajoute des tags et une **popularité**
- Il existe plus de 450 000 packages
- Bien sûr, parmi tous ces packages, la qualité diffère !

Ecosystème

- Utilitaires
 - Lodash
 - Winston
 - Date-fns
 - Q
- Web
 - Request
 - Express
 - Pug
 - Hapi
- Test
 - Mocha
 - Jest
 - Chai
 - Cucumber
- Build
 - Gulp
 - Webpack
 - Babel
 - Nodemon

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
v8.2.1
$ 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`

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



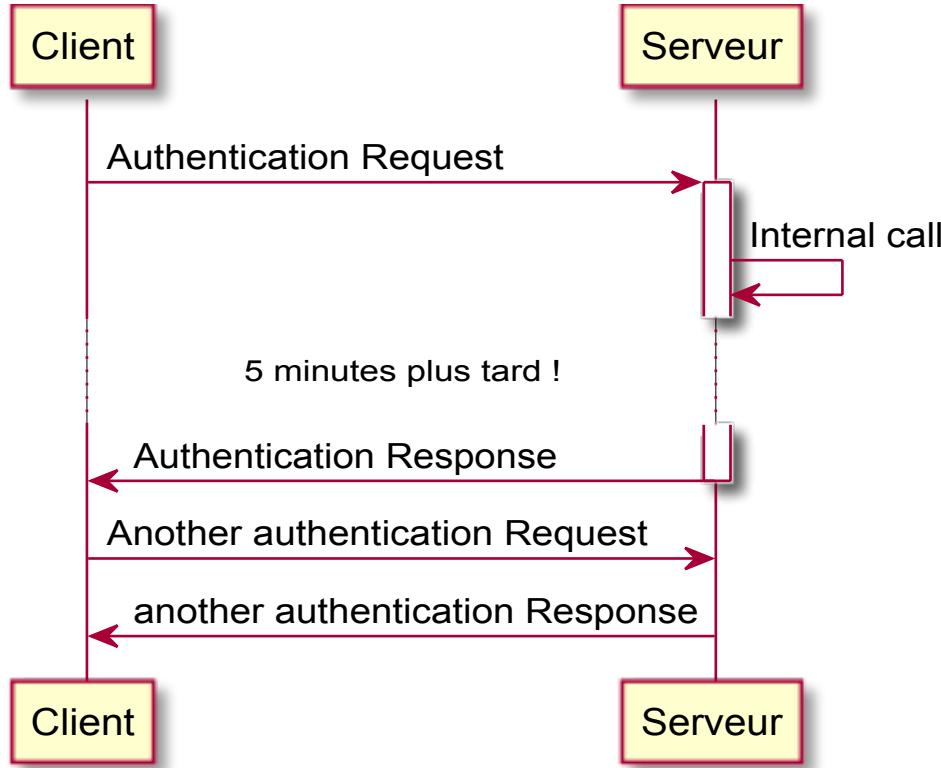
Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- *Architecture de Node.js*
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js

Asynchronisme

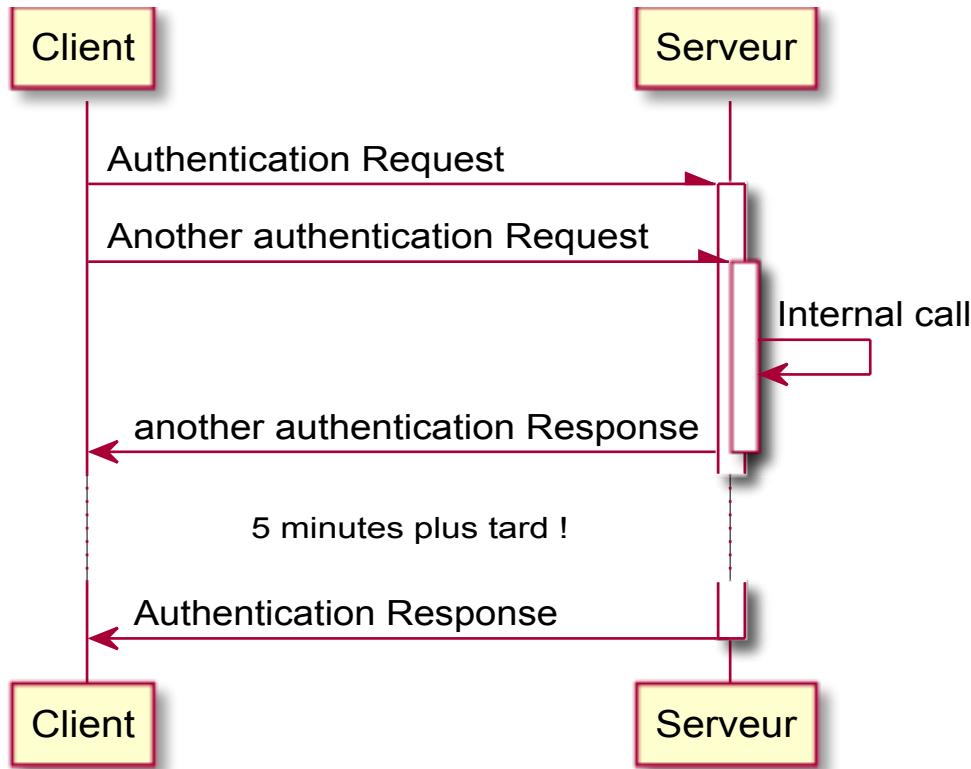
- La majorité des programmes sont écrits de manière séquentielle
- **La plupart** des autres langages / plateformes / frameworks
 - Incitent à programmer séquentiellement
 - Multiplient les threads pour augmenter les performances
 - Ce qui crée des problèmes d'accès concurrents
 - Peuvent utiliser des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

Asynchronisme : exemple



Asynchronisme : exemple

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

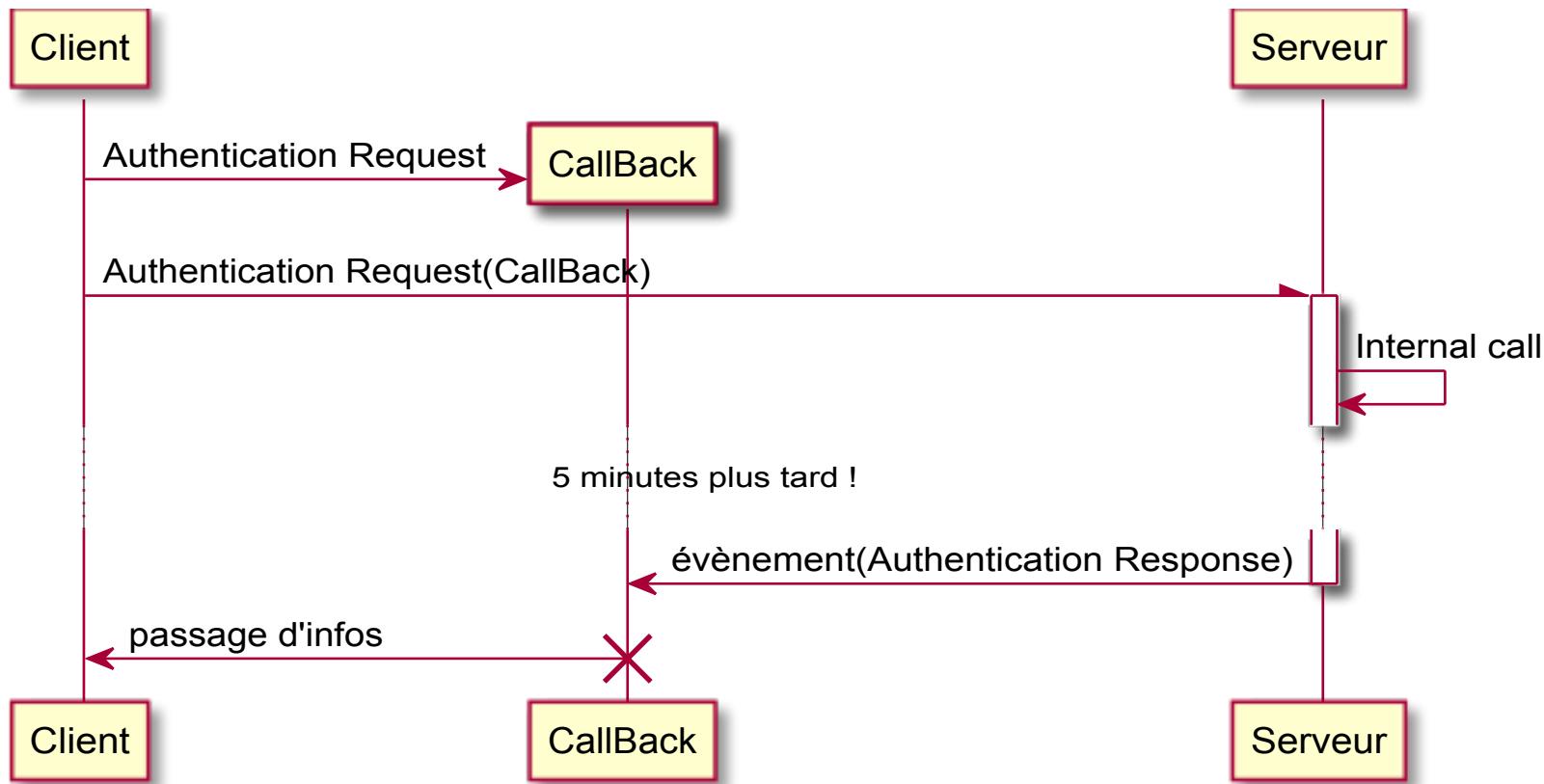


Asynchronisme : complexité

- 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 !)

```
let file = fs.readFileSync('essai.json');
console.log(file);
//-> '[ { "nom": "Doe", "prenom": "John" } ]'
```

- De manière asynchrone

```
let file = fs.readFile('essai.json', function(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

```
let callback = function(err, result) {  
  if (err) {  
    console.error(err.message);  
  } else {  
    console.log(result)  
  }  
};
```

Callbacks

- Les APIs de Node.js ont toujours cette structure

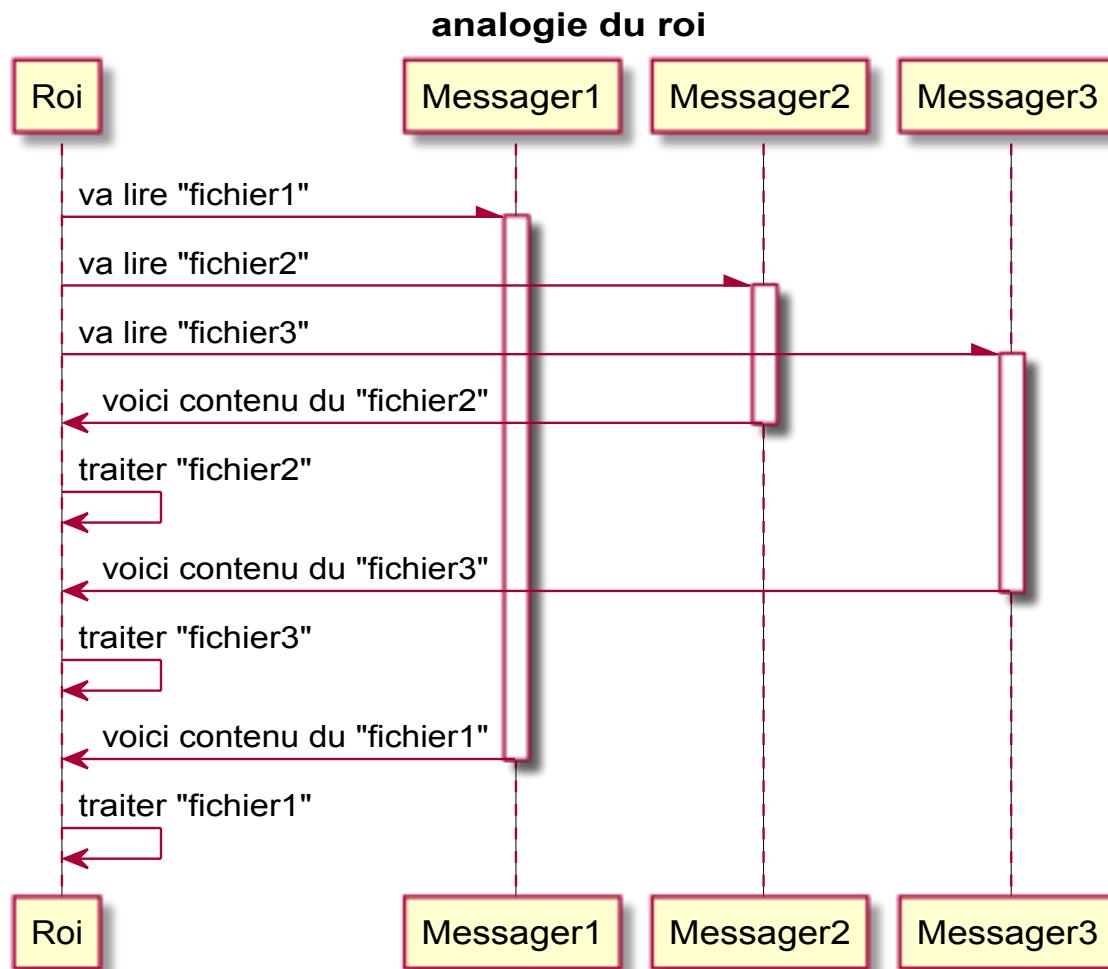
```
module.action(/*[args*]*/, callback);
```

- Le plus souvent, la callback est définie sous forme de fonction anonyme dans l'appel du module

