

301 – Introduction

node.js



WIK-NJS301

Durée estimée : 6h

Intervenant : Jeremy Trufier <jeremy@wikodit.fr>



WIK-NJS

Programme nodeJS

- 301 – Introduction**
- 302 – Scripting et CLI**
- 303 – Express.js**
- 304 – MVC Frameworks**
- 305 – Tests unitaires**

1XX – 1er année (pas de notion d'algorithmie)
2XX – 2e année (notions d'algorithmie succinctes)
3XX – 3e année (rappels et pratique, niveau moyen d'algorithmie)
4XX – 4e année (concepts avancés, niveau avancé d'algorithmie)
5XX – 5e année (approfondissement experts)

Au préalable

Qui l'utilise ?

ebay

NETFLIX

yammer

UBER

The New York Times



Linked in



Préparation

Tout d'abord, créez un dossier pour les cours nodejs

- Ouvrir un terminal ou PowerShell
- Naviguez vers ce dossier (avec la commande ``cd dossier1/sousdossier/...``)
- Créez un dossier appelé njs-301 (avec la commande ``mkdir njs-301``)
- Naviguez vers ce dossier (``cd njs-301``)

Tout ce qui se passera dans ce cours
devra se passer dans ce dossier "njs-301" dans le terminal

Avec Docker

Téléchargez et installez Docker :

<https://www.docker.com/community-edition#/download>

- Ouvrir un terminal ou PowerShell

```
$ docker run --rm -ti -v $(pwd):/srv -w /srv node:alpine node  
> console.log('hello')  
hello  
undefined
```

*sous windows remplacez : **\$(pwd)** par **%cd%***

--rm Supprime le container Docker après utilisation

-ti Attach to TTY and keep STDIN open

-v folderPath:/srv Monte le repertoire **folderPath** de l'hôte à l'emplacement **/srv** du container

-w /srv Défini **/srv** en tant que repertoire de travail dans le container

node:alpine Image Docker à utiliser (alpine est un Linux très léger)

node La commande à lancer dans le container (**sh** fonctionne aussi pour un accès au container)

Téléchargement

Ignorer ce slide si utilisation de Docker

- <https://nodejs.org>
- Version "Current"



Dans Powershell ou le Terminal

```
$ node  
> console.log('hello')  
hello  
undefined
```

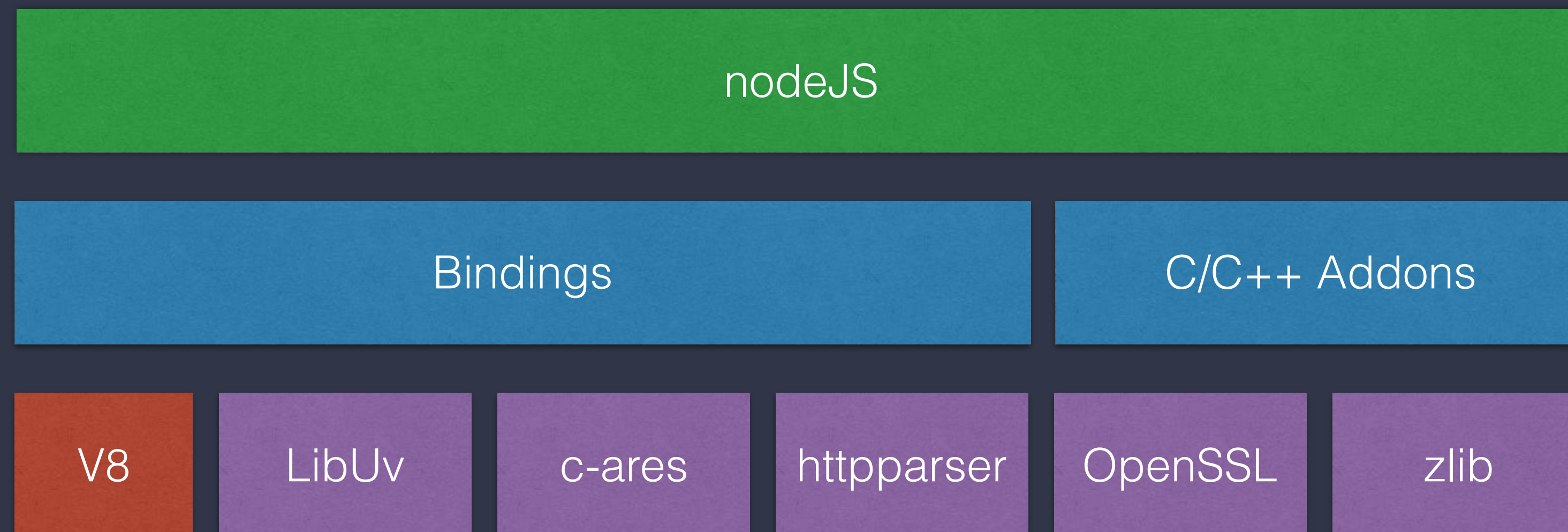
Installation Windows

Utilisateurs Docker, OSX & Linux : Ignorez ce slide :)

Utilisateurs Windows :

- Ouvrir PowerShell avec clic droit + ouvrir en tant qu'administrateur
 - exécuter la commande suivante : `npm install -g windows-build-tools`
- Fermer PowerShell, et réouvrir en utilisateur standard
 - exécuter : `npm install sqlite3`
- Si ça ne fonctionne pas et qu'une erreur rouge concernant "CL.exe" apparaît
 - Si vous avez Visual Studio, l'ouvrir, créer un projet Visual C++ pour télécharger les outils Visual C++ 2015
 - Réessayez
 - Sinon, ouvrir "Invite de commande développeur" et reessayez
- Sinon... essayer ce qui est en note de ce slide
- Sinon... utilisez une VM

La mécanique



nodeJS s'appuie un maximum sur des fonctions systèmes écrites en C/C++

Les avantages

- Asynchrone
- Événementiel
- Javascript
- Communauté
- Gros volumes de connections



