NODE.JS

APPLICATIONS TEMPS RÉEL ET PERFORMANTES

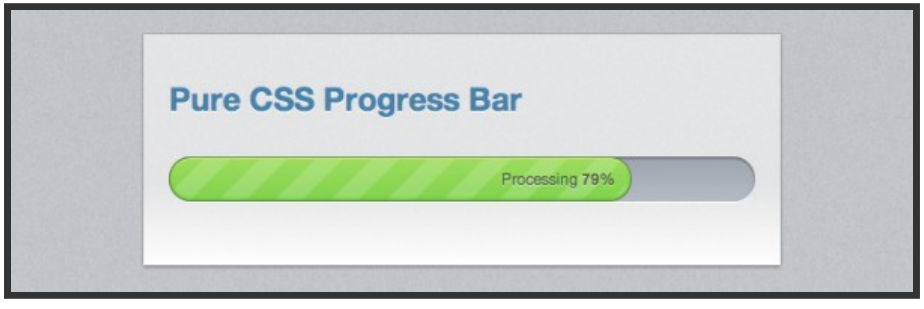
PLAN (THÉORIQUE)

- Jour 1 : **Node et son écosystème**
  - Présentation
  - Modules, npm et tests unitaires
  - Le paradigme asynchrone
- Jour 2 : **Application web**
  - HTTP et Express
  - Bases de données
  - Couche de configuration
- Jour 3 : **Utilisation avancée**
  - WebSockets
  - Qualité et performances
  - Conclusion, bonnes pratiques…

HISTORIQUE

INSPIRATION

2006 - FLICKR EFFECT



A cette époque Ryan Dahl est un étudiant en mathématique parcourant l'Europe.

Il prend conscience de l'impact du temps réél en observant la barre de progression de l'upload de photos sur Flickr qui se met à jour toute seule.

Les websockets sont encore un doux rêve, des techniques moins glorieuses doivent être utilisées (comet, long polling, flash). Elles seront évoquées avec Socket.io

FIRST COMMIT - V0.0.3 - JUIN 2009

> *" To provide a purely evented, non-blocking infrastructure to script highly concurrent programs. "*

Node porte ce nom pour le découplage qu'il représente, ou des unités atomiques travaillent ensemble pour accomplir des taches importantes.

Paradoxalement, Ryan avait Ruby (EventMachine) en tête et ce n'est que suite à une heureuse prise de conscience que JavaScript fut choisi.

JS CONF EU - 8 NOV 2009

LATENCES I/O

- L1: 3 cycles
- L2: 14 cycles
- RAM: 250 cycles
- Disque: 41 000 000 cycles
- Réseau: 240 000 000 cycles

De nature assez timide et peu à l'aise, Ryan Dahl arrive à convaincre une audience qui croyait avoir tout vu.

Pourtant, executer du JS coté serveur n'est pas un concept nouveau puis qu'il est prévu depuis l'inception même du langage avec LiveWire tournant sur le Netscape Entreprise Server.

FINANCEMENT

Ryan n'a plus d'économie pour mener à bien le projet, il cède le copyright à Joyent dont il devient un employé.

Le projet déjà *hype* dans la communauté open source commence à se crédibiliser dans le monde de l'entreprise.

DISPARITION

30 JANVIER 2012

Le BDNFL devient Isaac Schlueter, le créateur de NPM. Ryan Dahl disparait mysterieusement dans la nature vers d'autres conquêtes.

Desinteret, Burn Out ?

LA 0.10 DE FONTAINE

15 JANVIER 2014

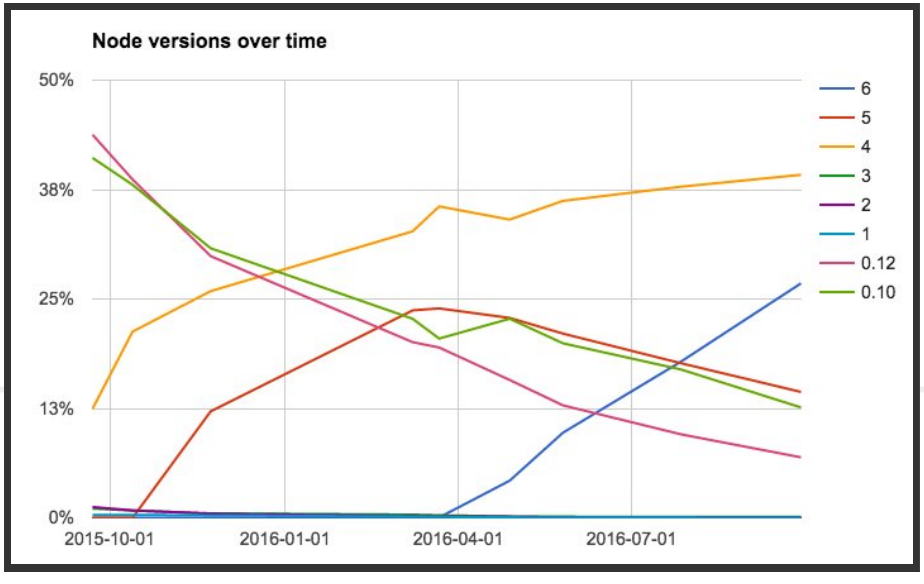
Le registry NPM commence à souffrir et les lenteurs sont de plus en plus récurrents malgré les miroirs.

Isaacs quitte le bateau pour se focaliser sur NPM inc, TJ Fontaine reprend les rennes d'une version qui s'enlise pendant 2 ans et va conduire malgré sa bonne volonté au schisme IO.JS

VERSIONS ACTUELLES

4.6.0 LTS

6.7.0 STABLE



PRÉSENTATION

Node.js c'est d'abord du…

JAVASCRIPT

- Langage de facto du web
- **Des inconvénients**:
  - Typage dynamique (conversions hasardeuses)
  - `this` indomptable
- **De sérieux atouts**:
  - Typage implicite
  - Des interpréteurs hyper efficace
  - Asynchrone par nature
  - Vivant (EcmaScript)

Node.js c'est aussi…

V8

- Interpréteur JavaScript de Chrome
- Développé par Google
- Open-source
- Très performant (mais la course reste ouverte)

Les équivalents chez Mozilla sont les monkeys : SpiderMonkey, IonMonkey, JaegerMonkey. Trident sur IE, mais qui va laisser la place à Spartan pour la version 12.

Node.js multiplateforme ?

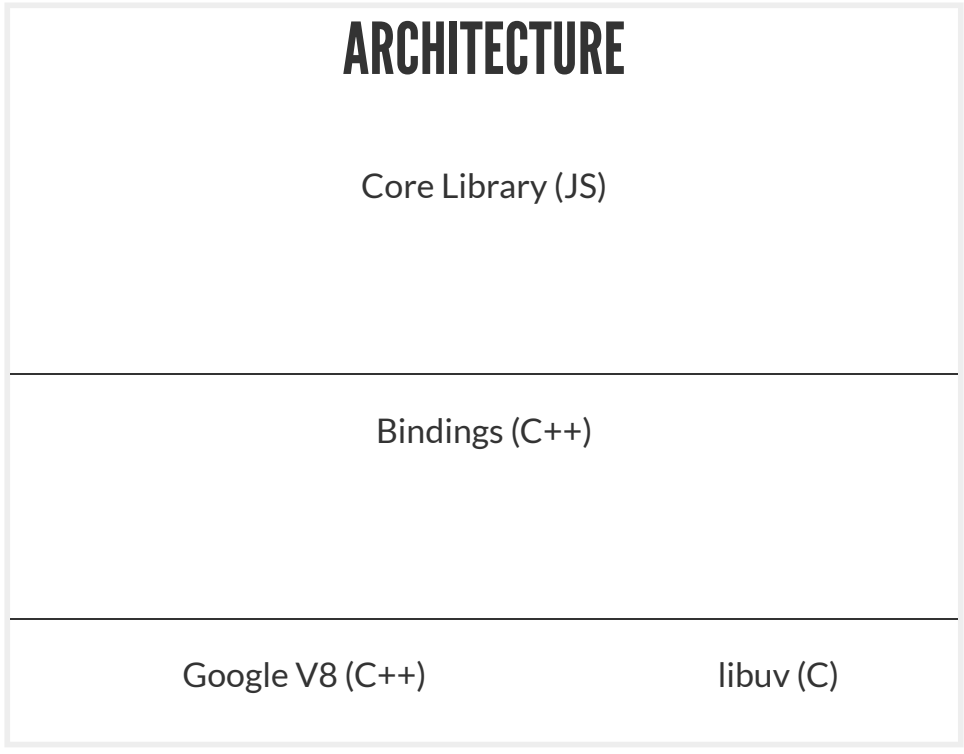
LIBUV

- Couche d'abstraction en C
- Multi OS
- En charge de l'event loop
- Aussi utilisée hors node (luvit)

Initialement écrite par Bert Belder (membre du TC) C'est grâce à cette brique que le support de Windows est possible.

Précedement: epool, kqueue libeio, libev, IOCP

Bientôt utilisée dans neovim.



En explorant l'arborescence du project sur GitHub on retrouve ces catégories. Parmi les briques écrites en C++ figure http_parser de ry.

INSTALLER NODE.JS

- Niveau système
  - Par l'installateur officiel (le plus simple sous Windows)
  - Par les paquets (brew, apt, pacman, etc.)
  - Compiler depuis les sources (long)
- Niveau utilisateur : nvm, nave
  - Ne nécessite pas npm ou node pour être installé
  - Basculer entre les versions de node

```
curl $URL | bash
```

Warning sur `curl|bash` quand-même.