```
module.action(/*[args*]*/, function(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

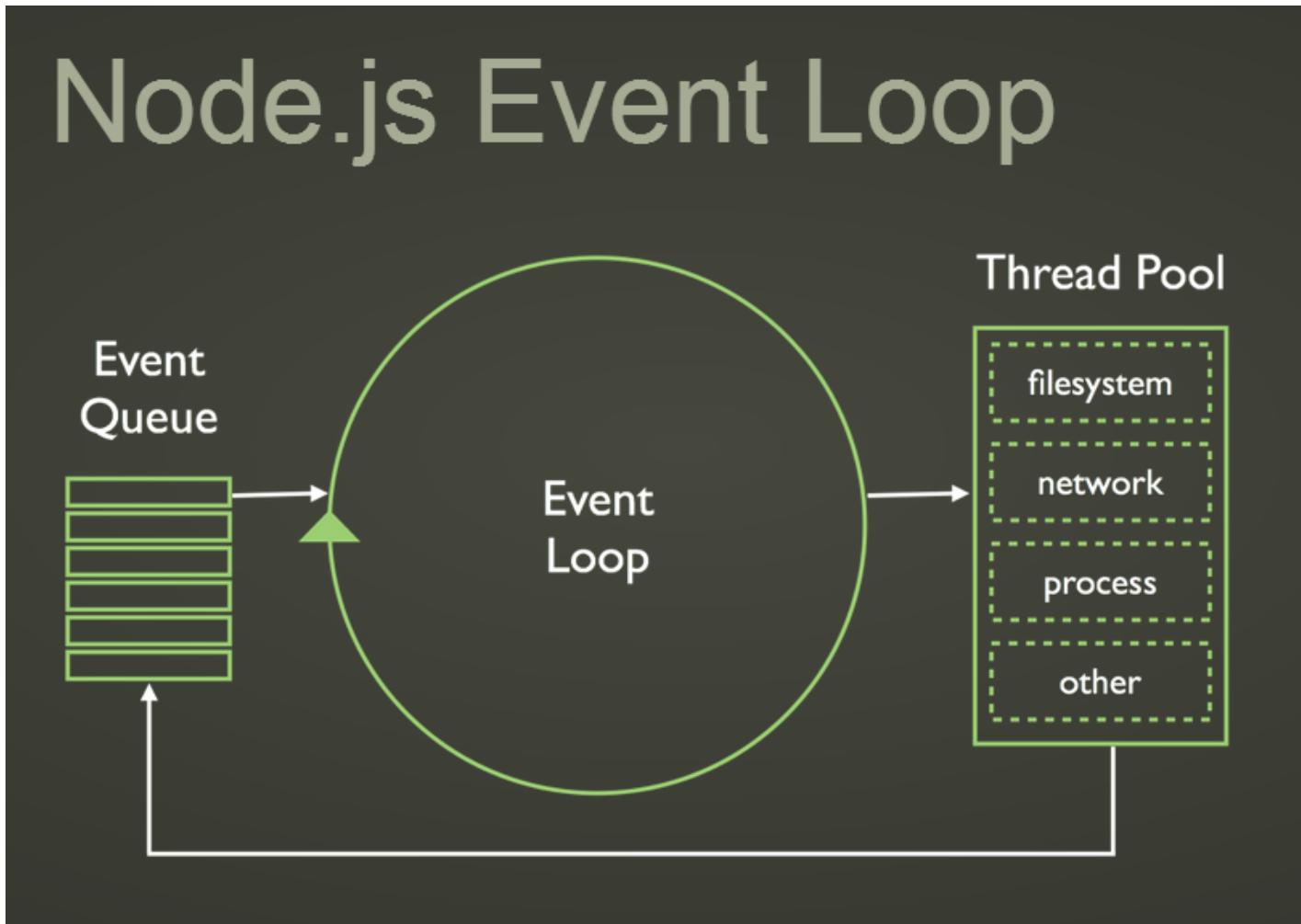
L'analogie du roi



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
- Pour cela Node.js met en place un certain nombre de briques
 - 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







Lab 3

Modules et gestion des dépendances



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- *Modules et gestion de dépendances*
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js

Les modules

- Le langage JavaScript (ES6) apporte la notion de module (import/export)
- Cette notion n'est pas encore implémentée sous la même forme dans Node.js
- Node.js est différente par l'API require/exports
- Chaque fichier `.js` dans Node est un module
- 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 : 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 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



npm install

- npm est un outil en ligne de commande (écrit avec Node.js)
- Il permet de télécharger les modules disponibles sur 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
- 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/api/docs>

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, 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...**

package.json : fonctionnement

- `engines` : définit les moteurs et leur version

```
{ "engines" : { "node" : "<0.12 >=0.10.3" } }
{ "engines" : { "npm" : "~1.0.20" } }
```

- `main` : script principal (point d'entrée)
- `bin` : alias des commandes fournies par le module

```
{ "bin" : { "npm" : "./cli.js" } }
```

- `config` : paramètres de configuration par défaut
- `scripts` : scripts à lancer dans le cycle de vie
 - `publish`, `install`, `update`, `uninstall`, `test`...
 - Le plus souvent utilisé pour définir les tests

Modules : pattern

- Les modules Node correspondent au **module pattern**
- Ou composant en français



- Tout composant doit être conçu comme une boîte noire
- Il publie une interface qui permet d'interagir avec lui



Modules

- La modularisation est ajoutée au JavaScript par Node.js
- <http://nodejs.org/api/modules.html>
- Principe fondateur : **Chaque fichier .js est un module isolé**
- Attention, c'est une notion en rupture avec le **côté client** !
- La modularisation apporte les commandes :
 - `require` : pour charger un module
 - `exports` : pour publier une API

Modules : require

- Modules core : fournis par Node.js, ils sont toujours accessibles

```
const fs = require('fs');
```

- Modules npm : sont accessibles par leur nom s'ils se trouvent dans le répertoire `./node_modules`

```
const yargs = require('yargs');
```

- N'importe quel fichier `.js` : tout fichier `.js` est un module.

```
const submodule = require('./dir/submodule.js');
```

- Il existe différentes astuces (ex : le `.js` peut être omis)
- C'est cette fonctionnalité qu'on utilise pour découper un programme Node.js en plusieurs fichiers.

Modules : require

- **L'opération est synchrone !**
- Les modules sont lus et interprétés dans l'ordre d'appel
- Les dépendances circulaires sont possibles et gérées
- Si le même module est chargé deux fois :
 - Le contenu n'est interprété qu'une seule fois
 - Le module est mis en cache
 - Il est rendu immédiatement s'il a déjà été chargé
 - L'identité d'un module est associé à son fichier `.js`

Modules : exports

- Par défaut, ce qui est dans votre module n'est pas visible
- La partie visible est dans `module.exports`
- Pour publier quelque chose, il faut le référencer dans `exports`

```
// MonModule.js
exports.maMethode = function() {
  console.log('ok');
}
```

```
// AutreModule.js
const monModule = require('./MonModule');

monModule.maMethode();
//-> ok
```

Modules : exports

- `exports` est une variable comme une autre
 - Elle est définie sous forme d'objet par défaut
 - Il est possible de l'affecter à tout type de données
 - Le plus souvent, on choisira un objet ou une fonction
- Astuces à comprendre avec `module.exports` et `exports` :
 - `exports` est un alias de `module.exports`
 - `exports` est pratique pour ajouter des propriétés à l'objet
 - Il n'est pas possible d'affecter `exports`
 - Pour affecter une fonction ou un nouvel objet, il est nécessaire de passer par `module.exports`

Modules : exemple

```
//myApp.js
const fs = require('fs'); //noyau
const chalk = require('chalk'); //npm
const stringUtils = require('./stringUtils'); //local

fs.readFile('./lorem.txt', function(error, data) {
  const upperCased = stringUtils.upperCase(data.toString());
  console.log(upperCased);
});

//stringUtils.js
exports.upperCase = function(string) {
  return string.toUpperCase();
}

//ou module.exports = { upperCase: function... }
```

Core modules - Les modules inclus de NodeJS

- Node.js propose un ensemble de modules noyau (**core**)
- Ils apportent au JavaScript une API pour
 - Accéder au processus et au système d'exploitation
 - Manipuler le système de fichiers
 - Manipuler les interfaces réseau
- Ces interfaces sont écrites en C et dépendantes de l'OS
- Elles sont toujours asynchrones
- <http://nodejs.org/api/>

Modules noyaux

- Certains (rares) modules n'ont pas besoin d'être importés
 - Le module `module`, mais aussi `console` et `process`...
- `console` : outil permettant de logguer facilement
 - `log` : loggue tous les arguments qui lui sont passés
 - `error` : loggue dans `stderr`
- `process` : représente le processus courant
 - `exit` : sort du programme
 - `abort` : sort du programme brutalement
 - `stdout`, `stderr`, `stdin` : les flux du processus
 - Tous les paramètres standards sur le processus courant

Modules noyaux

- Les autres modules noyaux sont à charger avec `require`
- `os` : permet d'avoir des informations sur le système d'exploitation sur lequel s'exécute le processus
- `path` : outils de manipulation des chemins d'accès aux fichiers
- `util` : méthodes utilitaires (`format`, `is*`, `inspect...`)
- `fs` : accès au système de fichiers
 - `readFile`, `writeFile` : lire et écrire dans un fichier
 - `rename`, `chmod`, `rmdir`, ... : manipulation de fichiers
- `net` : fournit des APIs pour l'accès au réseau
- Et bien d'autres : `http`, `https`, `dns`, `child_process`...

Gestion des dépendances

- Tous les modules sont isolés avec Node.js
- Ils sont identifiés par leur fichier `.js`
- Il est donc possible de charger un même module dans différentes versions
- Cette fonctionnalité est assez rare parmi les équivalents
- En contrepartie, toutes les dépendances sont répliquées

```
yourmodule/
  node_modules/
    dep1/
      node_modules/
        subdep/ (1.0)
    dep2/
      node_modules/
        subdep/ (2.0)
```

Publier un module npm

- Il est bien sûr conseillé de suivre toutes les bonnes pratiques
 - Utiliser la numérotation recommandée
 - Avoir des tests unitaires
 - Avoir un minimum d'informations dans le `package.json`
- Il n'y a pas d'autorité de validation
- Il faut par contre trouver un nom disponible
- La suite nécessite seulement la commande `npm`
 - `npm adduser` : enregistrer son compte
 - `npm publish` : uploader un module sur `npmjs.org`





Lab 4

Node et le Web : HTTP, Connect & Express



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- *Node et le Web : HTTP, Connect & Express*
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js

Node.js et le Web

- Node.js a été créé à l'origine pour faire du Web
- Sa focalisation sur les I/O provient de cet objectif
- Mais Node.js ne peut pas être considéré comme un serveur Web
- Node.js est une plateforme qui permet de faire un serveur Web
- Il faut programmer un serveur Web pour en avoir un
- Heureusement, c'est très simple à faire !