CLI

Command Line Interface

Commande `node`

Terminal ou PowerShell

Evaluation de script

Mode interactif

Code javascript

Output

Return

Pour quitter

Execution d'un fichier JS

```
$ node -e "console.log('hello')"
```

```
$ node
```

```
> console.log('hello')
```

```
hello
```

```
undefined
```

```
> 1+3
```

```
4
```

```
> .exit
```

```
$ node ./hello.js
```

```
Hello World
```

```
$
```

Un serveur Web

```
const http = require('http')  
  
http.createServer((req, res) => {  
  res.end('Bonjour à tous !')  
}).listen(8080)
```

JavaScript ES6, ES7, ES8

Notions & Rappels

Les bases

constante `const hello = 'Bonjour'`

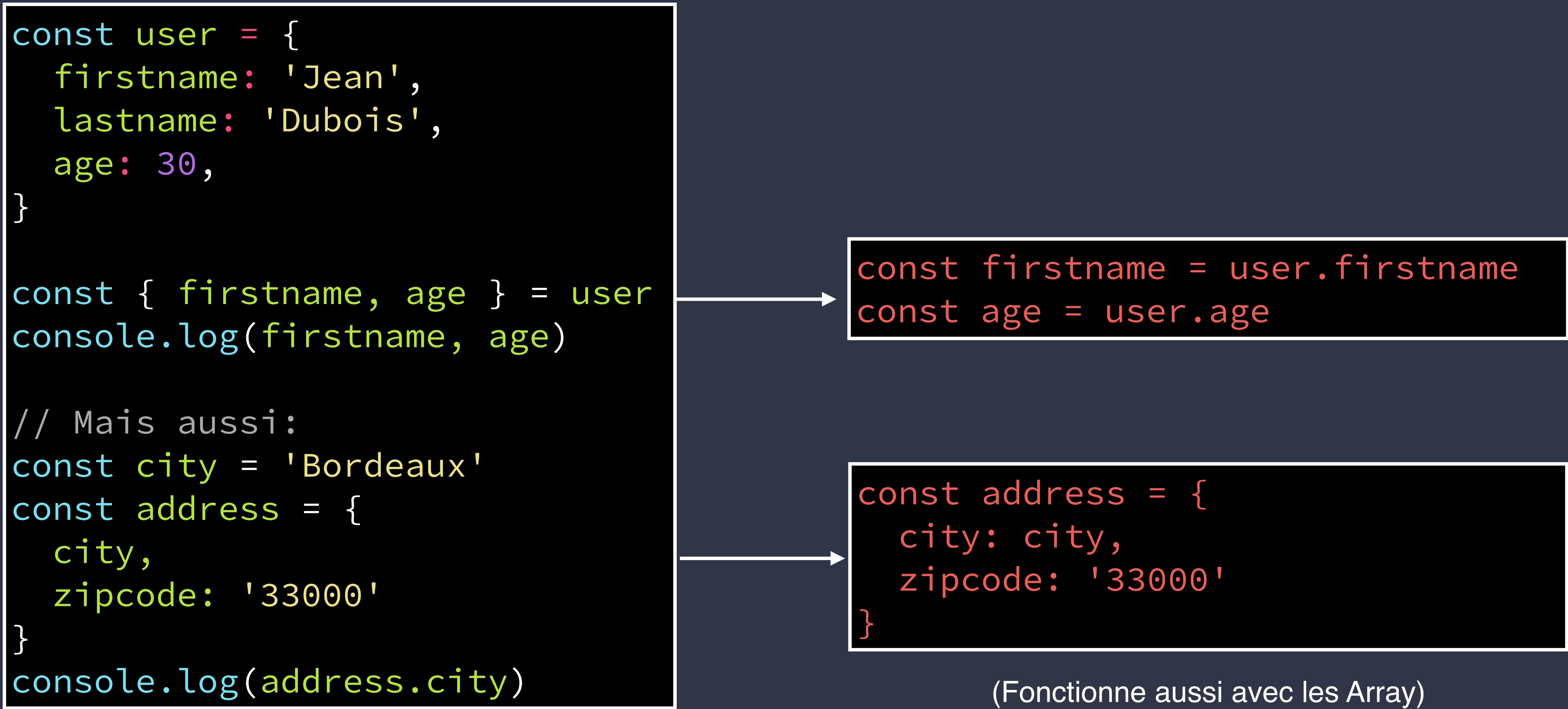
variable et objet `let currentUser = {
 firstname: 'Jean',
 lastname: 'Dubois',
}`

fonction nommée `function welcome (person) {
 return `${hello} ${person.firstname} ${person.lastname}`
}`

`console.log(welcome(currentUser))`

Astuces

```
const user = {  
  firstname: 'Jean',  
  lastname: 'Dubois',  
  age: 30,  
}  
  
const { firstname, age } = user  
console.log(firstname, age)  
  
// Mais aussi:  
const city = 'Bordeaux'  
const address = {  
  city,  
  zipcode: '33000'  
}  
console.log(address.city)
```



```
const firstname = user.firstname  
const age = user.age
```

```
const address = {  
  city: city,  
  zipcode: '33000'  
}
```

(Fonctionne aussi avec les Array)

Les types

- boolean
- number
- string
- undefined
- null
- object
- symbol (ES6)

Que remarquez-vous ?

```
typeof 'une chaine'  
typeof true  
typeof undefined  
typeof { name: 'Jean' }  
typeof [0, 1, 2, 3]  
typeof null  
typeof function() {}
```

Les conditions

Faible égalité

```
let a = '2'

if (a == 2) {
  console.log('a est égal à 2')
} else {
  console.log('a n\'est pas égal à 2')
}
```

Strict égalité : le type est pris en compte

```
let a = '2'

if (a === 2) {
  console.log('a est le chiffre 2')
} else {
  console.log('a n\'est pas le chiffre 2')
}
```

Condition ternaire

```
let a = '2'
console.log(a == '2' ? 'a égal 2' : 'a n\'est pas égal à 2')
```

Les conditions

Opérateurs de comparaison

<	Inférieur à
>	Supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
=	Égalité faible
===	Égalité stricte
!=	Inégalité faible
!==	Inégalité stricte