Marche moyennement sous Windows, tester nvmw.

```
nvm install $version
# x, x.y, x.y.z, stable, unstable, iojs
nvm use $version
nvm run $version [script.js]

nvm ls
nvm ls-remote
```

LE REPL

```
node
```

- **R**ead
- **E**valuate
- **P**rint
- **L**oop

```
> 1+1
2
> 'node'.toUpperCase()
'NODE'
```

Pratique pour tester des bouts de code, vérifier des comportements JS…

Les modules du cœur sont pré-chargés. En revanche pas d'accès au DOM evidement, faire le test avec `window`.

Explorer le global : `Object.keys(global)`

Noter que `Buffer` est directement dispo.

EXÉCUTER UN SCRIPT

Le REPL c'est bien, interpréter un fichier script c'est mieux :

```
// hello-world.js

var world = "world";

console.log("Hello, %s", world);
```

```
node hello-world.js
```

```
Hello, world
```

EXERCICE

Écrire le script `add.js`

TP

- Présentation de `process.argv`
  - `0` = chemin vers l'exe node
  - `1` = chemin vers le script
  - `2` = début des arguments
- En profiter pour parler de `process` (versions etc...)
- Puis de `global`
- Note: `#!/usr/bin/env node` est converti par npm pour que ça marche bien sous Windows
- Faire la version avec un simple `for` puis parler de fonctionnel

```
var sum = 0;

for (var i = 0; i < process.argv.length; i++) {
  var n = Number(process.argv[i]);
  if (isNaN(n)) {
    continue;
  }
  sum += n;
}
```

```
var args = process.argv.slice( 2);
var numbers = args.map( Number).filter( function (n) {
  return !isNaN(n);
});
// ici on peut aussi créér une function sum generique
var sum = args.reduce( function (_sum, n) {
  return _sum + n;
});
```

MODULES

MODULES COMMONJS

UN MODULE = UN FICHIER

- Charger un module: `require(…)`
- Définir un module: `module.exports.… = …`

```
// monmodule.js
module.exports = {
  hello: function () {
    return "Hello, world";
  }
};
```

```
// script.js
var mod = require("/path/to/monmodule.js");
console.log(mod.hello());
```

- Spécification: http://wiki.commonjs.org/wiki/Modules/1.0
- Chaque module est isolé dans son scope (aller voir dans src/node.js)
- Penser à `"use strict"`
- Noter que `require("./mod.js") === require("./mod")`

VARIABLE `MODULE`

- Une variable "module" est toujours définie
- `exports === module.exports`
- Possibilité d'exporter une API directement avec `module.exports = …`

Factory

```
function (exports, require, module, __filename, __dirname  ) {
  const foo = 'bar';
  module.exports.bar = bar;
}
```

MODULES COMMONJS

UN MODULE = UN FICHIER

- Écrivons nos premiers modules !

```
// cli-args.js
…

// sum.js
…
```

- Utiliser dans le REPL
- Utiliser dans un script

TP

- `cli-args.js` = exposer `process.argv.slice(2)`
- `sum.js` = exposer une fonction qui somme un tableau
- Jouer avec dans le REPL
  - Observer qu'il faut relancer le REPL pour "recharger" un require
- Réécrire le script de tout-à-l'heure avec ces deux modules

TOUJOURS DÉMARRER SES MODULES PAR `"use strict"`

- Éviter les globales involontaires
- Éviter `this === global`
- http://301.tl/moz-fr-strict

MODULES COMMONJS

UN MODULE = UN DOSSIER

Quand un module prend de l'ampleur…

- Découper l'implémentation en modules internes
- Tout en gardant le `require` simple

```
monmodule/
  index.js
  sous-module1.js
  sous-module2.js
```

TP OU DÉMO

- Réécrire notre script sous forme de script `index.js` avec `cli-args.js` et `sum.js` des sous-scripts
- Noter que `node dossier` fonctionne
- Définir une API
- Jouer avec le REPL

SPÉC. *PACKAGES*

LE FICHIER `PACKAGE.JSON`

```
{
  "name": "monmodule",
  "version": "1.0.0",
  "main": "index.js",
  "dependencies": {
    "sum": "^1.0.0"
  }
  …
}
```

- Format complet : http://wiki.commonjs.org/wiki/Packages/1.0#Required_Fields

TP OU DÉMO

- Écrire un `package.json` simpliste pour notre module
- Modifier le nom de `index.js` et modifier "main" en fonction
  - Noter que `node dossier` fonctionne toujours

LES MODULES DU CŒUR

- `require("…/fichier.js")` = module personnel
- `require("module")` = module du cœur de Node.js

LES PRINCIPAUX

- Fichiers: `fs`, `path`
- Réseaux: `http`, `net`
- Système: `os`, `child_process`
- Utilitaires: `util`

- `assert`: assertions
- `buffer`: buffer (zone mémoire hors heap)
- **`child_process`**: exécuter des process
- **`cluster`**: passer son app en cluster
- `crypto`: cryptographie
- `dgram`: sockets UDP
- `dns`: requêtes DNS
- **`domain`**: zones de code isolées
- **`events`**: classe `EventEmitter`
- **`fs`**: manipulation des fichiers
- **`http`**: client/serveur HTTP
- `https`: client/serveur HTTPS
- `net`: sockets
- `os`: infos système
- `path`: manipulation des chemins
- `punycode`: convertisseur Punycode (NDD internationalisés)
- `querystring`: parsing de query-string
- `readline`: lecture de stream ligne par ligne
- `stream`: création de streams
- `tls`: protocole TLS/SSL
- `tty`: accès aux streams en mode TTY
- `url`: parsing d'url
- **`util`**: tests de type, héritage, etc…
- `vm`: permet d'exécuter du JS dans une nouvelle VM
- `zlib`: compression gzip

En gras ceux qu'on manipulera effectivement durant la formation

ES6

- `import` / `export`

```
// export (module.js)

const life = 42

export default life

export const named1 = 1
export function named2 () {
  return 2
}


// import

import defaultExport from 'module'
import { named1, named2 } from 'module'
```

- En ES6 les symboles importés / exportés sont connus lexicalement
- En CommonJS les symboles sont connus au runtime

NPM

- un outil en ligne de commande
- inclus dans node
- un gigantesque dépôt de modules JS

```
node -v
npm -v
```

NPM : INSTALLATION

TROUVER UN MODULE

- `npm search`
- Plus efficace : Google "npm" + recherche…
- Plus précis : http://npms.io

INSTALLER UN MODULE

```
npm install $nom_du_module
```

UTILISER UN MODULE NPM

```
var mod = require("nom_du_module")
```

- Pas besoin de spécifier le chemin dans `require`
- Dossier `node_modules`
- Installer des modules simples : `async`, `bluebird`, `lodash`
- Installer des modules avec plus de sous-dépendances : `socket.io`, `express`

DEPENDENCY HELL

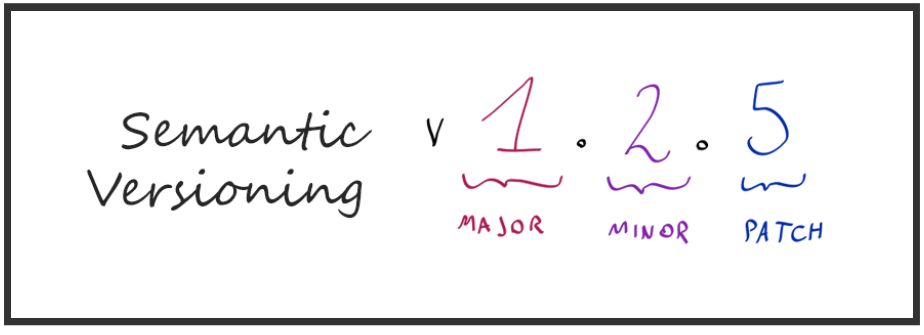
- Observer que `express` et `socket.io` dépendent tous les deux de `debug`
  - Un a fixé "2.1.0"
  - L'autre est plus souple : "~2.1.1"

Solution simple pour la plupart des packagers : 2.1.0

Mais si versions incompatibles ? Foutu !

Comme `require` = chemin vers un simple fichier .js, on peut avoir plusieurs copies d'un module sans conflit, npm installe pour chacun sa version. Gagné !

NPM : VERSIONS



Respecter le semantic versioning

```
npm version "major"|"minor"|"patch"
```

- Spéc: http://semver.org
- Checker: http://semver.npmjs.com

RÈGLES À APPLIQUER

Version "M.m.p"

- Je casse la compatibilité (suppression d'une fonction, ajout d'un paramètre obligatoire)?
  - M ++
- J'ajoute une fonctionnalité?
  - m ++
- Je corrige un bug sans effet sur le champ fonctionnel?
  - p ++

La correction d'un bug discutable peut représenter une rupture de compatibilité.