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('hello world');
}).listen(1234);
```

Module HTTP

- Node propose un module noyau **HTTP** (et **HTTPS**)
- Il fournit un certain nombre d'APIs de bas niveau :
 - Pour communiquer avec un Serveur HTTP :

```
let request = http.request(options, callback);
```
 - Pour créer un serveur HTTP :

```
const server = http.createServer(requestListener).listen(port);
```
- Il supporte tout le protocole HTTP/1.1
 - Le cache : ETag, If-Modified-Since
 - Réseau: Connexions persistantes, Pipelining, Chunk Transfer Encoding...
 - Un nouveau verbe : UPGRADE

Module HTTP : client

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

let options = {
  hostname: 'registry.npmjs.org',
  port: 80,
  path: '/',
  method: 'GET'
};

let req = http.request(options, function(res) {
  console.log('Code HTTP : ', res.statusCode);
  console.log('Headers : ', res.headers);
  console.log('Contenu : ')
  res.pipe(process.stdout);
}).on('error', function(e) {
  console.log('Problem with request: ' + e.message);
});

console.log('envoi');
req.end();
console.log('envoyé');
```



Module HTTP : client

envoi

envoyé

Code HTTP : 200

Headers : { server: 'CouchDB/1.5.0 (Erlang OTP/R14B04)',
'content-type': 'text/plain; charset=utf-8', ...}

Contenu :

{"db_name": "registry", "doc_count": 75380, ...}

Module HTTP : client

- La requête et la réponse sont des **streams**
- L'utilisation des streams est détaillée dans un chapitre suivant
- La requête ne part vraiment qu'à l'appel de `req.end()`
- Le module `request` propose une surcouche pour avoir une API plus avenante

Module HTTP : serveur

- Le module HTTP permet également de créer un serveur
- Il s'agit par contre uniquement du protocole HTTP
- Rien à voir avec les fonctionnalités d'un serveur **Apache** ou **nginx**
- Pour cela il faut :
 1. Charger le module HTTP
 2. Créer un serveur HTTP
 3. Définir un callback pour traiter les requêtes
 4. Mettre le serveur a l'écoute d'un port

Module HTTP : serveur

1. Charger le module HTTP

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

2. Créer un serveur HTTP

```
const serveur = http.createServer();
```

3. Définir un callback pour traiter les requêtes

```
serveur.on('request', function(req, res) {
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.write('hello world');
  res.end();
});
```

4. Mettre le serveur à l'écoute d'un port

```
serveur.listen(port);
```



Module HTTP : serveur

```
curl http://localhost:1234 -v
...
- About to connect() to localhost port 1234 (#0)
- Trying 127.0.0.1...
- Connected to localhost (127.0.0.1) port 1234 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:1234
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Date: Thu, 15 May 2014 08:33:39 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
- Connection #0 to host localhost left intact
hello world%
```

Serveur Web mono-threadé

- Comme nous l'avons vu, Node.js est construit sur
 - L'event loop mono-threadé
 - Des I/O asynchrones
- Node.js comme serveur HTTP a donc des particularités
 - + Un serveur Web faisant majoritairement de l'I/O Node.js peut se révéler extrêmement performant
 - - Une seule requête peut bloquer le serveur si son traitement est long et synchrone
- Il peut être nécessaire de
 - Découpler des traitements coûteux par messaging
 - Monter un cluster (voir les chapitres suivants)

Module HTTP : limites

- Le module HTTP s'arrête à la gestion du protocole
- Le callback de gestion des requêtes doit alors
 - Prendre en compte l'**URL** et la **method** HTTP
 - Les headers, paramètres, types de contenus...
 - Router les requêtes pour ne pas tout traiter au même endroit
 - Gérer les headers de la réponse
- **Le module HTTP est trop bas niveau**
 - Il faut s'outiller pour créer un vrai serveur
 - Il existe de nombreuses librairies : **director**, **connect**, **express**, **dispatch**

Connect

- Le module **Connect**
 - Framework de serveur HTTP pour Node.js
 - Ultra extensible par plugins nommés **middlewares**
 - Développé par TJ Holowaychuk, un grand nom des modules Node.js
 - Rarement utilisé directement, il est en dépendance de nombreuses solutions notamment **Express**

```
const app = connect()  
  .use(logger())  
  .use(serveStatic('public'))  
  .use((req, res) => res.end('hello world'));  
http.createServer(app).listen(3000);
```

Connect : les middlewares

- **static** : Servir automatiquement des fichiers statiques
- **favicon** : Définir la favicon du site
- **logger** : Trace les requêtes reçues avec différents formats (standard, dev, tiny, ...)
- **query** : Décoder la requête
- **errorHandler** : Permet de générer un traitement d'erreur particulier
- **directory** : Permettre la visualisation du contenu de sous dossiers
- **bodyParser** : Décode le body des requêtes, notamment JSON
- **session** : Active un système de gestion de sessions
- Une fonctionnalité n'est pas adressée : le **routage**

Connect : exemple

```
const http = require('http');
const connect = require('connect');
const serveStatic = require('serve-static');
const serveIndex = require('serve-index');
const serveFavicon = require('serve-favicon');
const query = require('connect-query');
const errorhandler = require('errorhandler');
const logger = require('morgan');

const app = connect()
  .use(logger('dev'))
  .use(query())
  .use(serveFavicon(__dirname + '/public/favicon.ico'))
  .use(serveStatic('public'))
  .use(serveIndex('public', {
    hidden: true,
    icons: true
}))
  .use(errorhandler())
  .use(function(req, res){
    res.end(JSON.stringify(req.query));
})
http.createServer(app).listen(3000);
```

Express

- Le module **Express**
 - Toujours développé par TJ Holowaychuk
 - S'appuie sur le module **Connect**
 - Intègre un système de routage
 - Fournit un générateur d'application **express-generator**

```
const express = require('express');
const app = express();

app.get('/hello', function(req, res) {
  res.send('Hello World');
});

app.listen(3000);
```

Express : generator

- Propose de générer la structure d'une application Express
- La commande va générer les fichiers dans le répertoire

```
$ express -h

Usage: express [options] [dir]

Options:

  -h, --help          output usage information
  -V, --version       output the version number
  -e, --ejs           add ejs engine support (defaults to pug)
  -H, --hogan          add hogan.js engine support
  -c, --css <engine>  add stylesheet <engine> support
                      (less|stylus|compass)
                      (defaults to plain css)
  -f, --force          force on non-empty directory
```

Express : structure

- La structure d'un serveur Express ressemble à cela

```
const express = require('express');
const logger = require('morgan');
const app = express();

/* Chargement des middlewares connect */
app.use(express.static('./public'));
app.use(logger());

/* Ajout de routes */
app.get('/hello', function(req, res) {
  res.send('Hello World');
});

app.listen(8080);
```

Express : router

- Il existe de nombreuses façons de réagir aux requêtes

- Sous forme de middleware

```
app.use(function(req, res, next) {});
```

- Avec la méthode HTTP et l'URL

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

- Répondre à toutes les méthodes

```
app.all('/resource', function(req, res) {});
```

- Capturer des paramètres dans l'URL

```
var callback = function(req, res) { /*req.params*/ }
app.get('/resource/:id', callback);
app.get('/^\/resource\/(\d+)$/', callback);
```

Express : la requête

- L'objet requête ressemble à celui manipulé par le module HTTP
- Express propose une API plus simple d'utilisation
 - `request.params` : les paramètres capturés dans l'URL
 - `request.query` : les paramètres 'query-string'
 - `request.body` : le body de la requête, parsé si on utilise le middleware **bodyParser**
 - `request.headers` : les headers de la requête
 - `request.cookies` : les cookies passés dans la requête
 - <http://expressjs.com/4x/api.html#req.params>

Express : la réponse

- L'objet réponse ressemble à celui manipulé par le module HTTP
- Express propose une API plus simple d'utilisation
 - `res.status(code)` : spécifie un code de retour HTTP
 - `res.send(status, body)` : envoie un contenu (sans stream)
 - `res.sendFile(path)` : envoie un fichier
 - `res.redirect(url)` : renvoie une redirection HTTP
 - `res.cookie(name, value)` : positionne un cookie
 - <http://expressjs.com/4x/api.html#res.status>

Express : templates

- Express propose l'intégration d'un moteur de templates
 1. Définir le dossier contenant les templates

```
app.set('views', __dirname + '/views');
```

2. Choisir le moteur de templates

```
app.set('view engine', 'pug');
```

3. Créer le template

```
vim views/index.pug
```

4. L'utiliser dans une route

```
res.render('index', { message: 'Vive Express' });
```

Express : templates

- Il existe différents moteurs de templates pour Node.js
 - Pug
 - Le choix par défaut pour Express
 - Propose une notation sans balise avec indentation significative
 - Les variables sont prefixées par =
 - EJS
 - HTML classique
 - Dynamisme introduit par des syntaxes **type JSP** <% opérations %>, <%= expression %>
 - Et beaucoup d'autres...

Express : templates

- Pug est parfois utilisé simplement pour alléger la syntaxe HTML
- Exemple de template Pug

```
doctype html
html(lang="en")
  head
    title= pageTitle
  body
    h1 Pug - node template engine
    #container.col
      if youAreUsingPug
        p You are amazing
      else
        p Get on it!
    ul
      each val, index in ['zero', 'one', 'two']
        li= index + ': ' + val
```

Aller plus loin

- Authentification avec Express
 - Connect propose une gestion automatique de sessions
 - Il utilise un cookie et conserve l'état en mémoire (stateful)
- Passport
 - Middleware Express pour gérer l'authentification
 - Très riche fonctionnellement : oAuth, SSO...
- Alternatives à Express
 - Express est incontestablement leader
 - Un concurrent important : Hapi
 - <https://github.com/hapijs/hapi>





Lab 5

L'asynchrone en détails



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- *L'asynchrone en détails*
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js

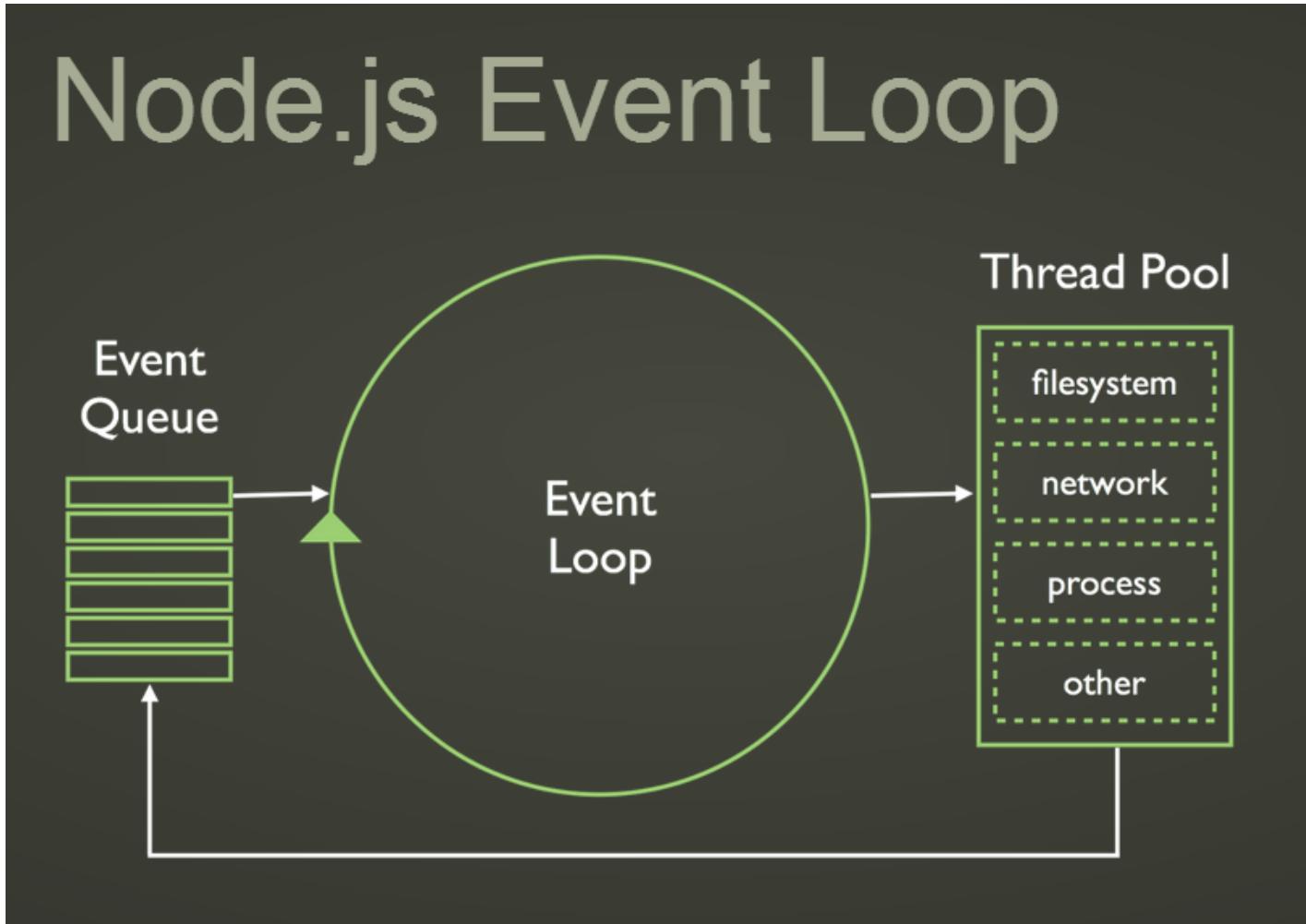
Asynchronisme

- Node.js est **monothreadé** mais **non bloquant pour les entrées/sorties (I/O)** comme la lecture d'un fichier, le requêtage en base de données ou la réponse à une requête HTTP.

```
let hello = function() {  
    for(let i = 0;  
        i <= 1000000000; i++ ) {  
        if(i == 100000000) {  
            console.log("Hello");  
        }  
    }  
};  
  
hello();  
console.log("world!");  
//-> Hello  
//-> world!
```

```
const fs = require('fs');  
  
let hello = null;  
fs.readFile('hello.txt', 'utf8',  
    function (err, data) {  
        if (err) throw err;  
        console.log(data);  
        hello = data;  
    });  
  
console.log(hello, "world!");  
//-> null world  
//-> Hello
```

Asynchronisme : event loop



Callback Hell ou Pyramid of Doom

- Faire appel de façon **répétée** à des fonctions auxquelles on fournit une fonction de **callback** le plus souvent **anonyme**.

```
const fs = require('fs');
let myCallbackHell = function(jsonData,callback) {
    // On sauvegarde la donnée dans un fichier
    fs.writeFile('data.json', jsonData, function (err) {
        // 1er callback, on ajoute un élément lu dans un fichier
        fs.readFile('data2.json', 'utf8', function (err, data2) {
            // 2ème callback
            jsonData["data2"] = data2;
            // 3ème callback
            callback(err, jsonData);
        });
    });
};

myCallbackHell({titi: tata}, function(err,result) {
    console.log(result);
});
```

Callback Hell ou Pyramid of Doom

- Callback Hell ou Pyramid of Doom
 - Désagréable à lire à cause de l'indentation
 - Ne fonctionne qu'avec des actions séquentielles
 - Ne répond pas à la problématique de traitements parallèles
- Déclencher un traitement quand tous ses prérequis sont prêts peut devenir en programmation asynchrone un vrai défi
- La gestion des erreurs devient également vite problématique

Gestion de l'asynchronisme

- Deux concepts se proposent de répondre à cette complexité
 - Changer de paradigme et utiliser des **promises**
 - Différentes implémentations existent
 - L'objet `Promise` natif à V8 depuis Node 0.12
 - `Q` l'implémentation d'origine
 - `Bluebird`, l'implémentation populaire pour ses fonctionnalités et ses performances
 - Utiliser les Générateurs
- Ces modules sont utilisables avec Node ou en frontend

Les Promesses

- C'est avant tout un nouveau concept
- Petite révolution dans le monde du JavaScript
- Les **promesses** proposent de remplacer les **callbacks**
- **Une promesse est un objet qui sera résolu de façon asynchrone**
- Permet à un traitement asynchrone d'avoir une valeur de retour
- Il est ensuite possible d'écouter l'évènement de résolution de la promesse (avec un callback !)
- Permet de ne pas implémenter le comportement au même moment ou avant l'appel à une fonction asynchrone.