Opérateurs logiques

&&	ET logique
	OU logique
!	NON logique

```
let a = '2'

if (typeof a !== undefined && a !== null) {
  if (a === 1) {
    console.log('a est 1')
  } else if (a >= 5 || a === 10) {
    console.log('a est soit 8, soit supérieur à 2')
  } else {
    console.log('a n\'est pas entre 1 et 5')
  }
} else {
  console.log('a est null ou non défini')
}
```

Les fonctions

```
// Fonction nommée (hoisted)
console.log(sum(7, 4)) // => 11
function sum(x, y) {
  return x + y
}
```

```
// Fonction anonyme
console.log(diff(7, 4)) // => ERREUR
const diff = function (x, y) {
  return x - y
}
console.log(diff(7, 4)) // => 3
```

```
// Fonction anonyme avec flèche (conserve le scope)
const times = (x, y) => { return x * y }
```

```
// Fonction anonyme avec return implicite (conserve le scope)
const times = (x, y) => x * y
```

Si un seul paramètre, les parenthèses des fonctions fléchées sont facultatifs

Les objets et tableaux

```
// Les objets
const user = {
  firstname: 'Jean'
  lastname: 'Dubois'
}

const user2 = user
user2.firstname = 'Marc'
console.log(user.firstname) // => Marc
console.log(user['firstname']) // => Marc

// Les tableaux
const languages = [ 'fr', 'en', 'es' ]
user.language = languages[0]

// Le résultat ??
console.log(user)
```

Les itérations (for)

```
const fruits = ['apple', 'orange', 'strawberry', 'blueberry']
const l = fruits.length
for (let i = 0; i < l ; i++) {
  console.log(1, fruits[i])
}
```

```
for (let i = fruits.length; i--; ) {
  console.log(2, fruits[i])
}
```

```
for (let i = 0; i < l ; i++) {
  if (fruits[i] === 'apple') { continue }
  console.log(3, fruits[i])
  if (fruits[i] === 'strawberry') { break }
}
```

Les itérations (while)

```
let found = false
let i = 0
while (!found) {
  if (fruits[i] === 'strawberry') {
    found = true
  }
  console.log(4, fruits[i])
  i++
}
```

La portée des variables (SCOPE)

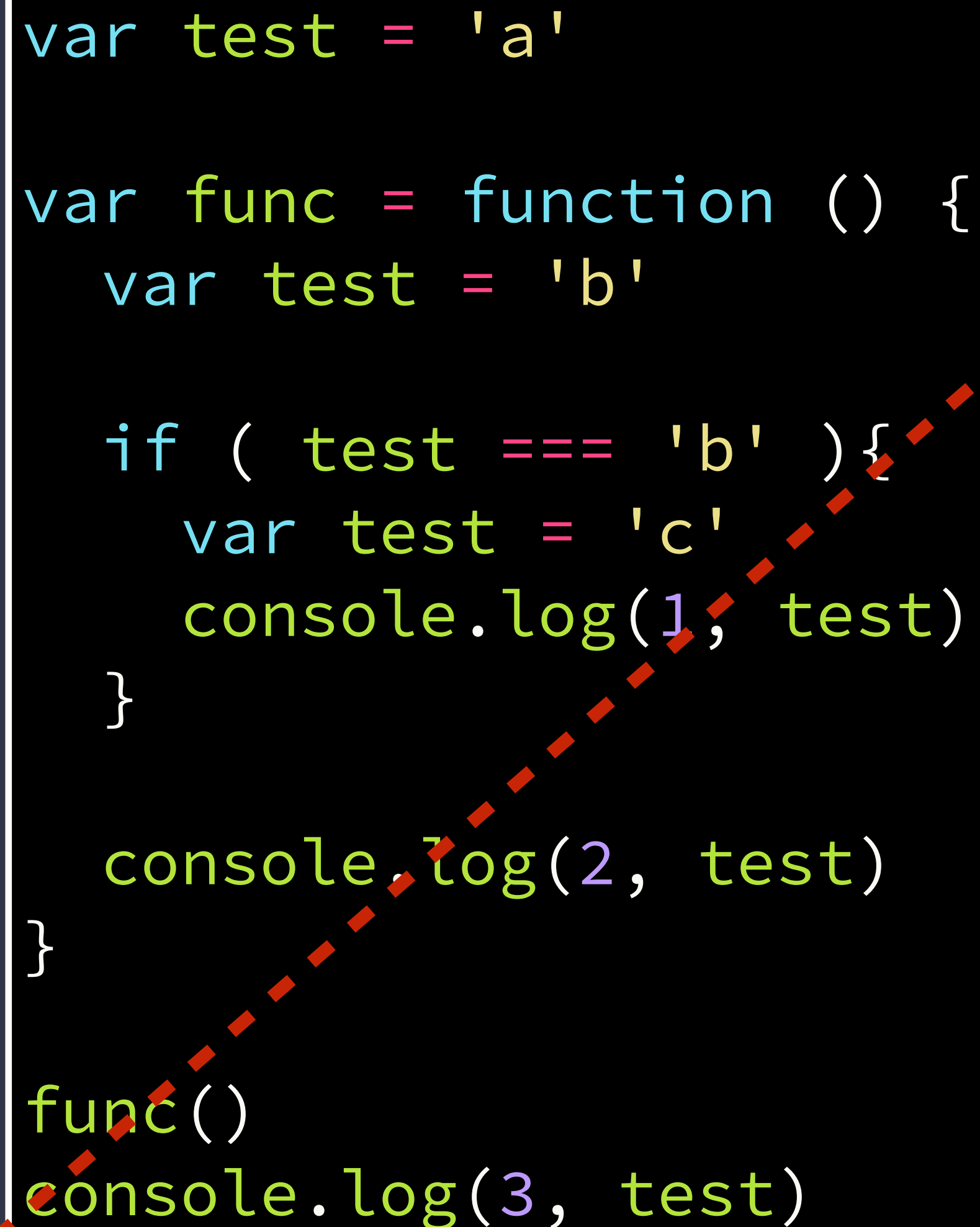
```
var test = 'a'

var func = function () {
  var test = 'b'

  if ( test === 'b' ) {
    var test = 'c'
    console.log(1, test)
  }

  console.log(2, test)
}

func()
console.log(3, test)
```



```
let test2 = 'a'

const func2 = function() {
  let test2 = 'b'

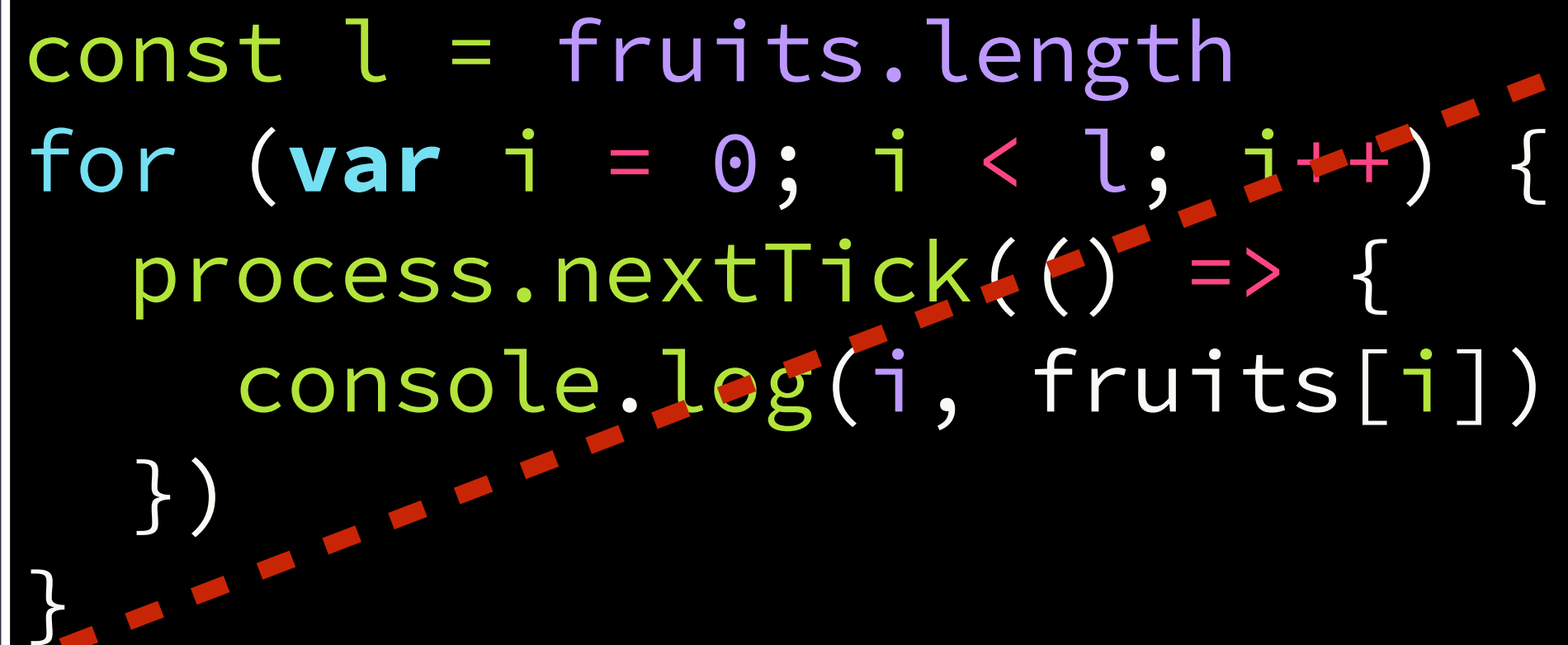
  if ( test2 === 'b' ) {
    let test2 = 'c'
    console.log(11, test2)
  }

  console.log(12, test2)
}

func2()
console.log(13, test2)
```