GÉRER SES DÉPENDANCES

```
npm install --save express
```

- Le `package.json` décrit les dépendances

```
{
  "dependencies": {
    "socket.io": "^1.3.2"
  }
}
```

- Pour installer toutes les dépendances d'un projet :

```
npm install
```

Supprimer une dépendance (penser au "--save")

```
npm remove --save express
```

DEVDEPENDENCIES

- Option `--save-dev`
- Installée seulement quand `npm install` depuis le projet
- Pas installées quand `npm install lemodule`

VERSIONS

- `M.m.p` = la version `M.m.p` précisément
- `>= ...`, `<= ...`, etc…
- **chapeau** `^M.m.p` = toutes les versions `M.x.y` où `x ≥ m || y ≥ p`
- **tilde** `~M.m.p` = toutes les versions `M.m.x` où `x ≥ p`
- Tout ça marche bien si tout le monde respect semver :
  - **RESPECTEZ SEMVER**

PUBLIER SES MODULES

- S'identifier sur le dépôt : `npm adduser`
- Publier le module : `npm publish`

Attention à ne pas publier par erreur !

```
{
  "private": true
}
```

`npm link` pour tester son module sans le publier

MODULES GLOBAUX

```
npm install -g monmodule
```

- Utile pour installer des outils CLI
  - `less`, `eslint`, `gulp`… `npm` !
- On **ne peut PAS** `require()` un module global
- Attention à sudo, plutôt choisir un prefix dans son ~/.npmrc

DÉCLARER UN OUTIL CLI

```
{
  "bin": {
    "nom-outil": "bin/script.js"
  }
}
```

NPM AVANCÉ

SCRIPTS

```
{
  "scripts": {
    "nom-script": "commande"
  }
}
```

```
npm run nom-script
```

ALLER PLUS LOIN...

- Gérer le package.json
- Gérer le cycle d'installation
- etc…

RTFM: NPM HELP [COMMANDE]

- Accéder à la home d'un module: `npm home $module`
  - **C'est la commande que j'utilise le plus souvent**
- Créer un package.json initial: `npm init`
  - Note: elle fonctionne aussi en mise à jour
  - Pour accelerer: `--force`
- Scripts standard (pas besoin de "run"):
  - start, restart, stop
  - test
- Scripts spéciaux (hooks):
  - prepublish, postpublish
  - preinstall, postinstall
  - preuninstall, postuninstall
  - prestart, poststart, prestop, poststop, prerestart, postrestart
  - pretest, posttest
- Gérer les mainteneurs: `npm owner`
- Figer l'arbre des versions: `npm shrinkwrap` -> automatique en npm 3.0
- Exporter une archive de son module: `npm pack`
- npm-run-all
- ~/.nmprc

BROWSERIFY

- utiliser les modules node coté client
- réimplemente `require` via un AST
- les modules du coeur sont dispos (presque)
- plugins de transformation (reactify, uglify)

WEBPACK

- peut inliner du CSS, HTML, des images etc...
- Hot Module Reloading (React components par exemple)
- tree shaking (en v2)

TESTS UNITAIRES

TESTER ?

- Ça sert à débusquer les bugs
- à documenter son API
- à pouvoir refactorer tranquille

C'EST LONG À ÉCRIRE

On appelle ça un investissement

C'EST LONG DE TESTER

C'est mieux quand c'est automatisé

QUELS TYPES DE TESTS ?

- **Tests unitaires**
  - On teste des modules séparément
  - On se concentre sur les API
- **Tests fonctionnels**
  - On teste **l'application** au complet
  - On vérifie les **fonctionnalités**
- **Tests d'intégration**
  - On teste tout l'écosystème
  - Souvent avec de vraies jeux de données

TEST RUNNER

Le "test runner" est l'outil qui va prendre nos fichiers de test et les exécuter, et indiquer le résultat des tests.

POURQUOI PASSER PAR UN OUTIL TIERS ?

- API spécifique injectée
- génération de rapports (Jenkins)

`MOCHA`

mocha est un choix populaire pour Node.

ASSERTIONS

- Un test échoue si son code plante
- Assertion = plante si une condition n'est pas vérifiée
  - lecture agréable
  - message d'erreur utilisable

ASSERT++

Chai offre une syntaxe sexy (expect ou should) :

```
expect(3.14)
  .to.be.a("number")
  .and.to.be.below(4)
  .and.to.be.above(3);
```

TP

- Installer `mocha` et `chai` en devDependencies

```
npm install --save-dev chai mocha
```

- Créer un premier test dans `test/sample.js`

```
define("Sample", function () {
  it("should add 1 and 1" , function () {
    expect(1+1).toBeA("number").toEqual(2);
  });

  define("Array", function () {
    var arr = ["x", "y", "z"];
    it("should contain \"x\"" , function () {
      expect(arr).toBeAn( "array").toContain("x");
    });
  });
});
```

Commande : `./node_modules/.bin/mocha`

- Mais c'est chiant, installer en global ?
  - Plutôt utiliser les scripts npm (chapitre suivant)
- Tester les reporters `--reporter` (liste avec `--reporters`)
  - Pour la console : `spec`, `list`, `dot`…
  - Pour les outils : `tap`, `xunit`…
  - Pour le fun : `landing`, `nyan`
- Modifier les tests pour introduire des erreurs et observer

NPM TEST

- `./node_modules/.bin/mocha`: long, erreurs possibles
- Installation globale : une étape de plus

Des solutions :

- `Makefile` : standard, mais Mac/Linux
- scripts npm : devient standard, multi-plateforme

```
{
  "scripts": {
    "test": "mocha -R spec -G --check-leaks test/"
  }
}
```

TP

- Écrire des tests pour notre application CLI d'addition
  - Tester les modules séparément
  - Tester l'API globale

```
define("sum CLI", function () {

  // Test fonctionnel
  it("should take arguments and sum them" );

  define("module: cli-args" , function () {
    it("should extract CLI arguments from process.argv"  );
  });

  define("module: sum", function () {
    it("should sum array of numbers" );
    it("should sum array of numeric strings"  );
  });

});
```

- Tester les modules internes ? Discutable :
  - API privée
  - mais `require`able quand-même (introduction de `require("module/sous-module")`)
- Observer que les `it()` sans fonctions apparaissent en "pending"

GESTION DU CODE ASYNCHRONE

CONCURRENCE VS. MULTI-TÂCHE VS. BLOQUANT

Multi-tâche

- Vraiment parallèle (sauf si pas de CPU disponible)
- Risque d'accès simultanés à la mémoire

Concurrence

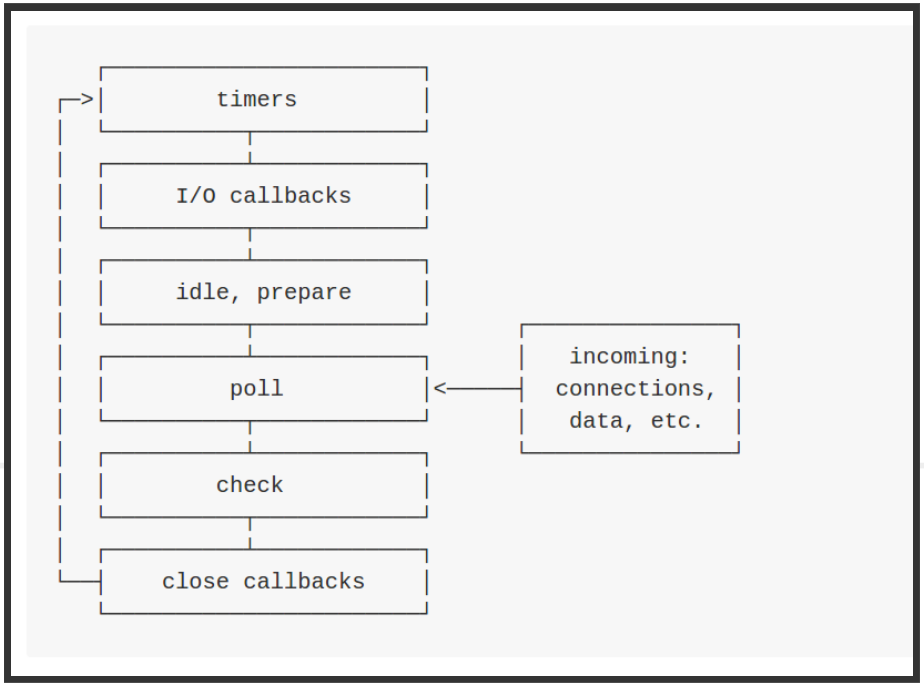
- Les traitements synchrones sont bloquants
- On traite le retour dès que le système est disponible

Bloquant

- On bloque tout pendant que la base de données bosse

NODE EVENT LOOP

https://github.com/nodejs/node/blob/master/doc/topics/the-event-loop-timers-and-nexttick.md

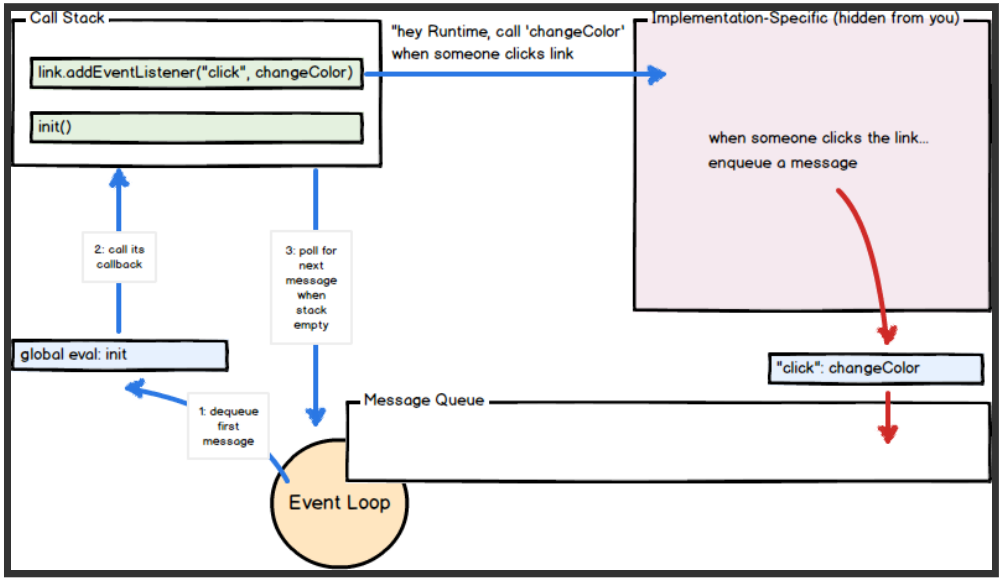


Visuel : http://latentflip.com/loupe/

En détail : https://youtu.be/8aGhZQkoFbQ

http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/

Exemple browser :



CALLBACKS

- On passe à la fonction asynchrone la fonction qui devra être exécutée quand le résultat sera prêt
- Format choisi par Node.js

Signature typique d'une fonction asynchrone par callback :

```
function foo (...args, callback) {
  // When an error occurs:
  callback(err)
  // When a result is available:
  callback(null, result)
}
```

- *error-first callbacks* (errback) : une erreur est la seule donnée qu'on est sûre de pouvoir recevoir
- Exception à l'error-first: fs.exists
- Ne pas confondre avec les *iteratee* qui sont synchrones.