Les Promesses : les implémentations

- Il y a de nombreuses implémentations pour le concept.
- `Q`
 - Implémentation historique et de référence
 - <https://github.com/kriskowal/q>
- `Promise`
 - Depuis Node v0.12, l'objet Promise existe nativement
 - Tend à devenir le standard puisque natif
 - Les performances de cette implémentation sont remises en cause
- `Bluebird`
 - Implémentation très utilisée car riche et performante

Utiliser une promesse

- Pour utiliser une API utilisant les promesses

```
funcPromise(param1, ..., param[n])
  .then(function success() {
    /* ... */
  });
}
```

- Ou en version complète avec gestion des erreurs

```
const promise = funcPromise(param1, ..., param[n]);

promise.then(
  function success() { /*... */ },
  function error() { /* ... */ }
);
```

Chaînage des promesses

- Pour chaîner une série de traitements asynchrones
- Voir plus loin pour la définition de la fonction `readFile`

```
readFile('file1.txt')
  .then(function(arg) {
    // arg vaut le contenu de file1
    return readFile('file2.txt');
  })
  .then(function(arg) {
    // arg vaut le contenu de file2
    return readFile('file3.txt');
  })
  .then(function(arg) {
    // arg vaut le contenu de file3
  })
  .catch(function(err) {
    console.log('Erreur', err);
});
```

Parallèle et promesses

- Pour traiter en parallèle des traitements asynchrones
- Et avoir un callback quand tous sont terminés

```
Promise.all([
  readFile('file1.txt'),
  readFile('file2.txt'),
  readFile('file3.txt')
])
  .then(function(results) {
    // results est un tableau des contenus des trois fichiers
    // (dans l'ordre des promesses)
  })
  .catch(function() {
    console.log('Erreur', err);
});
```

Node et les promesses

- L'API de Node ne propose pas de retourner une promesse directement
- Il faut encapsuler une fonction avec callback
- Méthode `promisify` du module core `util` apparu en v8.0.0

```
const util = require('util');
const action = util.promisify(module.action);
action(param1, ..., param[n]);
```

```
const readFile = util.promisify(fs.readFile);
readFile('file1.txt')
  .then(function(content) {
    console.log('file content = ', content);
});;
```

- Pour des versions de Node plus anciennes, des librairies existent, exemple : `denodeify`

Créer une promesse

- Préférer le chaînage de promesses plutôt que d'en créer !

```
function funcCallback(param1, ..., paramn, callback) {  
  /* ... */  
  if (condition) {  
    callback(new Error('example error'), params);  
  } else {  
    callback(null, results);  
  }  
}  
  
function funcPromise(param1, ..., paramn) {  
  return new Promise(function(resolve, reject) {  
    /* ... */  
    if (condition) {  
      reject(new Error('example error'));  
    } else {  
      resolve(results);  
    }  
  });  
}
```

Traitement des erreurs asynchrones

- En JavaScript les **erreurs** levées remontent la pile d'exécution
- L'**asynchronisme** brise fréquemment cette pile d'exécution
- Une erreur dans une chaîne de callbacks sera alors **invisible**
- De ce fait, la gestion des erreurs est une préoccupation

```
try {
  //Simulation d'asynchronisme
  setTimeout(function() {
    throw new Error('example error');
  });
} catch(error) {
  console.log('Une erreur a été interceptée:', error.message);
}
```

- Dans cet exemple, l'erreur n'est pas **attrapée** par le bloc catch.

Traitement des erreurs asynchrones : Promesses

- Si une erreur est reportée, la chaîne est **rompue**
- Plus aucun traitement n'est effectué
- A la résolution d'une promesse (optionnel).

```
aPromise.then(  
  function(resolvedValue) {},  
  function(error) {}  
)
```

- Avec **catch** (optionnel) attrape n'importe quelle erreur non attrapée plus haut

```
aPromise.catch(function(error) {});
```

- En bout de chaîne **done ()** (optionnel) lève toute erreur non attrapée, attention peut interrompre le serveur !

Async / Await

- Depuis ES2017, il est possible d'avoir une écriture plus proche des appels synchrones avec les Promises, Async/Await

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

var add = async function(x) {
  var a = await resolveAfter2Seconds(20);
  var b = await resolveAfter2Seconds(30);
  return x + a + b;
};

add(10).then(v => {
  console.log(v); // affiche 60 après 4 secondes.
});
```





Lab 6

Communication temps réel



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- *Communication temps réel*
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js

Web temps réel

- Par Web temps réel, on parle principalement de **push**
- Le serveur Web doit pouvoir contacter les clients
- Node.js a été inventé pour répondre à ce besoin
- Les autres technologies utilisent un thread par client
- Les I/O asynchrones permettent de n'utiliser qu'un seul thread
- Node.js est particulièrement adapté puisqu'il s'agit le plus souvent de communications très nombreuses mais très courtes
- C'est encore aujourd'hui une application courante de Node.js

Web temps réel

- Pour faire du Web temps réel, il faut se conformer aux normes du Web et aux capacités des navigateurs
- La technologie la plus naturelle correspond aux **Web Sockets**
- Elle n'est pas encore disponible partout et son fonctionnement dépend d'éléments réseau tels que les proxys Web
- La plupart des frameworks Node.js implémentent des **fallbacks**
 - Server Sent Event
 - Flash Socket
 - Ajax long polling
 - Forever iframe
 - JSONP polling

Web temps réel : WebSockets

- Spécification normalisée par l'IETF (Internet Engineering Task Force) dans le RFC 6455 et fait partie d'**HTML5**
- <http://www.w3.org/TR/2009/WD-websockets-20091222/>
- Principe
 - Connexion permanente entre le client et serveur
 - Possibilité de communication dans les deux sens
- Le protocole est constitué de deux étapes
 - Handshake
 - Transfert de données

Socket.IO



- C'est un framework pour faire des applications temps réel
- Il propose une API très simple
 - Pour établir la connexion avec les clients
 - Transmettre des messages dans les deux sens
- Il existe depuis très longtemps (V0.1 en 2010, V1 en 2014!)
- Mis en cause pour ses performances
 - Mais seulement dans des contextes TRÈS sollicités

Socket.IO : transports

- Socket.IO fonctionne avec un système de ***transports***
- La cible est l'utilisation des **WebSockets**
- Si la connexion n'est pas disponible ou échoue, il utilise automatiquement une autre solution
- Le transport utilisé est transparent à l'utilisation
- Mais peut se ressentir sur les performances !
- Transports existants par défaut :
 - WebSocket, Adobe® Flash® Socket , AJAX long polling, AJAX multipart streaming, Forever Iframe, JSONP Polling

Socket.IO : API

- Socket.IO fonctionne avec une librairie côté client
 - Elle implémente le choix du transport
 - L'API haut niveau pour l'échange de messages
- Socket.IO ne reproduit pas l'API des WebSockets
 - Contrairement à un de ces principaux concurrents **SockJS**
 - Il propose une API plus haut niveau
 - Ajoute notamment la notion de message avec un **nom**
 - Notion très pratique pour faire le routage des messages

Socket.IO : API serveur

- Connection d'un client

```
io.on('connection', function(socket) {});
```

- Émission d'un message à tous les clients

```
io.emit(name, content);
```

- Émission d'un message

```
socket.emit(name, content);
```

- Émission d'un message à tous les clients sauf celui-ci

```
socket.broadcast(name, content);
```

- Écoute de la réception d'un message

```
socket.on(name, function(data) {});
```

Socket.IO : API cliente

- Connexion au serveur

```
const socket = io.connect(url);
```

- Écoute de la réception d'un message

```
socket.on(name, function(data) {});
```

- Émission d'un message

```
socket.emit(name, content);
```

- Attention à ne pas mélanger les noms des messages

- Qui vont du serveur vers le client
 - Qui vont du client vers le serveur

- Les **content** peuvent être du texte ou du **JSON**

Socket.IO : exemple

- Les différentes étapes pour la mise en place d'un socket sont :
 1. Créer un serveur HTTP
 2. Associer Socket.IO en le mettant à l'écoute du serveur
 3. Traiter l'évènement de `connection`
 4. Une fois le socket établi
 - Émettre des évènements à destination du client
 - Écouter des messages émis par le client
- Toute la plomberie concernant le choix du transport et son implémentation est transparente !
- Il est bien sûr possible de rajouter du paramétrage

Socket.IO : exemple côté serveur

```
const express = require('express');
const logger = require('morgan');
const http = require('http');

const app = require('express')();
const server = require('http').Server(app);
const io = require('socket.io').listen(server);

app.use(express.static('public'))
app.use(logger());

io.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });
  socket.on('my other event', function (data) {
    console.log(data);
  });
});

server.listen(8000);
```



Socket.IO : exemple côté client

```
< html>
  < head>
    < title>
      Socket
    < /title>
    < script src="scripts/socket.io.js"></ script>
    < script>
      var socket = io.connect('http://localhost:8000');
      socket.on('news', function (data) {
        console.log(data);
        socket.emit('my other event', { my: 'data' });
      });
    </ script>
  </ head>
  < body>
    < /body>
< /html>
```





Lab 7

La gestion des streams



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- *La gestion des streams*
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js

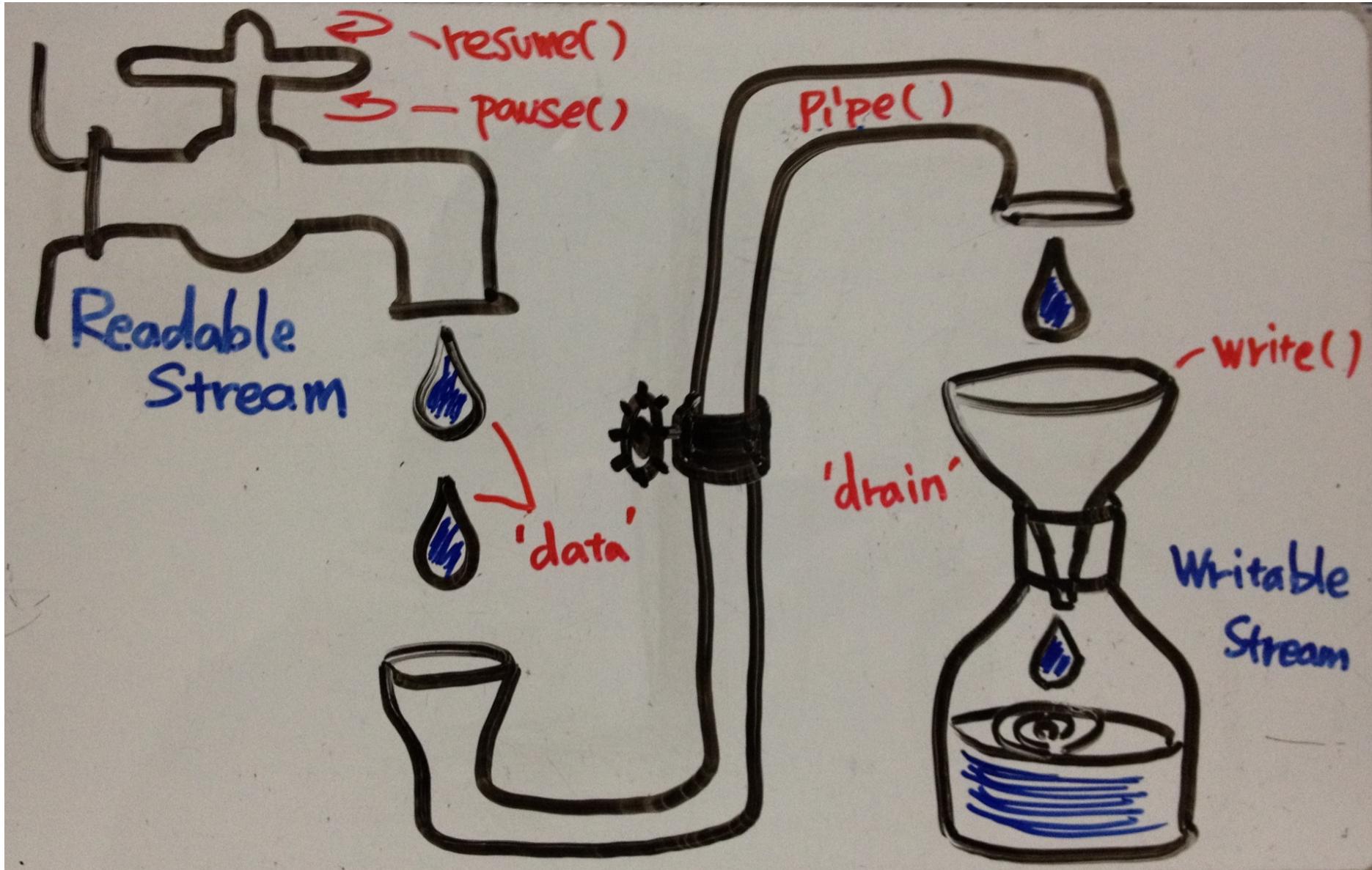
Définition

- Les **streams** sont une notion centrale de Node.js
- Il s'agit, comme son nom l'indique, de flux de données
- Il est possible de réaliser de très nombreuses opérations
 - Chaîner des flux
 - Réaliser des transformations
- On utilise très rapidement des streams sans même le savoir
 - Request et Response HTTP sont des streams
 - Websockets
 - Toutes les manipulations de fichiers sont basées sur les streams

Définition

- Les **streams** sont des objets Node.js
- Il existe les classes : `Readable`, `Writable`, `Duplex` et `Transform`
- Tous les streams sont `EventEmitter`
- Pour les manipuler :
 - On peut en implémenter de nouveaux
 - Utiliser des librairies pour en créer rapidement
 - Les chaîner avec la méthode `.pipe()`
 - Écouter leurs évènements
- API : <http://nodejs.org/api/stream.html>

Définition : en une image



Pourquoi ?

- Les streams permettent de traiter les données progressivement
- C'est indispensable considérant la nature progressivement asynchrone de Node.js
- Il est par exemple intéressant de pouvoir traiter les données d'un fichier au fur et à mesure de sa lecture
- Traiter un gros fichier revient alors à un grand nombre de micros traitements ce qui est la force de Node.js
- C'est également avantageux en terme de mémoire
- Il n'est pas nécessaire d'avoir toutes les données en mémoire avant de pouvoir les traiter

Streams : un exemple