Careful of SCOPE

```
const l = fruits.length
for (var i = 0; i < l; i++) {
  process.nextTick(() => {
    console.log(i, fruits[i])
  })
}
```



```
const l = fruits.length
for (let i = 0; i < l; i++) {
  process.nextTick(() => {
    console.log(i, fruits[i])
  })
}
```

process.nextTick : Décale l'exécution de la fonction en paramètre à la prochaine boucle de l'event loop

NE JAMAIS UTILISER LE CODE DE GAUCHE

TD : Le jeu du plus ou moins

Memo & Tips

```
// Initialisation
const rl = require('readline').createInterface({
  input: process.stdin, output: process.stdout
})

// Pose une question à l'utilisateur
rl.question('Question ?', (answer) => {
  process.stdout.write("Tu as répondu : " + answer + "\n")
})

// Retourne un entier aléatoire entre 0 et 10
Math.floor(Math.random() * 10)

// RegExp qui retourne true si input contient 1 à 3 chiffres
/^\\d{1,3}$/.test(input)

// Quitte le programme
process.exit()
```

Au lancement, le script choisit un nombre de 0 à 999.

Le but du jeu est de trouver ce nombre.

À chaque mauvaise réponse, le script indique simplement si le nombre est supérieur ou inférieur.

Langage Prototypé

Langage prototypé

```
// foo hérite du prototype de Object
foo = new Object()
foo.bar = 'toto'

// le code ci dessus est l'équivalent de
foo = { bar: 'toto' }
```

- Tout type non primitif est un objet
- Objets ont des prototypes
- Objets héritent les propriétés de leur prototype
- L'opérateur `new` crée une instance d'un objet en appelant la méthode `constructor` du prototype

```
// Création d'un prototype avec un constructeur
const User = function (firstname, lastname) {
  this.firstname = firstname
  this.lastname = lastname
}

// Ajout d'une méthode au prototype de Person
User.prototype.name = function() {
  return this.firstname + ' ' + this.lastname
}

// Utilisation de notre classe
user = new User('Jean', 'Bon')
user.name() // => "Jean Bon"
```