TP

- Concaténer 3 fichiers texte ensemble
- Version synchrone à l'aide de fs.readFileSync
- Version asynchrone imbriquée à l'aide de fs.readFile

```
fs.readFile(…, (err, content1) => fs.readFile(…, …))
```

- Développer la version concurrente

```
var buffers = [];
var pending = 0;

function read (file, index) {
  pending++;
  fs.readFile(file, (err, content) => {
    // TODO handle error
    buffers[i] = content;
    pending--;
    if (pending === 0) {
      onEnd();
    }
  });
}

function onEnd () {
  process.stdout.write(Buffer.concat(buffers));
}

read("1.txt", 0);
read("2.txt", 1);
read("3.txt", 2);
```

- Buffer vs String : Buffer occupe la mémoire hors heap, ça évite le heap overflow (taille limitée par le runtime)

FLOW CONTROL

- Appels en série : "Pyramid of doom"

```
fs.readFile(file1, (…) => {
  fs.readFile(file2, (…) => {
    fs.readFile(file3, (…) => {
      // What?
    });
  });
});
```

- Appels en concurrence : complexe…

```
var pending = 42;
…
```

- Des solutions pour organiser son code : `async`, `contra`…

Callback Hell : http://callbackhell.com/

TP

L'unité de base de `async` est la fonction asynchrone minimale

```
function (callback) {
  callback(err, result);
}
```

Réécrire l'exercice précédent avec `async` :

```
// Introduction des "closures"  : pattern très fréquent
function read (file) {
  return cb => fs.readFile(file, cb)
}

async.parallel([
  read("1.txt"),
  read("2.txt"),
  read("3.txt")
], function (err, contents) {
  …
})
```

En série ? Remplace `async.parallel` par `async.series`.

NE JAMAIS ORCHESTRER SON CODE ASYNCHRONE SANS LIBRAIRIE TIERCE

ÉVÈNEMENTS

- Classe `events.EventEmitter`
- Exemples : socket (client http, serveur http, https, udp…), streams, process…
- `emit` = exécuter les listeners

Une API simple :

```
ee.on("eventName", callback); // === ee.addListener
ee.removeListener("eventName", callback);
ee.removeAllListeners("eventName");

ee.once("eventName", …); // your best friend against memory leak

ee.emit("eventName", arg1, arg2...);
```

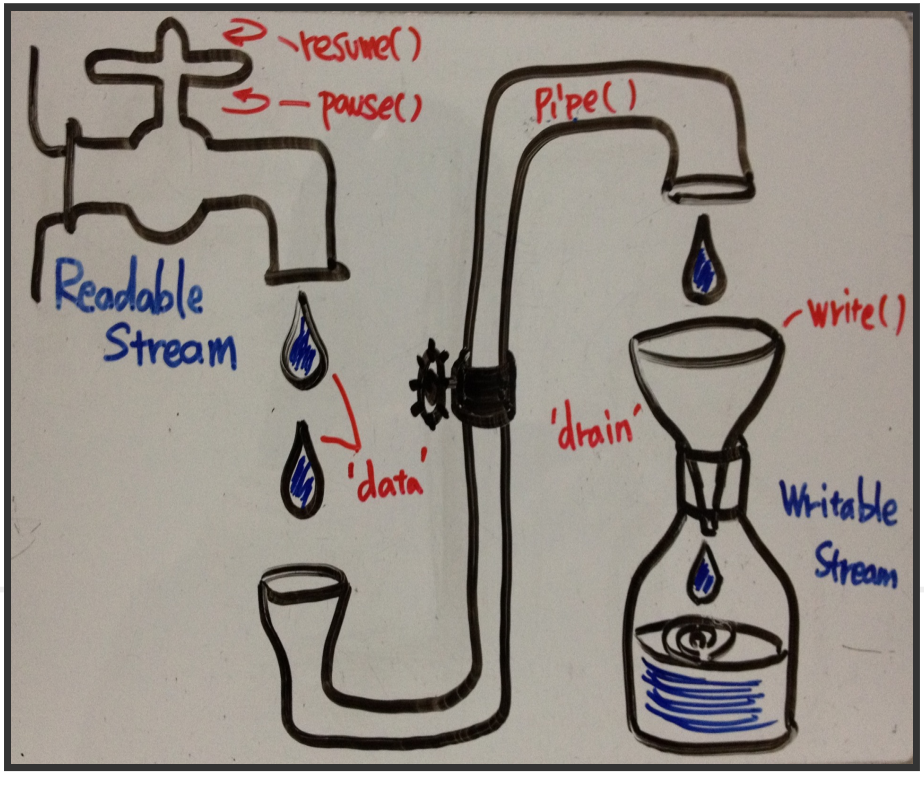
TP

Réécrire le précédent exercice avec des EventEmitter

```
var EventEmitter =  require("events").EventEmitter;

// Bus d'événements
var e = new EventEmitter();

var contents = [];
var pending =  0;

function read (file, index) {
  fs.readFile(file, (err, content) => {
    if (err) {
      e.emit("error", err);
    } else {
      e.emit("read", index, content);
    }
  });
}

var pending =  3;

e.on("read", function (i, content) {
  contents[i] = content;
});

e.on("read", function () {
  pending--;
  if (pending ===  0) {
    onEnd();
  }
});
```

ATTENTION À L'ÉVÉNEMENT SPÉCIAL "error"

STREAMS



fs.createReadStream, fs.createWriteStream, http.Request, http.Response…

Lecture en "push" (c'est la stream qui dicte son rythme) :

```
readable.on("data", function (chunk) { });

readable.on("end", function () { });
```

Lecture en "pull" (c'est le lecteur qui choisit le rythme) :

```
readable.on("readable", function () {
  var chunk = readable.read(…);  // null when end of buffer
});

readable.on("end", function () { });
```

Écriture :

```
writable.write(…);  // returns false if "drain" must be awaited

writable.on("drain", function () { /* ready to write again */  });

writable.close();
```

Pipe :

```
readable.pipe(writable);
// chain
readable.pipe(readableAndWritable).pipe(writable);
```

Tenter de réécrire le code précédent en mode streams :

- Compliqué d'orchestrer plusieurs streams, mauvais use case

PROMISE

Manipuler la représentation de la valeur future au lieu d'attendre sa disponibilité

- **Standard EcmaScript**
  - natif dans Node ≥ 0.12, utiliser `bluebird` sinon
- **État** : résolu reste résolu, cassé reste cassé
- **Chaînable**

```
var promiseOfBuffer = read("file");
// .then = when value really available
promiseOfBuffer.then(callback)  // callback(buffer)
// this returns the promise of callback(buffer)'s return value
.catch(onError)  // onError(err)
// this also returns a promise…
```

Aide à l'orchestration : `Promise.all(promises)`

- http://301.tl/moz-fr-promise
- https://www.promisejs.org/
- http://naholyr.fr/2014/03/promises/
- Création d'une promesse :

```
var promise = new Promise((resolve, reject) => {
  if (ok) {
    resolve(returnValue);
  } else {
    reject(err);
  }
});
```

- **Note : les `throw` dans les callbacks et dans l'executor sont interceptés**

TP

Réécrire l'exercice précédent avec les promesses

```
// callback to promise: pattern
function read (file) {
  return new Promise((resolve, reject) => {
    fs.readFile(file, (err, content) => err
      ? reject(err)
      : resolve(content));
  });
}
```

En concurrence :

```
Promise.all([
  read("1.txt"),
  read("2.txt"),
  read("3.txt")
]).then(contents => …);
```

En série :

```
var file1 = read( "1.txt");
var file2 = file1.then( function () {
  return read("2.txt");
});
var file3 = file2.then( function () {
  return read("3.txt");
});
```

TESTER SON CODE ASYNCHRONE

Une fonction asynchrone rend la main immédiatement !

```
it("should do some work", function () {
  doSomeWork(err => expect(err).toNotExist());
  // I'm here before the assertion
});
// And then I'm here, test is finished, assertion not checked
```

Mocha a pensé à tout :

- La fonction de test prend une fonction en paramètre ?
  - Mocha attend l'appel de ce *error-first* callback
- La fonction de test retourne une Promise ? Passer un callback à la fonction, ou retourner une Promise !
  - Mocha attend la **résolution de la promesse**

TP OU DÉMO

Mode callback

```
it("test async", function (cb) {
  doSomeWork(cb);  // cb(err) → test failed
});
```

Mode promise

```
it("test promis", function () {
  return doSomeWorkPromise();  // resolved = success, broken = fail
});
```