```
const childProcess = require('child_process');
const through2 = require('through2');

let spawned = childProcess.spawn('ls', ['-la']);

let transform = through2(function(chunk, enc, done) {
  this.push(chunk.toString().toUpperCase());
  done();
});

spawned.stdout
  .pipe(transform)
  .pipe(process.stdout);
```

Buffers

- Les **streams** manipulent les données sous forme de **Buffer**
- Les **Buffer** sont une notion importante dans Node (pas seulement pour les streams)
- Cela correspond à un type de données bas niveau de V8
- Pour afficher les données, il faut spécifier un encoding (par défaut UTF-8)
- Propose une API très complète pour les manipuler
- API : <http://nodejs.org/api/buffer.html>
- Les **streams** ont un **object mode** qui remplace les **Buffer** par du JavaScript standard (voir plus loin)

Type de flux

- Il existe différents types de flux
- Certains sont la composition des autres
- Dans certains cas, on sera en position
 - de créer le flux
 - d'être consommateur du flux

Type de flux : Readable

- Type de stream pour les entrées de données
- Exemples : réponses HTTP depuis le client, requêtes HTTP sur le serveur, lecture de fichiers, sockets, stdout et stderr des process enfants...
- Evènements : `readable`, `data`, `end`, `close`, `error`
- Méthodes : `read`, `setEncoding`, `resume`, `pause`, `pipe`, `unpipe`,
`unshift`, `wrap`
- Écouter l'évènement `data` ou utiliser `resume` ou `pause` passera le stream en `flowing mode`
- Lire les données "nouvelle version" : `readable.read([size])`
- `readable.pipe(writable, [options])` permet d'écrire toutes les données dans un stream writeable

Type de flux : Readable exemple

```
const fs = require('fs');

let readable = fs.createReadStream('../images/stream-image.jpg');
readable.on('readable', function() {
  let chunk;
  while (null !== (chunk = readable.read())) {
    console.log('got %d bytes of data', chunk.length);
  }
});
```

```
got 65536 bytes of data
got 36160 bytes of data
```

Type de flux : Readable implementor

```
const Readable = require('stream').Readable;

class Counter extends Readable {
  constructor(options) {
    super(options);
    this._index = 0;
  }

  _read() {
    if (this._index > 100) {
      this.push(null);
    } else {
      this.push(this._index++ + ' ');
    }
  }
}

let counter = new Counter();
counter.pipe(process.stdout);
```



Type de flux : Writable

- Type de stream pour les sorties de données
- Exemples : requêtes HTTP depuis le client, réponses HTTP sur le serveur, écriture de fichiers, sockets, stdout et stderr...
- Évènements : `drain`, `finish`, `pipe`, `unpipe`, `error`
- Méthodes : `write`, `end`

Type de flux : Writable exemple

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

let server = http.createServer(function (req, res) {
  res.write('hello, ');
  res.end('world!');
});

server.listen(1234);
```

```
HTTP/1.1 200 OK
Date: Mon, 12 May 2014 08:20:00 GMT
Connection: keep-alive
Transfer-Encoding: chunked
```

```
hello, world!
```

Type de flux : Writable implementor

```
const Writable = require('stream').Writable;
const chalk = require('chalk');
const util = require('util');
util.inherits(RedLogger, Writable);

function RedLogger(opt) {
  Writable.call(this, opt);
}

RedLogger.prototype._write = function(chunk, encoding, callback) {
  console.log(chalk.red(chunk.toString()));
};

const redLogger = new RedLogger();
process.stdin.pipe(redLogger);
```

Type de flux : Duplex

- Combine les types **Readable** et **Writable**
- Ne réalise pas de transformation sur les données
- Exemples : sockets, zlib, crypto
- Implementor : doit implémenter les méthodes `_read` et `_write`
- Considéré comme une **classe** abstraite

Type de flux : Transform

- Extension du concept de Duplex
- Prévu pour réaliser des transformations sur les données
- Exemples : zlib, crypto
- Implementor : permet d'éviter d'écrire les méthodes `_read` et `_write` au profit de `_transform` et `_flush`.
- Très utilisée, on passera par contre presque toujours par la librairie **`through2`**

Pipelining

- Définition : Permet de lire toutes les données d'un **stream Readable** pour les transférer à un **stream Writable**
- La gestion du flux est alors gérée automatiquement
- Le flux résultant étant retourné, s'il est **Duplex** (ou **Transform**), cela permet de chaîner les appels

```
let readable = fs.createReadStream('file.txt');
let transform = zlib.createGzip();
let writable = fs.createWriteStream('file.txt.gz');

readable.pipe(transform).pipe(writable);
```

Object Mode

- Par défaut les streams manipulent des **Buffer**
- Ils ont un ***object mode*** qui remplace les **Buffer**
- Le stream va alors manipuler des données JavaScript
- C'est pratique lorsque le flux est transformé en amont
 - HTML -> DOM
 - String -> JSON
 - Il existe des librairies pour réaliser ces opérations
- En ***object mode*** la lecture et l'écriture se comportent un peu différemment (plus possible de spécifier le nombre de bytes à lire)
- http://nodejs.org/api/stream.html#stream_object_mode

Librairies

- L'API standard de Node.js pour les streams peut être assez verbeuse
- De nombreuses micro-librairies existent pour accélérer la manipulation des streams
- La plus courante, presque indispensable : ***through2***
- D'autres implémentent des opérations courantes : concat, combine...
- Ne pas hésiter à utiliser ces librairies pour aérer le code

Librairies : through2

- En pratique, manipuler des streams revient souvent à créer des streams **Transform**
- C'est un code assez verbeux avec le système d'héritage
- La librairie **through2** répond à cette problématique en proposant la création d'un stream **Transform** en une instruction

```
const through2 = require('through2');
const colors = require('colors');

let rainbowStream = through2(function(chunk, encoding, callback) {
  this.push(chunk.toString().rainbow);
  callback();
});

process.stdin.pipe(rainbowStream).pipe(process.stdout);
```

Librairies : through2 map, filter, reduce & spy

- **through2** a des versions de plus haut niveau
 - **through2-map** applique une fonction à chaque `chunk`
 - **through2-filter** filtre les `chunk`
 - **through2-reduce** agrège les `chunk`
 - **through2-spy** espionne les chunks

```
const map = require('through2-map');
const colors = require('colors');
let rainbow = map(function(chunk) {
  return chunk.toString().rainbow;
});
process.stdin.pipe(rainbow).pipe(process.stdout);
```

Librairies : concat, duplexer, trumpet, combiner

- **concat-stream** : appelle un callback quand toutes les données entrantes ont été lues
- **duplexer** : combine un stream Readable et un Writable pour former un Duplex
- **combiner-stream** : transforme un pipeline en un seul stream
- **trumpet** : permet de manipuler des streams contenant du code HTML
- **JSONStream** : permet de manipuler des streams contenant du JSON
- Certaines librairies proposant une transformation quelconque proposent leur API sous forme de stream

Gulp



- Gulp est un outil de build JavaScript / Node.js
- Il se présente comme une alternative intéressante à Grunt
- Les tâches sont toutes gérées sous forme de stream Node.js
- Un plugin Gulp est un stream Transform





Lab 8

Liaison avec la persistance des données



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- *Liaison avec la persistance des données*
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- Au delà de Node.js

Liaison avec la persistance des données

- Node.js a besoin comme toutes les plateformes de communiquer avec les bases de données
- On distingue deux grandes catégories
 1. Les bases de données SQL : **SGBDR**
 - Oracle, MSSQL, PostgreSQL, MySQL, ...
 2. Les bases Not Only SQL : **NoSQL**
 - MongoDB, Redis, Neo4J, couchDb, ...
- Node.js est souvent trop rapidement associé au NoSQL
- Pourtant, Node peut très bien travailler avec des bases SQL

Drivers

- Comme sur les autres plateformes, il faut avoir le driver correspondant à la base de données
- On trouve sur **npmjs.org** un driver pour à peu près toutes les bases de données
- Il n'y a pas d'API normalisée pour Node.js
 - Chaque driver peut donc avoir sa propre API
 - Il existe des drivers qui s'imposent une API commune
 - Il existe des initiatives d'abstraction (voir plus loin)
- Les drivers ont la particularité de contenir du code natif

Drivers : exemple avec Oracle

- Il faut définir des variables d'environnement

```
OCI_LIB_DIR=/path/to/instant_client  
OCI_INCLUDE_DIR=/path/to/instant_client/sdk/include  
OCI_VERSION=<10, 11, or 12> # par défaut 11  
NLS_LANG=.UTF8
```

- Configuration du driver

```
const oracle = require('oracle');

let connectData = {
  hostname: 'localhost',
  port: 1521,
  database: 'sid', // System ID (SID)
  user: 'oracle',
  password: 'oracle'
}
```

Drivers : exemple avec Oracle

```
oracle.connect(connectData, function(err, connection) {  
  if (err) {  
    console.log('Error connecting to db:', err);  
    return;  
  }  
  
  connection.execute('SELECT systimestamp FROM dual', [],  
    function(err, results) {  
      if (err) {  
        console.log('Error executing query:', err);  
        return;  
      }  
  
      console.log(results);  
      // à ne déclencher que lorsque la requête est terminée  
      connection.close();  
    });  
});
```



Drivers : exemple avec PostgreSQL

```
const pg = require('pg');

let client = new pg.Client({
  user: 'postgres',
  password: 'postgres',
  host: 'localhost',
  database: 'postgres',
});

client.connect(function(err) {
  if (err) throw err;
  client.query('SELECT NOW() AS "theTime"',
    function(err, result) {
      if (err) throw err;

      console.log(result.rows[0].theTime);
      client.end();
    }
  );
});
```

Drivers : exemple avec MySQL

```
const MySQL = require('MySQL');

let pool = MySQL.createPool({
  host      : 'localhost',
  user      : 'root',
  password : 'root'
});

pool.query('SELECT 1 + 1 AS solution',
  function(err, rows, fields) {
    if (err) throw err;

    console.log('The solution is: ', rows[0].solution);
    pool.end();
  }
);
```



Drivers : MongoDB

```
const MongoClient = require('mongodb').MongoClient;

// Connect to the db
MongoClient.connect('mongodb://url/db', function(err, db) {
  if(err) { return console.dir(err); }

  let collection = db.collection('test');
  let docs = [{mykey:1}, {mykey:2}, {mykey:3}];

  collection.insert(docs, {w:1}, function(err, result) {

    collection.find().toArray(function(err, items) {});

    let stream = collection.find({mykey:{$ne:2}}).stream();
    stream.on('data', function(item) {});
    stream.on('end', function() {});

    collection.findOne({mykey:1}, function(err, item) {});

  });
});
```