Propriétés avancées

Définition de propriétés

```
const foo = {}

Object.defineProperty(foo, 'prop1', {
  value: 'Valeur 1',
  configurable: true,
  enumerable: false,
  writable: false
})

Object.defineProperty(foo, 'prop2', {
  value: 'Valeur 2',
  configurable: false,
  enumerable: true,
  writable: true
})

// prop1 n'est pas énumérable
console.log(foo)
console.log(foo.prop1)

// prop1 ne peut être modifié
foo.prop1 = 'New Valeur 1'
foo.prop2 = 'New Valeur 2'
console.log(foo.prop1, foo.prop2)

// prop2 n'est pas configurable
delete foo.prop1 // Working
delete foo.prop2 // Not possible
```

Getter / Setter

```
const bar = {}
let barValue = 3

Object.defineProperty(bar, 'value', {
  enumerable: true,
  get: function () { return barValue },
  set: function (val) { barValue = val }
})

console.log(bar)

bar.value = 10
console.log(barValue)
```

Plus de fun

```
// Lister les propriétés
console.log(Object.getOwnPropertyNames(foo))

// Récupérer des infos sur une propriété
console.log(Object.getOwnPropertyDescriptor(foo, 'prop1'))

// Sceler un objet
const qux = { test: 10 }
Object.seal(qux)

qux.newProp = 'kek' // not working
qux.test = 20 // still working
console.log(qux)
delete qux.test // not working

// Verrouiller un objet
const baz = { test: 10 }
Object.freeze(baz)

baz.newProp = 'kek' // not working
baz.test = 20 // not working
console.log(baz)
```

Plus facile avec les classes ES6

```
class Employee {
  constructor (firstname, lastname) {
    this.firstname = firstname
    this.lastname = lastname
  }

  static createFromName (name) {
    const p = new Employee()
    p.name = name
    return p
  }

  get name () {
    return `${this.firstname} ${this.lastname}`
  }

  set name (name) {
    [ this.firstname, this.lastname ] = name.split(' ')
  }

  sayHello () {
    return `Bonjour ${this.firstname} !`
  }
}

let jeanEmployee = new Employee('Jean', 'Bon')
let emilieEmployee = Employee.createFromName('Emilie Bond')
```

Les classes ES6 : Héritage

```
class Boss extends Employee {  
  get name () {  
    return `Mr. ${super.name}`  
  }  
  
  stressOut (person) {  
    return `Plus vite que ça ${person.firstname} !!`  
  }  
}  
  
let michelBoss = new Boss('Michel', 'Dubois')  
michelBoss.stressOut(jeanEmployee) // "Plus vite que ça Jean !!"  
michelBoss.name // "Mr. Michel Dubois"
```


Careful of SCOPE

```
class Hello {  
  constructor (message) {  
    this.message = message  
  }  
  
  waitAndLogMessage () {  
    setTimeout(() => {  
      console.log(this.message)  
    }, 1000)  
  }  
  
  waitAndTryToLogMessage () {  
    setTimeout(function () {  
      console.log(this.message || 'No message to  
log...')  
    }, 1000)  
  }  
}  
  
const hello = new Hello('Hey !')  
hello.waitAndLogMessage()  
hello.waitAndTryToLogMessage()
```

Lorsqu'on utilise une fonction fléchée anonyme, le scope est conservé donc **this** correspond bien à celui de l'instance de **Hello**.

Par contre avec une fonction anonyme, le scope n'est pas conservé, et le **this** correspond donc au contexte de la méthode qui appelle la fonction (en l'occurrence c'est le **setTimeout** qui décide ce que sera le **this**).

Langage prototypé

```
// foo hérite du prototype de Object
foo = new Object()
foo.bar = 'toto'

// le code ci dessus est l'équivalent de
foo = { bar: 'toto' }
```

- Tout type non primitif est un objet
- Objets ont des prototypes
- Objets héritent les propriétés de leur prototype
- L'opérateur `new` crée une instance d'un objet en appelant la méthode `constructor` du prototype

```
// Création d'un prototype avec un constructeur
const Person = function (firstname, lastname) {
  this.firstname = firstname
  this.lastname = lastname
}

// Ajout d'une méthode au prototype de Person
Person.prototype.name = function() {
  return this.firstname + ' ' + this.lastname
}

// Utilisation de notre classe
p = new Person('Jean', 'Bon')
p.name() // => "Jean Bon"
```

Propriétés avancées

Définition de propriétés

```
const foo = {}

Object.defineProperty(foo, 'prop1', {
  value: 'Valeur 1',
  configurable: true,
  enumerable: false,
  writable: false
})

Object.defineProperty(foo, 'prop2', {
  value: 'Valeur 2',
  configurable: false,
  enumerable: true,
  writable: true
})

// prop1 n'est pas énumérable
console.log(foo)
console.log(foo.prop1)

// prop2 n'est pas configurable
delete foo.prop1 // Working
delete foo.prop2 // Not possible

// prop1 ne peut être modifié
foo.prop1 = 'New Valeur 1'
foo.prop2 = 'New Valeur 2'
console.log(foo.prop1, foo.prop2)
```

Getter / Setter

```
const bar = {}
let barValue = 3

Object.defineProperty(bar, 'value', {
  enumerable: true,
  get: function () { return barValue },
  set: function (val) { barValue = val }
})

console.log(bar)

bar.value = 10
console.log(barValue)
```

Plus de fun

```
// Lister les propriétés
console.log(Object.getOwnPropertyNames(foo))

// Récupérer des infos sur une propriété
console.log(Object.getOwnPropertyDescriptor(foo, 'prop1'))

// Sceler un objet
const qux = { test: 10 }
Object.seal(qux)

qux.newProp = 'kek' // not working
qux.test = 20 // still working
console.log(qux)
delete qux.test // not working

// Verrouiller un objet
const baz = { test: 10 }
Object.freeze(baz)

baz.newProp = 'kek' // not working
baz.test = 20 // not working
console.log(baz)
```

Plus facile avec les classes ES6

```
class Employee {
  constructor (firstname, lastname) {
    this.firstname = firstname
    this.lastname = lastname
  }

  static createFromName (name) {
    const p = new Employee()
    p.name = name
    return p
  }

  get name () {
    return `${this.firstname} ${this.lastname}`
  }

  set name (name) {
    [ this.firstname, this.lastname ] = name.split(' ')
  }

  sayHello () {
    return `Bonjour ${this.firstname} !`
  }
}

let jeanEmployee = new Employee('Jean', 'Bon')
let emilieEmployee = Employee.createFromName('Emilie Bond')
```