ES2017

- Des générateurs capables de s'interrompre et redémarrer
- Un orchestrateur intégré branché sur des promesses

`ASYNC` / `AWAIT`

```
async function incr (promise) {
  var value = await promise
  return value + 1
}

async function main () {
  console.log(await incr(getFromHTTP()))
}

main()
```

https://tc39.github.io/ecmascript-asyncawait

HTTP

MODULE HTTP

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

http://nodejs.org/api/http.html

`createServer`: création de serveur HTTP

```
http.createServer(function (incomingMessage, serverResponse) { });
```

`request`: effectuer une requête HTTP

```
const clientRequest = http.request("http://www.google.com");
```

UN SERVEUR HTTP SIMPLE

Chaque requête passe par le *handler*

```
function handler (req, res) {
  // req instanceof http.IncomingMessage
  // res instanceof http.ServerResponse
  res.setHeader("content-type", "text/plain");
  res.write("You visit " + req.url);
  res.end();
}
```

- La *request* représente les données envoyées par le client
  - `method`, `url`, `headers`, données POST (*readable stream*)
- La *response* représente la réponse qui sera envoyée
  - définition des headers
  - envoi du contenu (*writable stream*)

TP (NOUVEAU PROJET)

- Écrire le script serveur complet

```
"use strict";

var http = require("http");

function handler (req, res) {
  …
}

var server = http.createServer(handler);

server.listen(8080);

server.on("listening", function () {
  console.log(this.address());
});
```

- Observer le comportement en cas d'erreur (événement "error")
- Démo streams : modifier le handler pour écrire un serveur echo

```
function handler (req, res) {
  req.pipe(res);
}
```

ATTENTION AUX CALCULS BLOQUANTS !

- Requête client = empilée dans l'*event loop*
- *Bloquer* = aucune requête n'est traitée pendant ce temps
  - Elles continuent de s'empiler

Apprenez à **VRAIMENT** craindre le code bloquant

Tester le fameux Fibonnacci, en profiter pour réaliser un routing simple avec `http`

```
function fibo (n) {
  if (n === 0) return 0;
  if (n === 1) return 1;
  return fibo(n - 1) + fibo(n - 2);
}

function handler (req, res) {
  if (req.url.substring( 0, 6) === "/fibo/") {
    var n = Number(req.url.substring( 6));
    if (isNaN(n)) {
      res.statusCode = 400;
      res.write( "ERROR: valid number expected" );
    } else {
      var result = fibo(n);
      res.write( "<code>fibo(" + n + ") = <strong>" + result + "</strong></c
    }
  } else if (req.url === "/") {
    res.write( "Usage: /fibo/{number}" );
  } else {
    res.statusCode = 404;
    res.write( "Not found");
  }

  // No res.end = browser never renders
  // Duplicate res.end = server crashes
  res.end();
}
```

ALLER PLUS LOIN

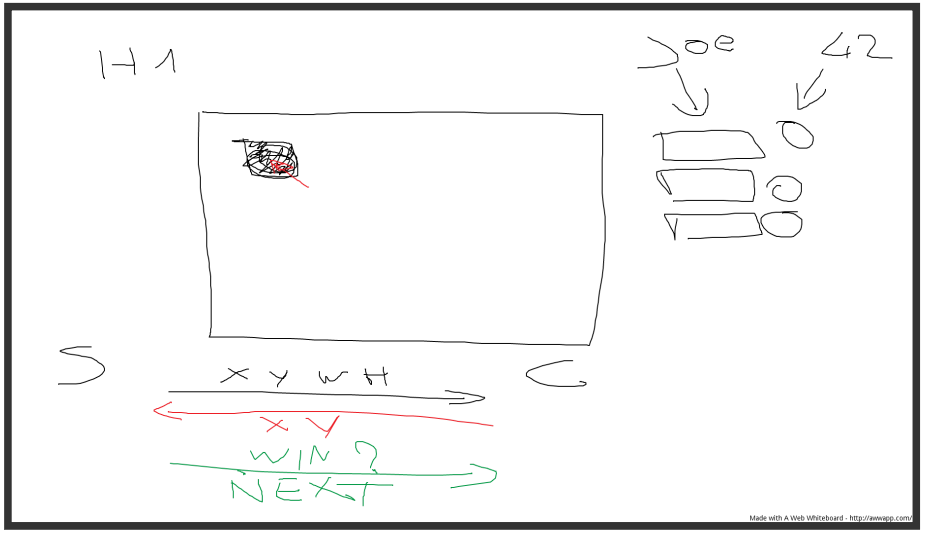
Le module `http` est minimaliste

There is a module for that

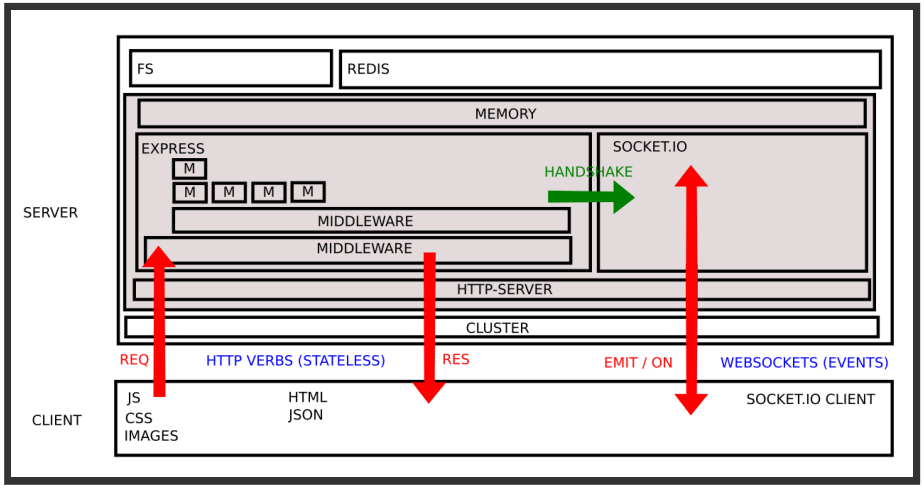
- Routage: `node-simple-router`, `connect`, `express`…
- Cookies: `cookie` (+ `res.setHeader`), `connect`, `express`…
- Formulaires: `formidable`, `busboy`, `connect`, `express`…

Simplifions-nous la vie

NOTRE JEU



ARCHI



EXPRESS

EXPRESS

Micro-framework pour les applications web Node :
http://expressjs.com/

- API élégante
- Extensible (*middlewares*)
- Configurable
- Rendu par templates

```
var app = express();

app.get("/", function (req, res) {
  // req: http://expressjs.com/4x/api.html#request
  // res: http://expressjs.com/4x/api.html#response
});

var server = http.createServer(app);
```

Installer express

```
npm install --save express
```

Écrire le serveur Hello world :

```
"use strict";

var express = require("express");
var http = require("http");

var app = express();

app.get("/", function (req, res) {
  res.send("Hello, world");
});

var server = http.createServer(app);

server.listen(8080);

server.on("listening", function () {
  var address = server.address();
  console.log("Server ready %s:%s", address.host, address.port);
});
```

CONFIGURATION

Supprimer l'entête automatique "X-Powered-By: Express"

```
app.set("x-powered-by", false);

app.get("x-powered-by") // two-face get
```

LE ROUTAGE

Route = méthode + url

```
app.méthode(url, handler)
```

FORMAT

```
// Simple
app.get("/hello/world", …)

// Parameter
app.get("/hello/:world", …) // req.params.world

// Complex URL using RegExp
app.get(/^hello\/(.*)$/, …) // req.params[1]

// Parameter constraint
app.get("/hello/:world([^\\s+])", …)
```

TP (NOUVEAU PROJET)

Application fil rouge : jeu en ligne

- `/` → rendu de la homepage = formulaire d'identification
- `/login` → traitement du formulaire d'identification, redirige vers
- `/welcome/:username` → rendu d'une page "bonjour machin"

Ne pas tester le flux complet avant code final

```
app.get("/", function (req, res) {
  res.render("login");
});

app.get("/welcome/:username" , function (req, res) {
  res.render("app", {
    "username" : req.params.username
  });
});

// On n'a pas encore les outils pour celle-là
app.post("/login", function (req, res) {
  var postData = …;  // ???
  res.redirect("/welcome/" + postData.username);
});
```

Lire les données POST ? **Middlewares**

CONFIGURATION MOTEUR DE TEMPLATES

https://github.com/tj/consolidate.js

```
app.engine("html", consolidate.swig);
app.set("views", path.join(__dirname,  "views"));
app.set("view engine", "html");
```

MIDDLEWARES

Un middleware s'intercale dans le traitement d'une route.

```
function log (req, res, next) {
  console.log(new Date, req.method, req.url);
  // Async: call next or chain is broken!
  next();
});
```

Ils peuvent être utilisés au niveau de l'application :

```
app.use(log);
…
```

Ou au niveau d'une route seulement :

```
app.get("/logged", log, function (req, res) { … });
app.get("/unlogged", function (req, res) { … });
```

TP

- Insérer le middleware "log" d'exemple dans l'application
- Écrire un middleware de sécurité qui vérifie le paramètre ":username"

```
function checkUsername (req, res, next ) {
  if (req.params.username …) {
    next();
  } else {
    // 500
    next( new Error ("…"));
    // Ou redirect
    res.redirect( "/");
  }
}
```

- Gestion d'erreur par middleware

```
app.use( function (err, req, res, next ) {
  res.render( "error", {
    "err": err.message
  });
});
```

MIDDLEWARES STANDARD

Express ne fait rien à part le routing, les middlewares vont assurer les fonctions auxiliaires:

- Formulaire: **`body-parser`** (pas d'upload), **`multer`**
- Logging: **`morgan`** (ce nom…)
- Serveur statique: **`serve-static`**, `serve-index`
- Cookies: **`cookie-parser`** (`req.cookies`, `req.signedCookies`, `res.cookie()`)
- Sessions: **`express-session`**
- Compression GZip: `compression`
- Sécurité CSRF: `csurf`
- etc… https://github.com/senchalabs/connect#middleware

TP

- Activer le logging
  - Note: `morgan` ne log que les requêtes terminées, oublier d'appeler `next()` dans un middleware et observer ce qui se produit
- Activer le serveur static pour les assets (css, js, images)
- Activer la session

```
app.use(morgan());
app.use(serveStatic(path.join(__dirname,   "public")));
app.use(cookieParser( "secret"));
app.use(expressSession( /* takes cookie-parser secret */ ));
app.use(bodyParser());
```

- Warnings "deprecated"

```
var secret =  "secret";
app.use(cookieParser(secret));
app.use(expressSession({ "secret": secret,  "resave": false,  "saveUninitializ
app.use(bodyParser.urlencoded({  "extended": true}));
```

- Utiliser la session pour l'identification

```
app.get("/", function  (req, res) {
  if (req.session.username) {
    // already logged in
    return res.redirect( "/welcome");
  }
  res.render( "login");
});

app.get("/welcome", function  (req, res) {
  res.render( "app", {
    "username": req.session.username
  });
});

app.post("/login", function  (req, res) {
  req.session.username = req.body.username;
  res.redirect( "/welcome");
});
```

- Réordonner les middlewares pour observer le comportement
- Authentification: voir module `passport`

ARCHITECTURE CLASSIQUE

Arborescence d'une application Express

```
.
├── app.js
├── config/
├── data/
├── lib/
│   ├── routes/
│   └── session-store.js
├── package.json
├── public/
├── server.js
└── views/
    └── layout.html
```

```
{
  "main": "app.js",
  "scripts": {
    "start": "node server.js",
    "watch": "supervisor -i public server.js"
  }
  …
}
```

Fichiers statiques en production

```
location / {
    root /path/to/app/public;
    try_files $uri $uri/ @node;
}

location @node {
    proxy_pass http://127.0.0.1:8080 ;
}
```

config/ et data/

- Stockage des fichiers de configuration (voir plus loin)
- Stockage de données fichiers, fixtures de test, etc…

Externaliser les routes

- Un module = une catégorie de routes

```
// app.js
var auth = require("./lib/routes/auth" );

app.get("/login", auth.loginForm);
app.post("/login", auth.login);
app.get("/logout", auth.logout);

app.get("/public", …);
app.get("/private", auth.isAuthenticated, …);
```

```
// lib/routes/auth.js
exports.loginForm = function (req, res) …
exports.login = function (req, res) …
exports.logout = function (req, res) …
exports.isAuthenticated = function (req, res, next ) …
```

Externaliser le session-store

- API: `get(sid, cb)`, `set(sid, data, cb)`, `destroy(sid, cb)`

```
// app.js
app.use(expressSession({
  …
  "store": require("./lib/session-store" )
}));
```

```
// lib/session-store.js
var FileStore = require("session-file-store" )(expressSession);

module.exports = new FileStore({
  "secret": "…",
  "path": path.join(__dirname, "..", "data", "sessions")
});
```

ENCORE + LOIN

SAILS.JS

- Scaffolder inspiré de RoR (yeoman like)
- Blueprints pour faciliter le CRUD
- ORM avec Waterline (mongoose like)
- Routes HTTP et/ou WebSocket avec JWR

BASES DE DONNÉES

NODE ET LES BASES DE DONNÉES

> *There's a module for that*

- `sqlite3`
- `mysql`
- `oracle`
- `pg` (+ `pg-native`)
- `mongodb`
- `cradle` (CouchDB)
- `redis` (+ `hiredis`)
- …

MONGODB

Base orientée documents.

- **La star des bases NoSQL**
- Le meilleur choix ?
  - Syntaxe **JavaScript**
  - Modélisation polyvalente (documents, *schemaless*) (*)
  - Scalable facilement
  - **Performances : attention !** (**)

(*) Attention : ne pas réfléchir relationnellement en NoSQL

(**) Utiliser l'aggrégation sans cluster = suicide

REDIS

Base clé/valeur **extrèmement rapide**

- Toutes les données chargées en mémoire
- Données structurées et opérations complexes
- Transactions
- Scripts
- Cluster

CONTREPARTIES

- Consommation RAM

TP

- Stocker les sessions dans Redis

NOTRE JEU

- Si partie en cours, récupérer les données
- Sinon générer une nouvelle partie

Une partie = coordonnées aléatoire d'un objet sur la page (stocké dans Redis)

Le premier qui clique dessus a "gagné" → stocker les scores + rafraichir la page

STRUCTURE DE DONNÉES

- game = string "x,y,w,h"
- scores = sorted set des usernames sur score

REDIS : TOUJOURS PLUS !

La boite à outils de la scalabilité !

- Synchroniser des opérations à travers un cluster :
  PUB/SUB (`PUBLISH`, `SUBSCRIBE`)
- Pool de workers pour déléguer les calculs :
  PUSH/PULL (`RPUSH`, `BLPOP`)

TESTS ET BASES DE DONNÉES

Penser à préparer ses données de test : `before`, `beforeEach`

```
define("my feature", function () {
  before(function (cb) {
    loadFixtures(cb);
  });
  // shorter: before(loadFixtures)
});
```

Faire le ménage en partant

```
after(function (cb) {
  cleanData(cb);
});
// shorter: after(cleanData)
```

Mais on casse mes données quand on lance les tests ??

GÉRER SA CONFIGURATION

ENVIRONNEMENTS

Le standard *de facto* : variable d'environnement `NODE_ENV`.

Valeurs usuelles : `development` ou `production`.

```
NODE_ENV="development" node "server.js"
```

Dans le script Node :

```
process.env.NODE_ENV;

// Express
app.get("env")
```

- Express : comportements spécifiques si `production`
- **Idée** : configuration fonction de l'environnement

MODULES JSON

`require` comprend le JSON automatiquement

```
var config = require("./config/config.json");
```

PLUS FORT

La configuration dépend de l'environnement

```
require("./config/config-" + process.env.NODE_ENV + ".json");
```

ENCORE PLUS FORT

un fichier de *defaults* + un fichier par environnement

```
// LoDash
_.defaults(config, require("./config/defaults.json"));
```

ALLER PLUS LOIN

Une configuration aux petits oignons avec `convict` (Mozilla)

- Schéma
- Sources : CLI, environnement, fichiers…

```
// config/index.js
var conf = module.exports = convict({
  env: {
    doc: "Applicaton environment.",
    format: ["production", "development", "test"],
    default: "development",
    env: "NODE_ENV", // override from environment NODE_ENV=…
    arg: "env" // override from CLI --env=…
  },
  …
});

// .json can have comments!
conf.loadFile([
  "config/config-defaults.json",
  "config/config-" + conf.get("env") + ".json"
]);

conf.validate();
```

TESTS ET CONFIGURATION

- Accès à la base de données en configuration

```
// defaults.json
{
  "db": "…" // default database
}

// config-test.json
{
  "db": "…" // test database
}
```

LANCER LES TESTS AVEC LE BON ENVIRONNEMENT

- Manuellement : risque d'oubli
- Dans les fichiers test : pénible
- `mocha.opts` + `node-env-test`

MANUELLEMENT

```
{
  "scripts": {
    "test": "NODE_ENV=test …"
  }
}
```

DANS LES FICHIERS TEST

```
process.env.NODE_ENV =  "test";

…
```

DANS LES OPTIONS MOCHA

```
npm install --save-dev env-test
```

```
// mocha.opts
--require=env-test
```

WEBSOCKETS AVEC SOCKET-IO

LES WEBSOCKETS

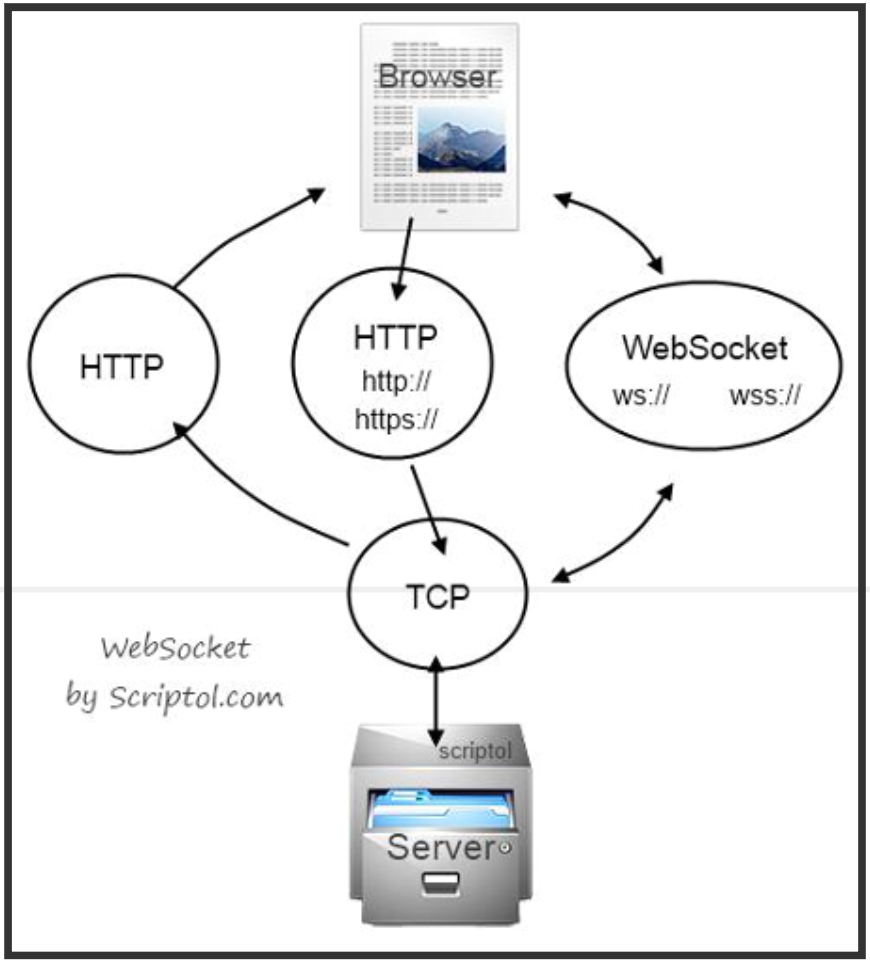
- "Temps réel"
- Full-duplex (requêtes & réponses se croisent)
- Protocole basé sur HTTP

```
Connection: Upgrade
Upgrade: websocket
```