Drivers : abstractions SQL

- Différents modules proposent d'uniformiser l'API
- Chacun propose une implémentation pour certaines bases
- **dbtool** : supporte MySQL, MsSql et oracle
- **jsdbc** : supporte MySQL, PostgreSQL, Oracle et SQLite ainsi que les transactions
- **Any-DB** : propose une solution modulaire
 - **any-db-MySQL**
 - **any-db-postgres**
 - **any-db-SQLite3**
 - **any-db-transaction**

ORM SQL

- Un ORM (Object-Relationnal Mapping) permet de gérer l'accès aux données à plus haut niveau
- Il existe de nombreuses solutions d'ORM pour Node.js
 - **ORM** (MySQL, PostgreSQL, Amazon Redshift, SQLite)
 - **light-orm** (MySQL, PostgreSQL, MSSQL, ...)
 - **Sequelize** (MySQL, PostgreSQL, SQLite, MariaDB)
 - **CORMO** (MySQL, MongoDB, SQLite3, PostgreSQL)
 - **Model** (PostgreSQL, MySQL, SQLite, Riak, MongoDB, LevelDB, In-memory, Filesystem)
 - **persist** (SQLite3, MySQL, PostgreSQL, Oracle)
 - **streamsSQL** (MySQL, SQLite3)

Sequelize

```
const Sequelize = require('sequelize');

let sequelize = new Sequelize('test', 'root', 'root', {
  dialect: 'MySQL', // ou 'SQLite', 'postgres', 'mariadb'
  port: 3306,
});

sequelize
  .authenticate()
  .complete(function(err) {
    if (err) throw err;
    console.log('Connection has been established successfully.');
  });

let User = sequelize.define('User', {
  username: {
    type: Sequelize.STRING,
    allowNull: false
  },
  password: Sequelize.STRING
}, {});
```



Sequelize : model

- Sequelize permet de définir le mapping des objets
- Nécessite de faire le lien entre
 - Les propriétés de l'objet
 - La colonne de la table
- Il est possible de définir les propriétés SQL
 - Type de la colonne
 - Contraintes : `null`, `default...`
- Sequelize propose une fonctionnalité `sync`
 - Permet de créer la structure dans la base
 - Gère la création et la mise à jour de la structure

Sequelize : sync, lecture, écriture

```
sequelize.sync({
  force: true
}).complete(function(err) {
  if (err) throw err;
  User.create({
    username: 'john',
    password: '1111'
}).complete(function(err, user1) {
  if (err) throw err;
  User.find({
    username: 'john'
}).complete(function(err, user2) {
  if (err) throw err;
  console.log(user1.values, user2.values);
})
})
})
});
```

Sequelize : sequelize-auto

- Sequelize fournit un outil pour générer le mapping à partir d'une base de données existante

```
$ npm install -g sequelize-auto  
...  
$ npm install -g pg      # for postgres  
$ npm install -g MySQL   # for MySQL  
...  
$ sequelize-auto -h localhost -d test -u test -o "./models"  
Executing (default): SHOW TABLES;  
Executing (default): DESCRIBE 'Users';  
Done!
```

- Ces fichiers peuvent être chargés par la fonction import

```
let Users = sequelize.import(__dirname + '/models/Users')
```

Mongoose

- Mongoose est un ORM pour MongoDB
- Il est très utilisé pour les applications Web Node.js
- Connection au serveur MongoDB

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');
```

- Définition du mapping des objets

```
const userSchema = new mongoose.Schema({
  username: String,
  password: String
});
const User = mongoose.model('User', userSchema);
```

Mongoose : Mapping

- Mongoose propose de nombreuses options de mapping
- Les types de données
 - String, Number, Date, Boolean, Sous-documents, Tableau, Mixed, ObjectId, Buffer
- Il est possible d'ajouter à un mapping
 - Des méthodes d'instance, statiques, getters et setters
 - Validations et intercepteurs
 - Defaults et required
 - Indexes
 - Plugins...
- <http://mongoosejs.com/docs/guide.html>

Mongoose : API

- Créer des objets et les sauvegarder

```
let user = new User({ username: 'john', password: '1111' });
user.save(function (err, user) {
  if (err) throw err;
  console.log('user', user)
});
```

- Requêtes

```
User.find({ username: 'john' }, function(err, user) {
  if (err) throw err;
  console.log('user', user);
})
```

```
User.find(function(err, users) {
  console.log('users', users);
})
```





Lab 9

Outilage et Usine Logicielle



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- *Outillage et Usine Logicielle*
- Node.js en mode Cluster
- Au delà de Node.js

Écrire une application Node.js

De nombreux outils existent pour développer du JavaScript :

- Les éditeurs de texte, certains étant plus riches que d'autres :
 - Notepad++
 - Sublime Text, très répandu parmi les développeurs Node.js
 - Atom, open source, écrit en Node.js
 - Visual Studio Code, open source, écrit en Node.js
- Les IDEs :
 - WebStorm, considéré comme une référence parmi les IDE pour le développeur web
 - Netbeans
 - Eclipse avec JSDT
 - Cloud9, disponible en local ou en ligne

Déboguer avec les logs

L'utilisation des logs est souvent la solution la plus rapide et la moins coûteuse à mettre en place.

```
let inc = 0;  
console.log('Avant incrementation ', inc);  
inc++;  
console.log('Après incrementation ', inc);
```

Sortie :

```
Avant incrementation 0  
Après incrementation 1
```

Cas d'utilisation :

- Débogage et affichage de valeurs simples avec utilisation des différents niveaux (log, warn, error...)
- Permet de ne pas impacter l'enchaînement des évènements

Déboguer avec le débogueur Node

Embarqué nativement dans Node.js

```
node debug server.js
```

Le code utilisé est le suivant :

```
let inc = 0;
debugger; // set a breakpoint in JS
inc++;
debugger;
```

Les commandes disponibles sont :

```
repl : permet d'accéder aux valeurs des différentes variables
next : va à la ligne suivante
cont : continue l'exécution jusqu'au prochain point d'arrêt
step : pour entrer dans la fonction à la ligne courante
out : pour sortir de la fonction actuelle
help : pour accéder à la liste des commandes disponibles
```

Déboguer avec le débogueur Node : une séquence

Déroulement d'un débogage :

```
pierre@formation > node debug server.js
< debugger listening on port 5858
connecting... ok
break in debugNodeDebugger.js:1
1 let inc = 0;
2 debugger; // set a breakpoint in JS
3 inc++;
debug> cont
break in debugNodeDebugger.js:2
1 let inc = 0;
2 debugger; // set a breakpoint in JS
3 inc++;
4 debugger;
debug> repl
Press Ctrl + C to leave debug repl
> inc
0
...
...
```

Déboguer avec le débogueur Node : une séquence

Déroulement d'un débogage (suite) :

```
...
debug> cont
break in debugNodeDebugger.js:4
  2 debugger; // set a breakpoint in JS
  3 inc++;
  4 debugger;
  5 });
debug> repl
Press Ctrl + C to leave debug repl
> inc
1
debug> cont
program terminated
```

Déboguer avec le débogueur Node

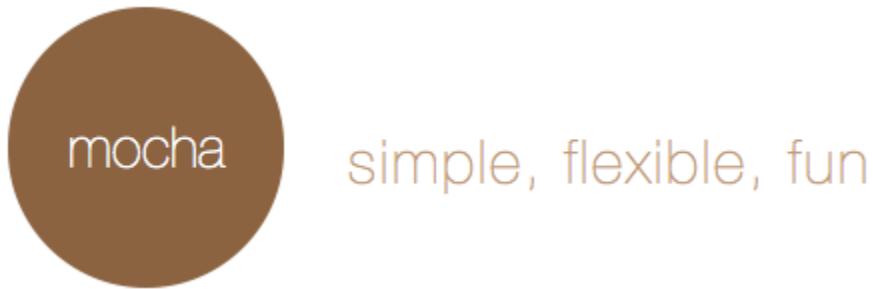
- Intégré nativement
- Fonctionnel avec les applications multi-fichiers (et les instructions require)
- Localisation facile des breakpoints via les instructions `debugger` du code
- Compatible avec les objets plus complexes et les appels imbriqués de méthodes
- Nécessite une étape de prise en main avant de manipuler le débogueur naturellement

Déboguer avec Node Inspector

- Node Inspector est une interface graphique pour le débogueur natif de Node.
- L'interface s'intègre, seulement, dans les developer tools de Chrome et Opera
- Permet de déboguer l'application dans un environnement familier et plus simple à prendre en main
- Disponible de base grâce à l'option `--inspect`

Tests

- Node.js dispose de très bons outils de test
- La nature souple du langage permet des notations élégantes
- Le framework de test le plus répandu est **mocha**



- Fournit une API simple et efficace
- Il est très léger et modulaire
- Il fonctionne également côté navigateur

Chai & Sinon.JS

- Mocha étant modulaire, il faut rapidement ajouter des librairies
- **Chai Assertion Library**
 - Librairie proposant différentes APIs pour les assertions
 - Trois styles possibles : **Should, Expect, Assert**
- **Sinon.JS** apporte de nombreux outils de test
 - des mocks, ce sont des objets programmables pour spécifier les attentes sur les appels qu'ils reçoivent
 - des stubs, ce sont des implémentations qui retournent toujours la même valeur pré-programmée
 - des spies, qui permettent de tracer les appels aux méthodes

Illustration

Pour les exemples de tests, nous allons considérer la base de code suivante

```
class RequestHandler {  
    onSuccess() {  
        return Math.random();  
    }  
    handleRequest() {  
        return this.onSuccess();  
    }  
}
```

Illustration avec Mocha

```
var assert = require('assert'); // librairie d'assertions

describe('Test using mocha', function() {
  let rh;
  beforeEach(function() {
    // Avant chaque test : réinitialisation du RequestHandler
    rh = new RequestHandler();
  });

  // Premier test avec une syntaxe lisible
  it('should handle the request and return a number
between 0 and 1', function () {
    let result = rh.handleRequest();
    assert(result < 1 && result >= 0);
  });
});

//$ mocha test.js
//-> 1 passing (4ms)
```

Illustration avec Sinon.JS

Utilisation des spies :

```
describe('Test using sinon.js', function() {
  let rh;
  beforeEach(function() {
    rh = new RequestHandler();
  });

  it('should call the onSuccess function once', function () {
    let spy = sinon.spy(rh, 'onSuccess');

    // Call the method which is going to call our onSuccess method
    rh.handleRequest();

    // Check we called it once
    assert(spy.calledOnce);
  });
});

//-> 1 passing (6ms)
```



Illustration avec SinonJS

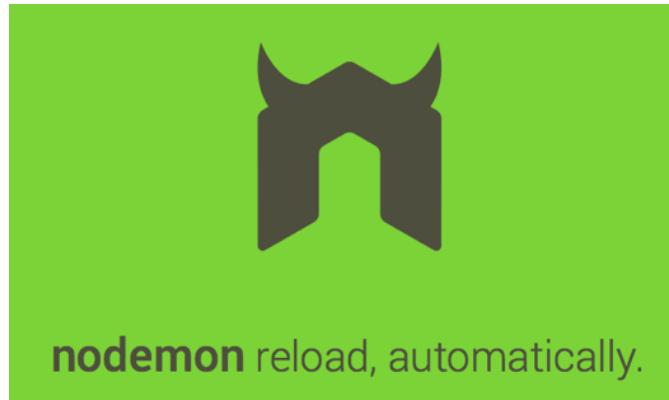
Utilisation des stubs :

```
describe('Test using sinon.js', function() {
  let rh;
  beforeEach(function() {
    rh = new RequestHandler();
  });
  it('should return one through mock request handler',
    function () {
      let stub = sinon.stub(rh, 'onSuccess',
        function() {
          return 1;
        }
      );
      let result = rh.handleRequest();
      // Assert that onSuccess returns 1
      assert(result, 1);
      // Assert that the method is called once
      assert(stub.calledOnce);
    }
  );
});
```

Autres outils de test

- Jasmine
 - Mocha like (langage similaire)
 - Plus répandu côté front-end
- Karma est un exécuteur de test développé par la team AngularJS
 - Permet de jouer un ensemble de tests (éventuellement automatiquement après chaque modification de fichier)
 - Compatible Mocha ou Jasmine

Nodemon



- Nodemon propose de surveiller vos fichiers sources
- De relancer votre application à chaque changement
- Très utile pour la phase de développement

```
$ npm install -g nodemon  
$ nodemon script
```

Build

- Les projets Node.js ont peu de tâches de build
 - Pas d'étape de compilation du JavaScript
 - Pas d'étape d'optimisation comme pour le frontend
- Il reste pourtant souvent des tâches à automatiser
 - Transpileur (CoffeeScript, Traceur...)
 - Lancer les tests
 - Piloter nodemon
 - Générer un livrable
 - Déploiements...

Grunt



The Javascript Task Runner

Grunt, the JavaScript task runner

- Une des applications Node.js les plus installées !
- Permet d'automatiser toutes vos tâches répétitives (ou presque)
- Description des tâches dans un fichier `Gruntfile.js`
 - Principalement du JSON, mais est en fait du Node.js
- Un exécutable en ligne de commande : `grunt task`
- De nombreux plugins apportent des tâches pré-définies
- Possibilité de définir à volonté de nouvelles tâches en Javascript
 - APIs pour manipuler les fichiers, le logging, le reporting...

Grunt, installation

- Installer l'exécutable en ligne de commande
 - Ajoute la commande `grunt` au path de l'OS.