Les classes ES6 : Héritage

```
class Boss extends Employee {  
  get name () {  
    return `Mr. ${super.name}`  
  }  
  
  stressOut (person) {  
    return `Plus vite que ça ${person.firstname} !!`  
  }  
}  
  
let michelBoss = new Boss('Michel', 'Dubois')  
michelBoss.stressOut(jeanEmployee) // "Plus vite que ça Jean !!"  
michelBoss.name // "Mr. Michel Dubois"
```

Careful of SCOPE

```
class Hello {
  constructor (message) {
    this.message = message
  }

  waitAndLogMessage () {
    setTimeout(() => {
      console.log(this.message)
    }, 1000)
  }

  waitAndTryToLogMessage () {
    setTimeout(function () {
      console.log(this.message || 'No message to
log...')
    }, 1000)
  }
}

const hello = new Hello('Hey !')
hello.waitAndLogMessage()
hello.waitAndTryToLogMessage()
```

Lorsqu'on utilise une fonction fléchée anonyme, le scope est conservé donc **this** correspond bien à celui de l'instance de **Hello**.

Par contre avec une fonction anonyme, le scope n'est pas conservé, et le **this** correspond donc au contexte de la méthode qui appelle la fonction (en l'occurrence c'est le **setTimeout** qui décide ce que sera le **this**).

TD : Le jeu du plus ou moins

Memo & Tips

```
// Initialisation
const rl = require('readline').createInterface({
  input: process.stdin, output: process.stdout
})

// Pose une question à l'utilisateur
rl.question('Question ?', (answer) => {
  process.stdout.write("Tu as répondu : " + answer + "\n")
})

// Retourne un entier aléatoire entre 0 et 10
Math.floor(Math.random() * 10)

// RegExp qui retourne true si input contient 1 à 3 chiffres
/^\\d{1,3}$/.test(input)

// Quitte le programme
process.exit()
```

Au lancement, le script choisit un nombre de 0 à 999.

Le but du jeu est de trouver ce nombre.

À chaque mauvaise réponse, le script indique simplement si le nombre est supérieur ou inférieur.

Gestionnaire de package

Ça sert à quoi ?

- Installation et réinstallation aisée des modules/library
- Gère les métadonnées d'un projet ou package
- Mise à jour des modules aisée
- Partage des dépendances avec l'équipe
- Verrouillage des versions des modules pour un projet donné

NPM

Node Package Manager

- Le plus grand nombre de package
- Compatible GIT
- Autour d'un fichier `package.json`
- Gestion des versions majeure, mineure, etc...
- Gestion de scripts
- Publication d'un nouveau package



Le package.json

<package.json>

```
{
  "name": « njs-301",
  "version": "0.0.1",
  "description": "Introduction à node.js",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "author": "Jeremy Trufier <jeremy@wikodit.fr>",
  "license": "MIT",
  "dependencies": {
    "lodash": "^4.16.2"
  }
}
```

Gestionnaire de package

Nouveau dossier
Navigation dans le dossier
Initialisation interactive du projet
Installation de lodash

Lancement de node
Appel d'une fonction de lodash
Echec: `_` n'existe pas
On charge lodash

La méthode fonctionne

Terminal

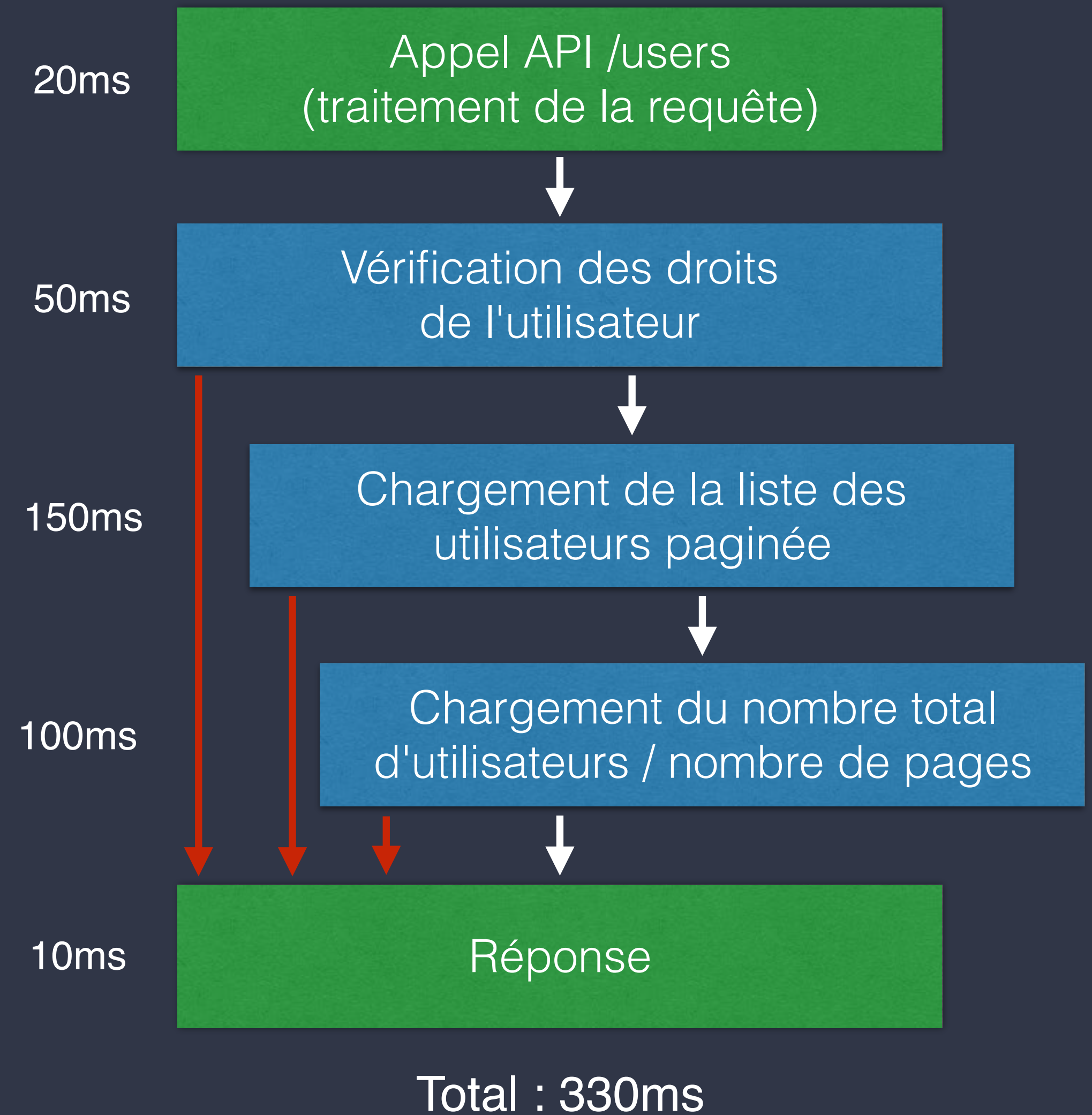
```
$ mkdir njs-301
$ cd njs-301
$ npm init
$ npm install --save lodash

$ node
> _.camelCase('Foo Bar')
TypeError: Cannot read property 'camelCase' of undefined
> const _ = require('lodash')
> _.camelCase('Foo Bar')
'fooBar'
```