Excellent support navigateur

http://caniuse.com/websocket

LES WEBSOCKETS



WEBSOCKETS : L'API DOM

http://301.tl/mozfr-websockets

```
var socket = new WebSocket("ws://url");

socket.send("message");

socket.onmessage = function (event) {
  console.log(event.data); // string
}
```

- pas de distinction des messages (simples strings)
- événement "open", "message", "close" et basta
- un peu limité…

SOCKET.IO À LA RESCOUSSE !

API de messaging basée sur `EventEmitter`

```
io.on("connection", function (socket) {

  socket.emit("hello", …);

  socket.on("hello-response", function (…) { … });

});
```

Multiples transports si *WebSocket* ne passe pas :

- Ajax Long Polling
- WebSocket
- On garde le long polling si WS ne marche pas

TP

Intégrer socket.io dans l'application, d'abord tout dans `server.js` puis externaliser dans un module à part, singleton assumé par simplicité :

```
// lib/websocket.js
var socketio = require("socket.io");

exports.io = null;

exports.init = function (server) {
  if (exports.io !== null) {
    throw …
  }

  return exports.io = socketio(server);
}
```

DU TEMPS RÉEL DANS SON APPLICATION

RÉDUIRE LE LAG

- Requête HTTP = headers + body
- Message WebSocket = seulement le message

NOTIFICATIONS

- Recevoir des événements du serveur

```
// Broadcast
io.emit("flash-news");
```

TP

Remplacer les intéraction HTTP par du websocket

```
// server
socket.on( "click", function (x, y) {
  // check if good position
  io.emit( "winner", "username???" );
  // next game soon
  setTimeout( … io.emit( "game", x, y, w, h) … );
});

// client
socket.on( "winner", …)
socket.on( "game", …)
```

Où récupérer le username ?

SOCKET.IO : ROOMS & NAMESPACES

REGROUPER LES SOCKETS AVEC LES ROOMS

```
socket.join("private");

io.to("private").emit("event");
```

SÉPARER LES SERVICES AVEC LES NAMESPACES

```
// server
io.of("/chat").on("connection", …);
io.of("/news").on("connection", …);

// client
var chatSocket = io.connect("…/chat");
var newsSocket = io.connect("…/news");
```

ROOM + ÉCOUTE

```
io.to("private").on("event", …);
```

ATTENTION les appels à `to` s'empilent ! (c'est `emit` qui flush)

```
io.to("room1").on("event1", …);

// broadcast to room1 + room2!
io.to("room2").emit("event2");
```

Solution :

```
io.rooms = [];
```

PHASE D'AUTHENTIFICATION

Filtrer les connexions avec les *middlewares*

```
io.use(function (socket, next) {
  next(error) // refuse connection
})
```

À l'origine une requête HTTP, qui reste disponible :

```
socket.request
socket.request.headers
// etc…

// addition:
socket.request.res // http://git.io/Fz1y
```

COUPLAGE AVEC EXPRESS

HTTP ≠ WEBSOCKET

Dans un handler Express on n'a aucun lien avec le socket

ÉTABLIR LE LIEN : LA SESSION

Handshaking → HTTP → headers → cookies → session

```
var sessionMiddleware = expressSession(…)
app.use(sessionMiddleware)

// …

io.use(function (socket, next) {
  sessionMiddleware(socket.request, socket.request.res, next);
  // socket.request.session = user's session
})
```

QUALITÉ

LA QUALITÉ

- fonctionnalité, expérience utilisateur…
- fiabilité (**tests unitaires**, **debugging**, **intégration continue**)
- performances (**profiling**)
- maintenabilité (**formatage**)
- portabilité assurée par Node

DEBUGGING

Passer son application en mode "debug" :

```
# Existing instance
kill -SIGUSR1 $PID

# Run in debug mode
node --debug-brk server.js
```

LA SOLUTION NODE-INSPECTOR

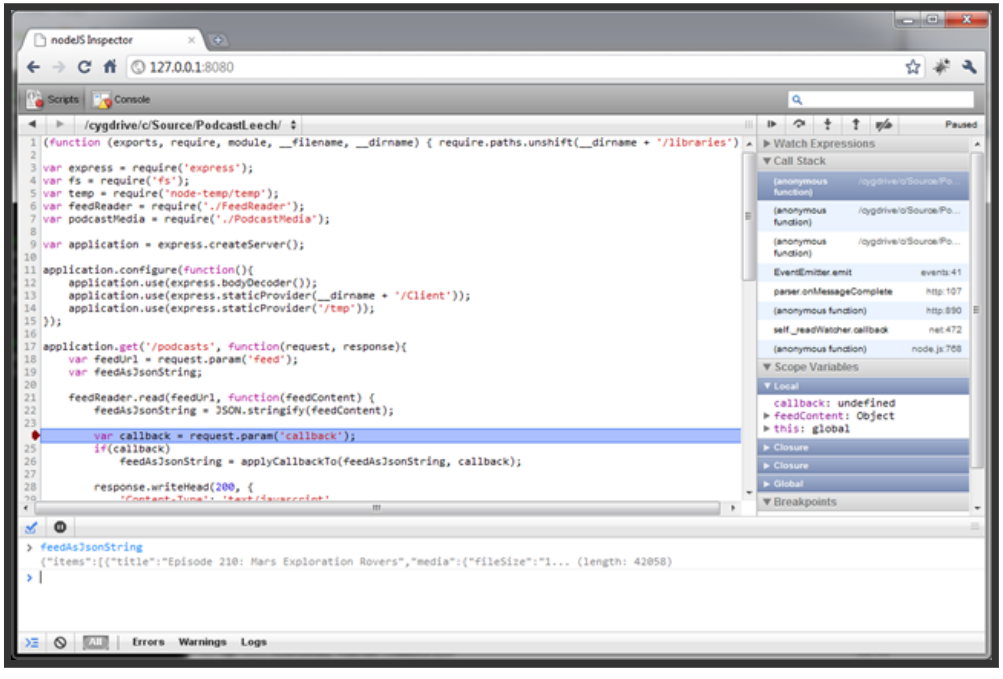
Debugger le JS serveur avec la console navigateur

```
node-inspector

google-chrome "http://127.0.0.1:8080/debug?port=5858"
```

- Éditer le code à chaud
- Inspecter les variables dans la console

https://www.youtube.com/watch?v=03qGA-GJXjI

Alternative: `node debug script.js`



DEBUGGING 2016

Pré-requis:

- Node v6.3
- Chrome canary v55 (avec les bons flags)

```
node --inspect server.js
```

https://github.com/nodejs/node/pull/6792
https://blog.hospodarets.com/nodejs-debugging-in-chrome-devtools

PROFILING

Onglet *Profiles* de `node-inspector`

- Profiling CPU
- Heap Snapshot
- Allocations over time

PROFILING

`v8-profiler`

```
var profiler = require('v8-profiler');
profiler.startProfiling('all');
…
profiler.stopProfiling('all');
```

`heapdump` (puis charger dans l'onglet "Profiles" de Chrome)

```
var heapdump = require('heapdump')
…
heapdump.writeSnapshot() // kill -SIGUSR2 $PID
```

`node-webkit-agent`

```
var Agent = require('webkit-devtools-agent')
var agent = new Agent;
agent.start(9999, 'localhost', 3333, true)
// http://c4milo.github.io/node-webkit-agent/26.0.1410.65/inspector.html?hos
```

FORMATAGE

JavaScript Linter (valident aussi un peu le style)

- `jslint` : par Crockford, un peu… rigide
- `jshint` : JSLint en plus souple
- `eslint` (mon préféré) : extensible, plus maintenable (AST, Espree)

JavaScript Code Style

- `jscs` : énormément de règles, une collection de presets pertinents
- `editorconfig` : halte à la guerre tabs vs spaces