```
npm install -g grunt-cli
```
- Installer Grunt sur vos projets
 - Permet de travailler avec des versions différentes

```
npm install --save-dev grunt
```

Syntaxe du Gruntfile

```
module.exports = function(grunt) {
  grunt.initConfig({
    //Déclaration d'une tâche
    task1: {
      options: { /*...*/ },
      target1: {
        src: '...',  
dest: '...',  
},
        target2: { /*...*/ },
      },
      task2: { /*...*/ },
    });
  // Charge la tâche npm "grunt-task1"
  grunt.loadNpmTasks('grunt-task1');
  // Charge la tâche à partir du fichier "task2.js"
  grunt.loadTasks('task2');
  // Déclarer une tâche composite : task1.target2 => task2
  grunt.registerTask('default', ['task1:target2', 'task2']);
};
```



Syntaxe du Gruntfile

- Une **tâche** est un objet contenant :
 - Une ou plusieurs cibles
 - Zéro ou un objet *options* qui valorise les options par défaut pour toutes les cibles
- Une **cible** est un objet contenant :
 - Zéro ou un objet *options* qui surcharge les options de la tâche pour cette cible seulement
 - des paramètres (en général des chemins source et destination)
- Se reporter à la documentation de la tâche pour les options et paramètres utilisables

Syntaxe du Gruntfile

```
grunt.initConfig({  
  copy: { // tâche  
    assets: { // cible  
      src: 'assets/**', // paramètre  
      dest: 'target/' , // paramètre  
    },  
  },  
});  
  
grunt.loadNpmTasks('grunt-contrib-copy');
```

```
$ ls  
assets  Gruntfile.js  
$ grunt copy  
Running "copy:assets" (copy) task  
Created 1 directories, copied 1 file  
Done, without errors.  
$ ls target  
assets
```

Plugins

- Grunt n'embarque aucune tâche pré définie
- Ces tâches sont disponibles sous forme de plugins
 - 25 sont maintenus par l'équipe Grunt : `grunt-contrib-*`
 - Des dizaines d'autres par la communauté
- Installation :
 - `npm install --save-dev <plugin>`
 - `grunt.loadNpmTasks('<plugin>')`

Plugins

- Concaténer : `grunt-contrib-concat`
- Minifier : `grunt-contrib-uglify`
- Compiler du Sass : `grunt-contrib-sass`
- Optimiser des images : `grunt-contrib-imagemin`
- Extraire de la documentation JSDoc : `grunt-jsdoc`
- Lancer des tests Jasmine : `grunt-contrib-jasmine`
- Déployer en FTP : `grunt-ftp-deploy`
- ...

Automatisation de la production des livrables

Avec Node nous pouvons désormais :

- Tester nos applications
- Automatiser le lancement des tests lors de la production de livrables
- La prochaine étape serait donc :
 - Builder automatiquement les livrables et lancer les tests à chaque nouvelle version sur le VCS.

Une solution bien connue dans le monde Java pour cette problématique s'appelle Jenkins.

Intégration continue

La mise en place de l'intégration continue (IC) se fait par une succession d'étapes :

1. Installation de NodeJS et du client Grunt sur la machine d'intégration continue
2. Configuration d'une tâche Grunt qui sera appelée par un outil d'intégration continue (Jenkins, Travis CI...) et qui doit :
 - Lancer les tests
 - Produire les rapports de tests dans un format interprétable par la plate-forme d'IC
3. Création du job qui va récupérer les sources, lancer la tâche Grunt et lire les rapports produits

Intégration avec Jenkins

- Installation de NodeJS et du client Grunt sur la machine d'intégration continue
 - Suivre les étapes présentées dans les parties précédentes
- Configuration d'une tâche Grunt qui sera appelée par Jenkins

```
module.exports = function (grunt) {
  grunt.loadNpmTasks('grunt-contrib-clean');
  grunt.loadNpmTasks("grunt-mocha-test");
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    clean: ['report/*'],
    mochaTest: {
      /* Slide suivante */
    }
  });
  grunt.registerTask('jenkins', ['clean', 'mochaTest']);
};
```

Intégration avec Jenkins

Configuration d'une tâche Grunt qui sera appelée par Jenkins

```
mochaTest: {  
  jenkins: {  
    src: ['test/*.js'],  
    options: {  
      reporter: 'xunit',  
      quiet: true,  
      captureFile: 'report/test-reports.xml'  
    }  
  }  
}
```

- Exécution des tests contenus dans les fichiers js du dossier test
- Cette tâche génère des rapports xUnit
 - Format XML - similaire aux rapports de tests JUnit
 - Compréhensibles par Jenkins

Intégration avec Jenkins

Création du job Jenkins :

- Créer un nouveau projet free-style
- Configurer la récupération des sources par CVS, Git ou SVN
- Ajouter une étape de build sous forme de script shell

```
npm install  
grunt jenkins
```

- Ajouter une étape post-build **Publier le rapport des résultats des tests JUnit**
 - La valeur XML des rapports de tests correspond à celle du Gruntfile : **report/test-reports.xml**
- Sauvegarder le job et le lancer.

Intégration avec Jenkins

Grâce à Jenkins, il est possible de

- **scruter** un VCS et de construire automatiquement le projet à chaque modification
- d'être prévenu en **temps réel** du statut du projet (build successful ou failed) suivant le résultat des tests
- de déployer **automatiquement** en cas de succès les livrables sur un environnement de recette





Lab 10

Node.js en mode Cluster



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- *Node.js en mode Cluster*
- Au delà de Node.js

Node.js en cluster

- **Avant** même d'envisager de créer une application Node.js distribuée sur **plusieurs machines physiques**
 - La nature **mono-threadée** de Node lui permet de monter en puissance dans la limite d'**un cœur du processeur**
 - Pour utiliser tous les cœurs du processeur de la machine, il faut lancer autant de processus que de cœurs
 - Mais si on les lance indépendamment
 - Il faut un système de gestion du cluster
 - Il faut qu'ils puissent communiquer les uns avec les autres
 - Il faut qu'ils puissent partager les mêmes ressources
- Exemple : une socket TCP

Module cluster : contexte

- Il existe un module noyau : **Cluster**
 - Il est en stabilité **stable**
 - <http://nodejs.org/api/cluster.html>
- Il existe aussi des modules NPM
 - **node-cluster-app**
 - **cluster-bomb**
 - **fork-pool**
- **Rappel** : Il ne s'agit pas ici de cluster multi-machines
 - On utilisera alors le plus souvent un **Redis**
 - C'est le cas pour **Connect** et **Socket.IO**

Module cluster : objectif

- Un cluster Node.js est constitué de plusieurs processus indépendants
 - Ils ne partagent aucune zone mémoire
 - Ils peuvent par contre partager les socket réseau
 - La distribution entre les processus peut être en question
- Le module cluster va permettre
 - De différencier un **master** et des **workers**
 - Proposer au master de créer des workers
 - Permettre l'échange de messages entre workers

Master & Workers

- `cluster.setupMaster(configuration)`
 - Définit la configuration principale
 - Ne peut être appelé qu'une seule fois
 - Possibilité de configurer un autre script pour les workers
- `cluster.fork(env)`
 - Démarre un nouveau worker
 - Il s'agit alors d'un sous-processus
 - La communication entre processus fonctionne par [IPC](#) (Inter-Processus Communication)

Master & Workers : API

- Savoir se positionner dans le cluster
 - `cluster.isMaster`, `cluster.isWorker`
 - `cluster.workers` : seulement depuis le master
 - `cluster.worker` : worker courant depuis un worker
- Gestion des serveurs TCP
 - Les commandes `listen` sont interceptées par le cluster
 - Le master gère la connexion et dispatche les messages
- `cluster.disconnect(callback)`
 - Lance `worker.disconnect()` sur tous les workers

Module cluster : exemple

```
const cluster = require('cluster');
const cpuCount = require('os').cpus().length;

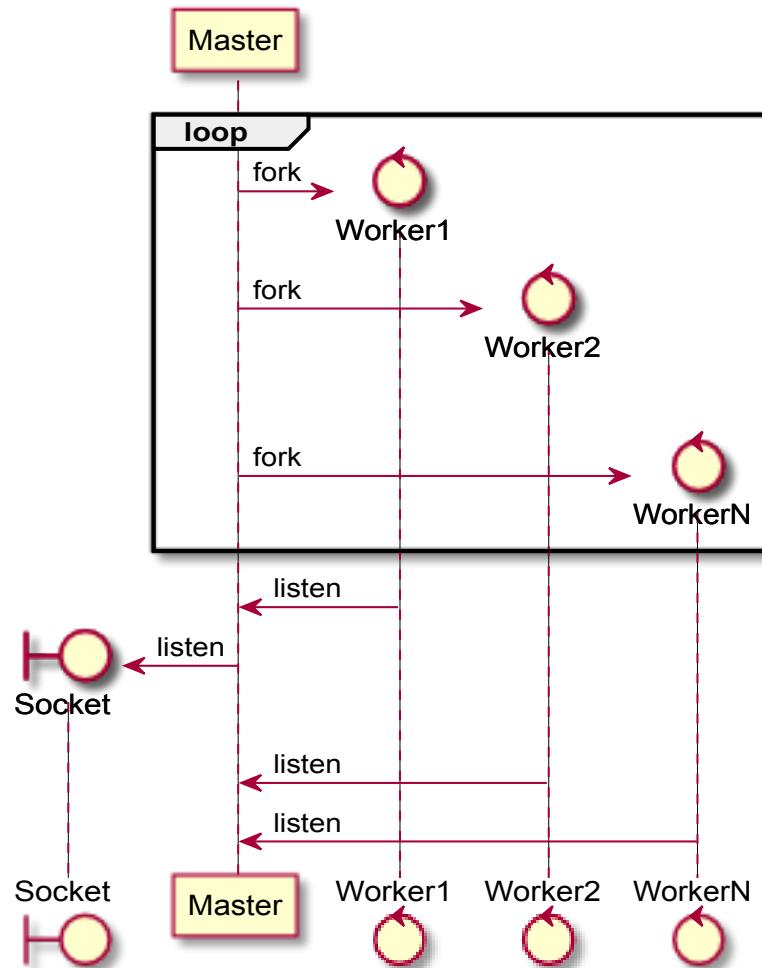
if (cluster.isMaster) {
  for (var i = 0; i < cpuCount; i++) {
    cluster.fork();
  }
} else {
  const http = require('http');
  server = http.createServer(function(req, res) {
    res.writeHead(200);
    res.end('hello world\n');
  });
  server.listen(3000);
}
```



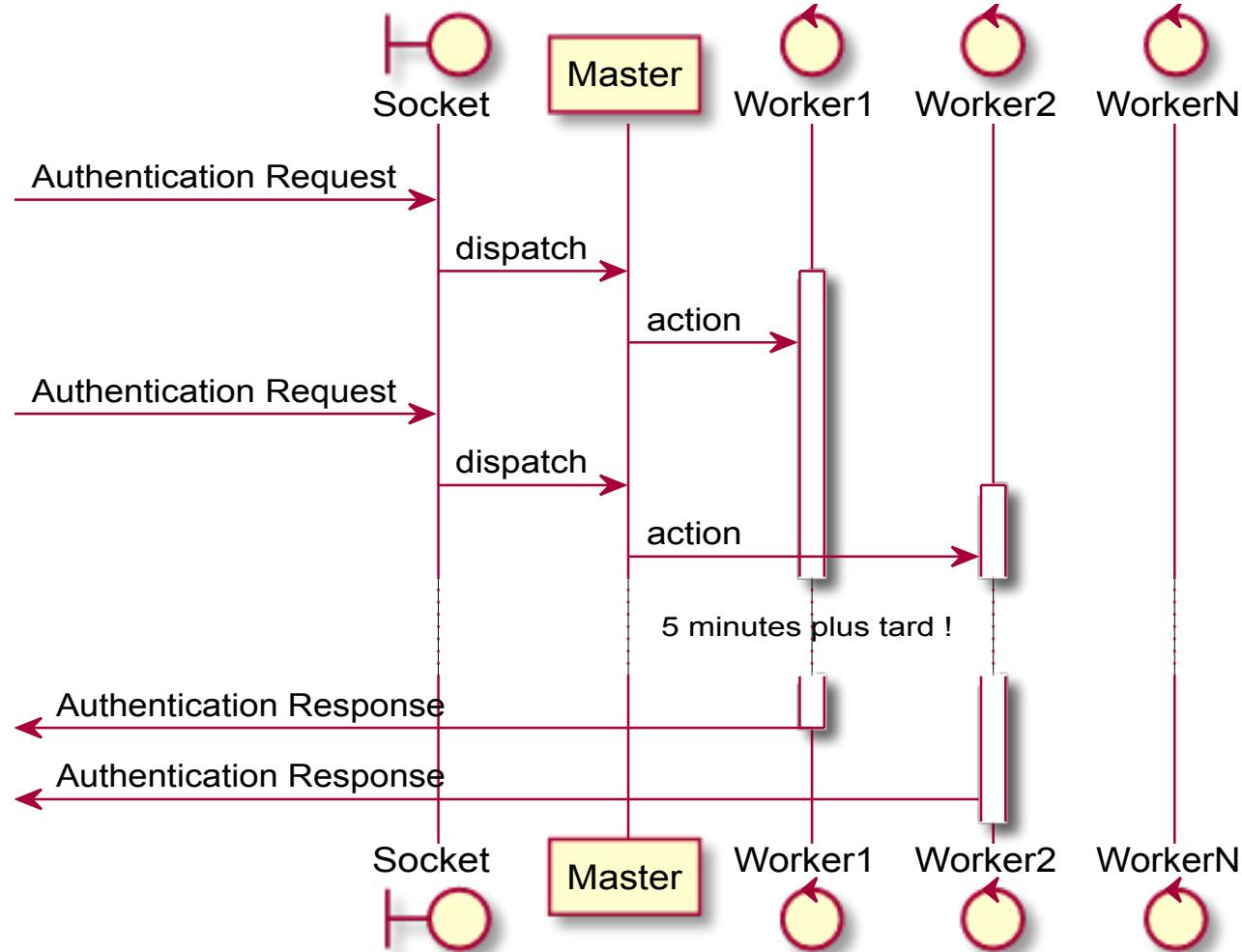
Cycle de vie

- Le worker peut s'appuyer sur des évènements
 - `setup`, `fork`, `online`, `listening`, ...
- L'objet worker fournit des informations
 - `worker.id` : contient un id unique du process
C'est la clef du worker dans workers
 - `worker.process` : le processus du worker
 - `worker.exitedAfterDisconnect` : détermine s'il est mort proprement
- L'objet worker peut aussi servir au pilotage
 - `worker.kill(signal)` : lance un signal au worker
 - `worker.disconnect()` : demande l'arrêt

Cycle de vie : démarrage



Partage de connexion



Messaging

- Le messaging IPC est le seul moyen de collaboration
- Il n'y a aucune mémoire partagée entre les processus
- Le protocole de communication est basé sur IPC
- C'est une communication bas niveau dans les OS
- La communication est synchrone et peu performante
- Les workers ne peuvent pas parler entre eux
- Elle peut servir au pilotage du cluster mais **pas pour l'applicatif**

Messaging : API

- `worker.send(message[, sendHandle])`
 - Envoi d'un message depuis le master à un worker
 - C'est un alias de `childProcess.send`
 - `sendHandle` peut être un serveur TCP
- `process.send(message)`
 - Envoi d'un message au processus parent
 - Dans le cas d'un worker, au master
- `process.on('message', callback)`
 - Écoute de messages IPC (master & worker)

Messaging : exemple