Asynchrone et événements

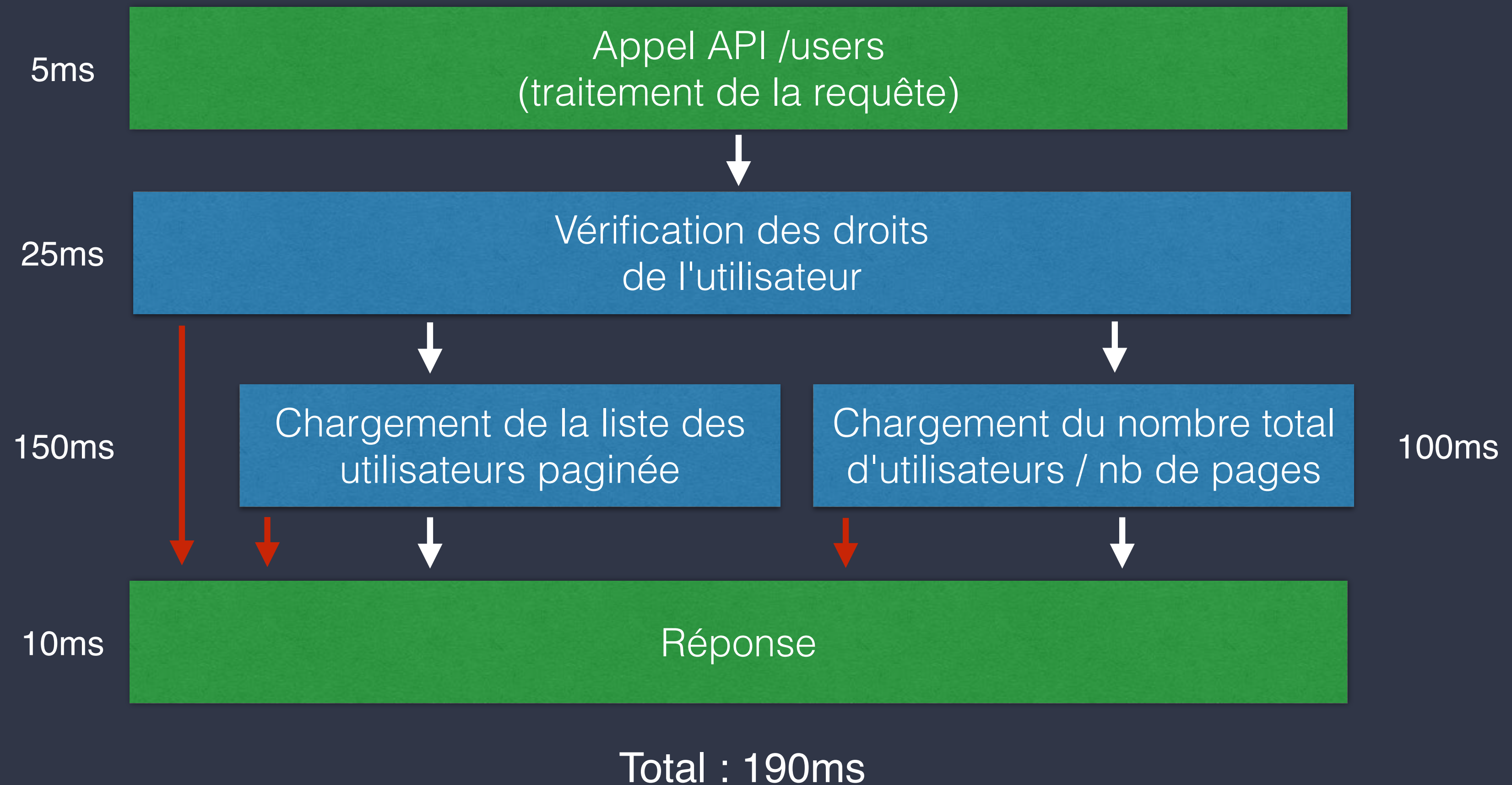
Le code synchrone

- Procédural
- En pause lors des I/O (bloquant)
- Dans certains langage : création d'un nouveau thread
- Temps de traitement total = temps de chaque partie additionnée



Le code asynchrone

- Parallélisation
- Optimisation
- Thread unique
- Event-Loop



nodeJS est asynchrone !