INTÉGRATION CONTINUE

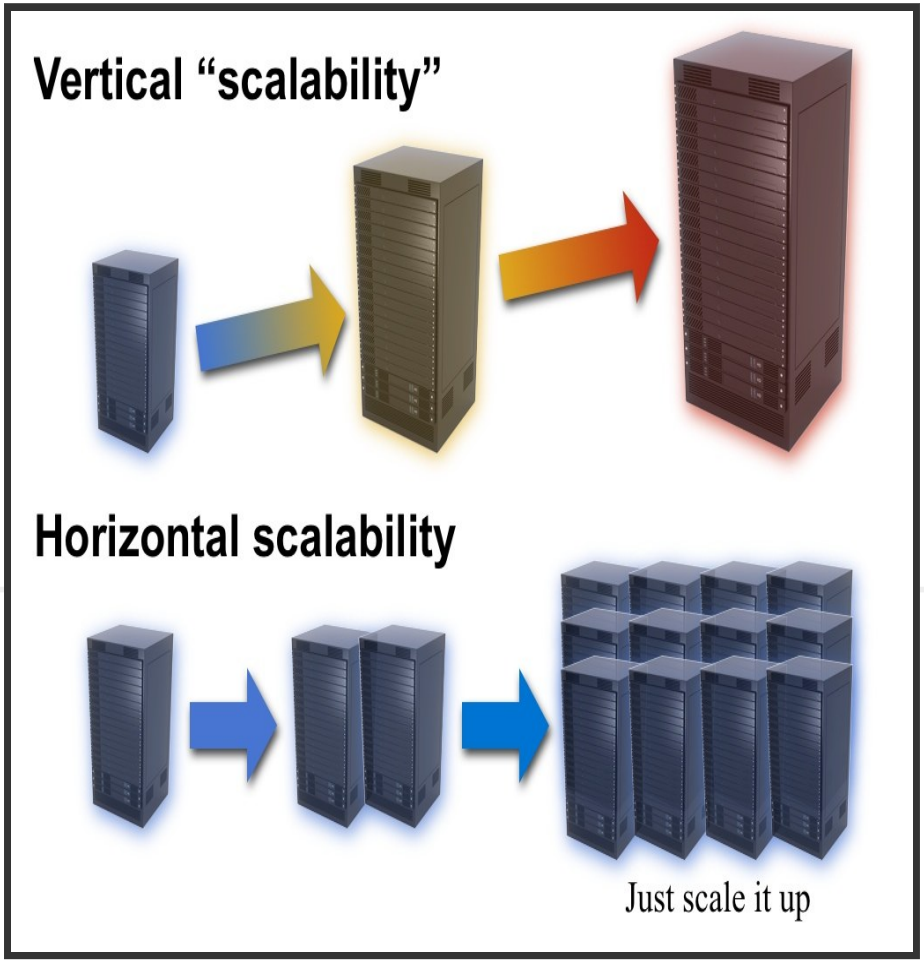
En mode relou

```
# Unit tests
npm test

# Linting
# .eslintrc: http://eslint.org/docs/configuring/
# .eslintignore: one pattern per line
eslint .

# Check style
# .jscsrc: http://jscs.info/overview.html#options
jscs .
```

Trouvez vos propres règles !

PERFORMANCES

SCALABILITÉ



VERTICAL

- Augmenter la RAM
- Augmenter la puissance CPU :
  - Horloge : limité
  - Nombre de cœurs : mono-threadé : (

HORIZONTAL

- Multiplier les instances
- Nécessite une application "scalable"

CLUSTER

Profiter des cœurs CPU disponibles :

```
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', function(worker, code, signal) {
    console.log('worker ' + worker.process.pid + ' died');
  });

  return; // nothing more in master
}

// child: continue normal execution
```

Principe de scalabilité horizontale appliqué à une machine

ÇA SCALE PAS !

Mémoire non partagée : **attention aux états globaux**

> *Mais je n'ai pas de variables globales !*

Ah oui ?

- Sessions ?
- Liste des sockets connectés ?
- D'autres états cachés ?

- Doc: https://github.com/elad/node-cluster-socket.io
- Implementation: https://github.com/bevacqua/lipstick

SESSIONS ? REDIS.

Synchroniser les données : **base de données**

REDIS EN TANT QUE BASE DE DONNÉES ULTRA-RAPIDE

```
var session = require("express-session");
var RedisStore = require("connect-redis")(session);

app.use(session({
    store: new RedisStore(options),
    secret: "…"
}));
```

SOCKET.IO ? REDIS !

Synchroniser les états : **PUB/SUB**

REDIS EN TANT QUE SERVEUR PUB/SUB LÉGER

```
SUBSCRIBE "channel" # client 1
SUBSCRIBE "channel" # client 2

PUBLISH "channel" "hello"
```

Adaptateur dédié à socket.io (stockage + PUB/SUB)

```
var ioRedis = require('socket.io-redis');

io.adapter(ioRedis({
  "host": "localhost",
  "port": 6379
}));
```

ANYTHING ? REDIS \0/

On l'a déjà vu : c'est la boite à outils de la scalabilité.

- **Synchroniser des états** : `PUBLISH`/`SUBSCRIBE`
- **Déléguer des calculs** : `RPUSH`/`BLPOP`
- **Données volatiles** : `EXPIRES`
- **Données souvent accédées** : Ultra-rapide
- Beaucoup de données : Nope.

OPTIMISATIONS V8

`JIT`: compilé, puis exécuté dans une VM.

- Chaque fonction est compilée indépendamment, dans un code machine soit "*générique*", soit "*optimisé*"
- On cherche au maximum à passer en mode "*optimisé*", mais tout n'est pas optimisable
  - **Beaucoup de fonctions**
  - Des **fonctions très simples**
- "*Hidden classes*" :
  - Ajouter une propriété à un objet existant = ***baaaaad***

RESSOURCES UTILES

- I-want-to-optimize-my-JS-application-on-V8 checklist
- Optimization killers
- Daniel Clifford on Optimizing JavaScript for the V8 Engine

GESTION DE LA MÉMOIRE

- Profiler la mémoire
- Limiter la *heap* (`Buffer`)
- Éviter les *memory leak* : **`EventEmitter`**

```
function (emitter) {
  // on/addListener: handler référencé par "emitter"
  emitter.on("event", function () {
    // closure: "emitter" référencé par handler
    console.log("event received");
  });
}
```

http://www.ibm.com/developerworks/library/wa-memleak/

CONCLUSION

ES6

Plein de fonctionnalités sympa :

- Destructuring
- Rest parameters
- Fat arrow
- Scoped variables
- Methods
- etc…

Les solutions ? `io.js`, `babel`…

```
// Destructuring

var [x, y, z] = [ 1, 2, 3];

var {name, phone} = person;

function showPersonInfo  ({name, phone} ) { }
showPersonInfo(person);

// Rest parameters

function add (number, numbers... ) {}
add(1, 2, 3)

// Arrow function

array.map(x => x +  1);

// Scoped variables

for (let i = 0 … ) { }
i; // ReferenceError

// Methods

var obj = {
  foo() { }
}

// Classes
// Modules
// etc…
```

ES6 NOW!

`babel`

```
# Exécuter du code ES6
babel-node server.js
```

Publier un module ES6

```
{
  // Transpiler en ES5 avant publication sur npm
  "scripts": {
    "prepublish": "babel --source-maps --out-dir lib src"
  }
}
```

Tests unitaires:

```
# test/mocha.opts
--compilers js:babel/register
```

Template ES6: https://github.com/lmtm/khaos-node-es6/

BONNES PRATIQUES

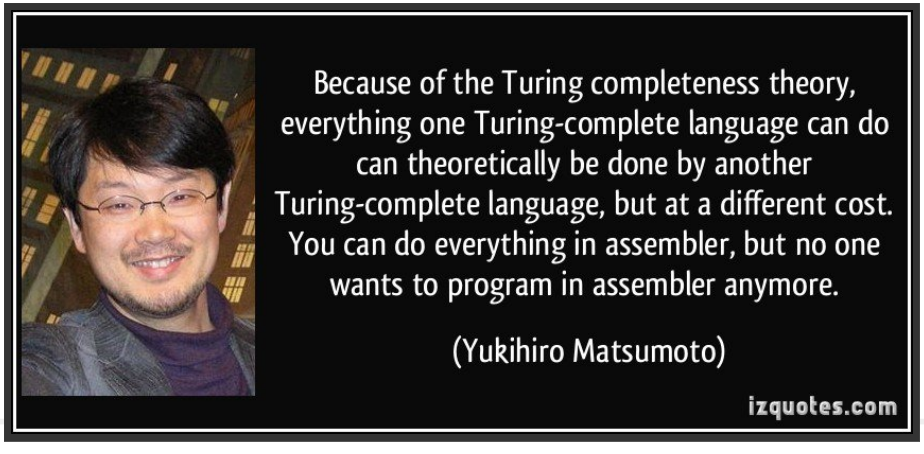
- Éviter la **Pyramid of Doom**
  - flow control, Promise…
- Écrire du code **scalable**
  - Pas de variables globales
  - Redis
- Surveiller les **performances**
- Bien packager son application
  - `{"private": true}`
- **Ne pas réinventer la roue !**

QUAND CHOISIR NODE ?

Node peut tout faire

QUAND NE PAS CHOISIR NODE ?

- On n'est pas à l'aise avec JavaScript
- On a des calculs "lourds" qu'on ne peut pas déporter



- http://nodejs.org/api/
- `npm home $nom_du_module` (voir slides précédents)
- https://www.npmjs.com/
- https://nodejsmodules.org/
- http://node-modules.com/
- https://github.com/iojs/io.js
- https://github.com/creationix/nvm
- http://redis.io/commands
- http://msgpack.org/
- http://www.echojs.com/
- http://dailyjs.com/
- http://www.echojs.com/
- https://github.com/sindresorhus/awesome-nodejs
- http://naholyr.fr/slides/js-pre-angular/
- http://naholyr.fr/slides/node-2015/

À BIENTÔT !

@naholyr | @Delapouite | @t8g