```
/** clusterMaster.js **/

const _ = require('lodash');
const cluster = require('cluster');
const cpuCount = require('os').cpus().length;

cluster.setupMaster({ exec : 'clusterWorker.js' });

for (var i = 0; i < cpuCount; i++) { cluster.fork(); }

console.log('Hello I\'m the master!');

_.each(cluster.workers, function(worker) {
  worker.on('message', function(data) {
    console.log('Master received message from worker',
      worker.id, ':', data);
  });
  worker.send('echo');
});
```



Messaging : exemple

```
/** clusterWorker.js **/


const cluster = require('cluster');
const worker = cluster.worker;


console.log('Hello I\'m the worker', worker.id);


process.on('message', function(data) {
  process.send(data + ' from ' + worker.id);
});



/*-> Hello I'm the master!
Hello I'm the worker 1
Master received message from worker 1 : echo from 1
Hello I'm the worker 2
Master received message from worker 2 : echo from 2
Hello I'm the worker 3
Hello I'm the worker 4
Master received message from worker 3 : echo from 3
Master received message from worker 4 : echo from 4 */
```

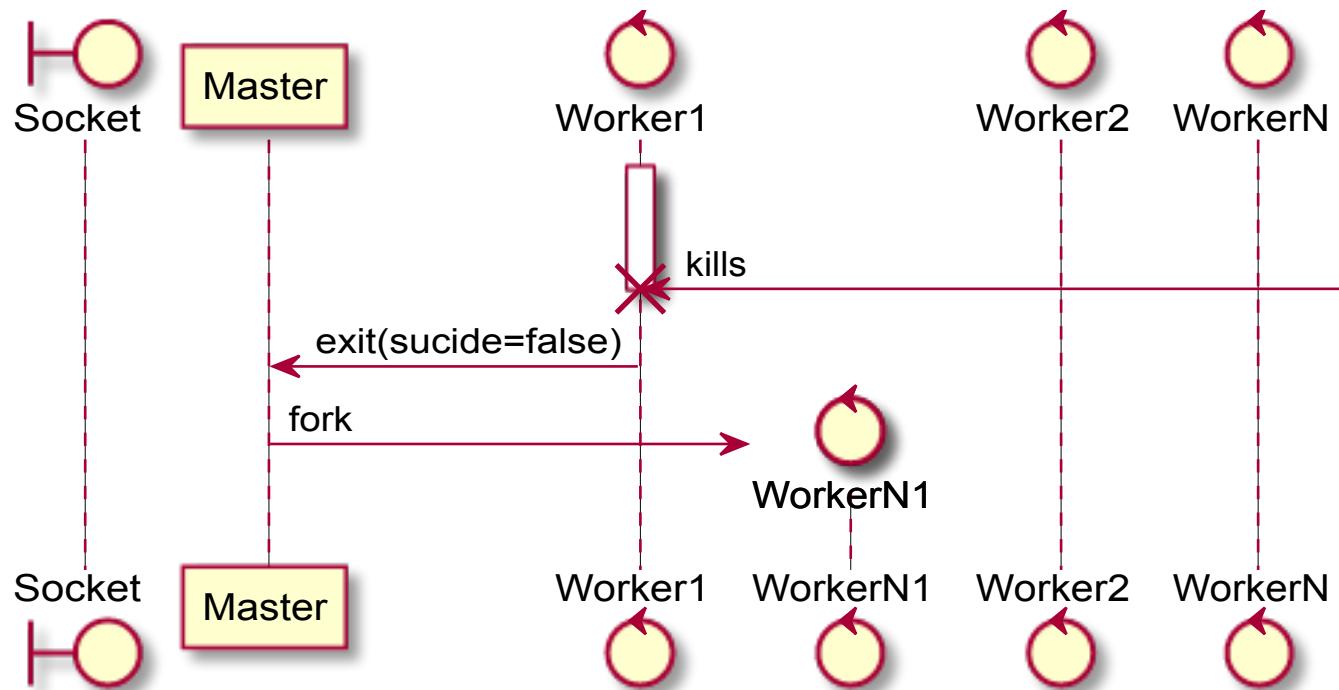


Messaging : applicatif

- Plusieurs stratégies pour les données applicatives
 - Tout faire passer par le master, peu conseillé
 - Monter un système de P2P, semble complexe !
 - Stocker dans une base NoSQL type Redis
 - Utiliser une système externe de messaging
- Les systèmes de messaging
 - **RabbitMQ, ActiveMQ, ...**
- Les bases NoSQL
 - **Redis, MemcacheDB, ...**

Gestion des erreurs

- En cas d'arrêt non sollicité d'un Worker, le Master peut redémarrer de nouveau un sous-processus



Gestion des erreurs : exemple

```
const cluster = require('cluster');

/* ... */

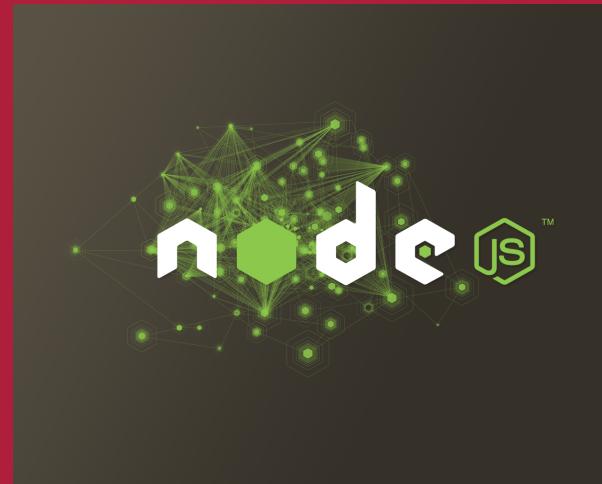
cluster.on('exit', function(worker, code, signal) {
  if(worker.exitedAfterDisconnect) {
    console.log('worker ', worker.process.pid , ' exit');
  } else {
    console.log('worker ', worker.process.pid , ' killed');
    cluster.fork();
  }
});
```





Lab 11

Au delà de NodeJS



Plan

- Rappel des bonnes pratiques JavaScript
- Introduction à Node.js
- Architecture de Node.js
- Modules et gestion de dépendances
- Node et le Web : HTTP, Connect & Express
- L'asynchrone en détails
- Communication temps réel
- La gestion des streams
- Liaison avec la persistance des données
- Outilage et Usine Logicielle
- Node.js en mode Cluster
- *Au delà de Node.js*

MEAN



- **MongoDb / Express / AngularJS / Node.js**
- <http://mean.io/>
- <http://meanjs.org/>

MEAN

- Que représente MEAN ?
 - Une stack technologique **full JavaScript**
 - Des technologies qui forment un ensemble **cohérent**
 - Des projets **seed (graine)** pour démarrer un projet
- Le JSON est la clé
 - **MongoDB** stocke les données en JSON
 - **Node.js** manipule nativement le JSON
 - **Express** transfère facilement le JSON en HTTP
 - **Angular** lie les données JSON au HTML

MEAN

- Epaillé par l'écosystème Node.js
 - **Mongoose** pour l'ORM Mongo
 - **Passport** pour l'authentification
- La stack MEAN permet
 - De démarrer rapidement sur une base Node.js
 - D'amener des développeurs à partir sur Node.js
 - De les ouvrir à tous les points forts de Node.js
 - De proposer une solution pour le grand public
 - Très facile à déployer dans le cloud
 - Pourrait concurrencer PHP ?

Cloud

- L'hébergement de Node.js dans le cloud est facile
- Énormément d'hébergeurs proposent Node.js
 - Heroku, Joyent, Nodejitsu
 - Windows Azure, Cloud Foundry, Gandi
 - Amazon Web Services...
- En fonction des solutions, il s'agira de
 - **PaaS : Platform as a Service** (Heroku...)
 - **IaaS : Infrastructure as a Service** (Amazon...)

Heroku

- Presque aussi simple qu'un `git push`
- Créer un fichier `Procfile`

```
web: node index.js
```

- Le port HTTP se trouve dans les variables d'environnement

```
const port = process.env.PORT || 3000;
```

- Ajouter Heroku comme remote dans git

```
heroku git:remote -a identifiant-git-heroku
```

- **Pusher** sur Heroku

```
git push heroku master
```

AWS

- Avec Amazon WebServices, la procédure est plus bas niveau
- Il faut administrer un serveur et installer le nécessaire
- Il faut disposer d'un compte AWS (nécessite un numéro de CB)
- Instancier un **Serveur Web** par la console de management



- Choisir une machine **Amazon Linux 64 bits**
- Lancer la nouvelle machine
- Se connecter à la machine en SSH

- Installer l'environnement sur la machine

```
$ sudo yum install gcc-c++ make  
$ sudo yum install openssl-devel  
$ sudo yum install git  
$ git clone git://github.com/joyent/node.git  
$ cd node
```

- Installer **Node.js** sur la machine

```
$ git checkout v0.10.26 # ou une autre version  
$ ./configure  
$ make  
$ sudo make install
```

AWS

- Sauvegarder votre instance comme modèle (AMI)

The screenshot shows the AWS Management Console interface for managing snapshots. At the top, there are several buttons: 'Create Snapshot' (highlighted in red), 'Delete', 'Permissions', 'Create Volume', 'Create Image', and 'Copy'. Below these are navigation icons: back, forward, refresh, settings, and help. A dropdown menu 'Viewing:' set to 'Owned By Me' and a search bar are also present. The main area displays a table of snapshots:

Name	Snapshot ID	Capacity	Description	Status	Started	Pro
empty	snap-c9d88f8e	8 GiB	Created by CreateImage(i-XXXXXX) for ami-5ee27837 from vol-566cc025	completed	2013-03-28 13:18 PDT	ava

- Installer son application sur la machine
 - scp, git clone, npm install...
- Installer l'application en tant que service
- Lancer l'application

Monitoring

- Il existe des modules de monitoring pour Node.js
- Ils sont particulièrement intéressants pour les clusters
- Les principaux sont **pm2** et **NewRelic**



Monitoring : pm2

- L'installation se fait par la commande suivante :

```
npm install pm2 -g
```

- **ping** : vérifie la présence du master et peut le relancer
- **web** : démarre un serveur web accessible sur le port 9615
- **start script** : démarre l'application en mode cluster monitoré
- **stop script** : arrête l'application
- **monit** : présente un écran de surveillance des process

Monitoring : NewRelic

- Il faut passer par la création d'un compte (<http://newrelic.com/>)
- Installer le module NewRelic

```
npm install newrelic
```

- Copier `newrelic.js` du dossier `node_modules/newrelic` dans le dossier racine de l'application
- Editer le fichier `newrelic.js` pour remplacer la clef de licence

```
license_key : 'license key here',
```

- rajouter la commande suivante en première ligne de l'application

```
require('newrelic');
```

- Lancer l'application puis aller sur le site de NewRelic pour monitorer l'application