Synchrone

```
const fs = require('fs')
let filepath = '/etc/passwd'

try {
  fs.accessSync(filepath, fs.constants.R_OK)
  console.info(`I can read ${filepath}`)
} catch (e) {
  console.error(`No access to ${filepath}`)
}

console.log('This is written last!')
```

Asynchrone

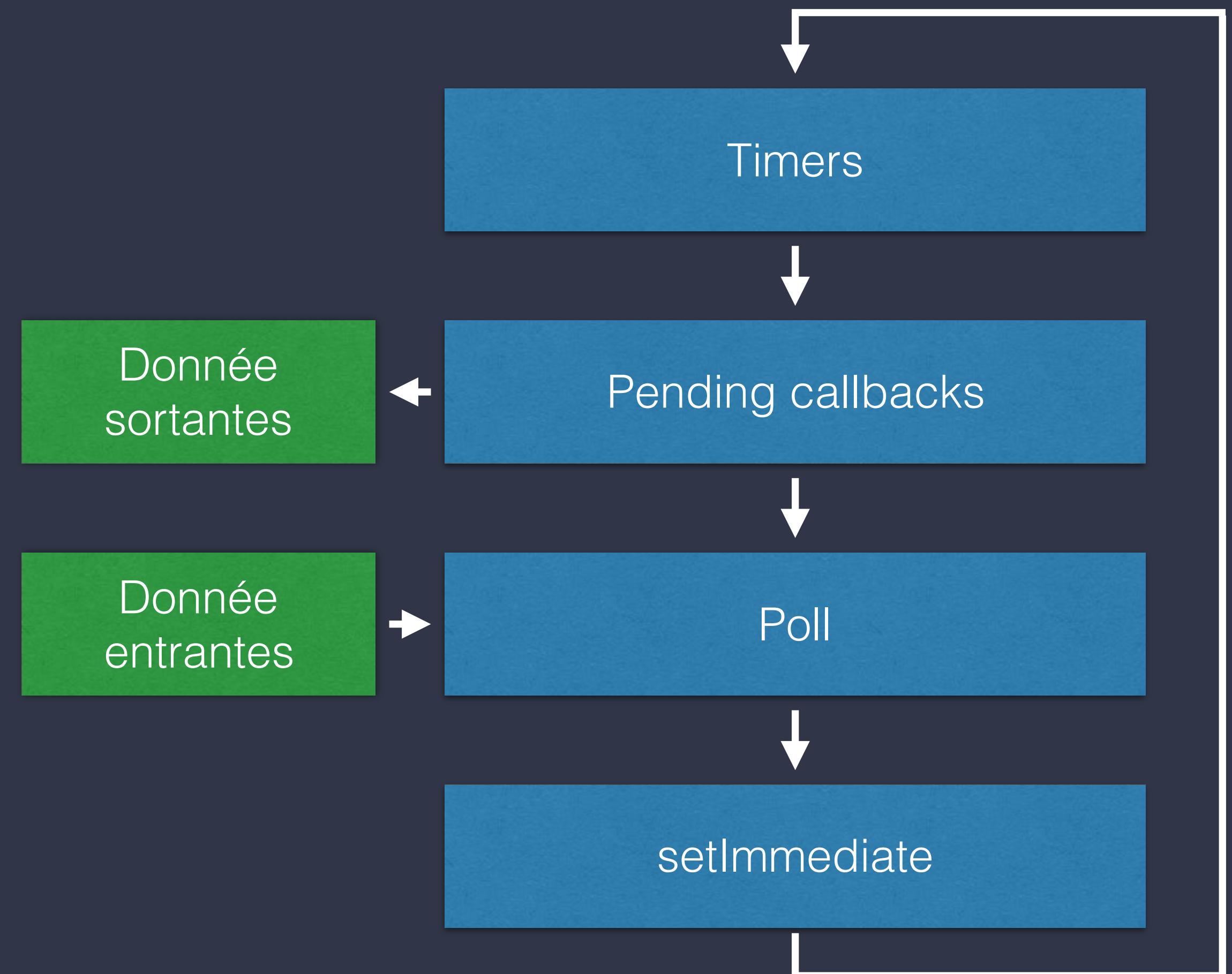
```
const fs = require('fs')
let filepath = '/etc/passwd'

fs.access(filepath, fs.constants.R_OK, (err) => {
  if (err) {
    console.error(`No access to ${filepath}`)
  } else {
    console.info(`I can read ${filepath}`)
  }
})

console.log('This is written first!')
```

Event loop

- Event Driven Programming
- Grande stabilité
- Peut gérer de grosses charges
- Jamais en pause ou en attente



Utiliser les évènements

register-event.js

```
process.stdin.setEncoding('utf8');

process.stdin.on('readable', () => {
  var chunk = process.stdin.read()
  if (chunk !== null) {
    process.stdout.write(`stdin datas: ${chunk}`)
  }
})

process.stdin.on('end', () => {
  process.stdout.write('end')
})

console.log('Program started')
```

Terminal

```
$ echo 'Hello world' | node register-events.js
Program started
stdin datas: Hello world
end
```

Simuler une fonction Asynchrone

```
function divide (a, b, callback) {  
  process.nextTick(() => {  
    try {  
      if (b == 0) { throw new Error('Can not divide by 0') }  
  
      let result = a / b  
  
      if(isNaN(result)) { throw new Error('Can\'t divide those') }  
  
      callback(null, result)  
    } catch (e) {  
      callback(e)  
    }  
  })  
}  
  
let showResults = (err, result) => console.log(err, result)  
  
divide(10, 4, showResults)  
divide(5, 0, showResults)  
console.log('Written first')
```

Emettre des évènements

```
const EventEmitter = require('events')

class Divider extends EventEmitter {
  constructor (divisor) {
    super()
    this.divisor = divisor
  }

  run (numerator) {
    process.nextTick(() => {
      this.emit('start')

      try {
        let result = numerator / this.divisor
        if(isNaN(result)) {
          throw new Error('Can\'t divide those')
        }

        this.emit('result', result)
      } catch (e) {
        this.emit('error', e)
      }

      this.emit('end')
    })
  }
}
```

```
let divideByTwo = new Divider(2)

divideByTwo.run(3)

divideByTwo.on('start', () => {
  console.log('started')
})
divideByTwo.on('result', (r) => {
  console.log(`result is ${r}`)
})
divideByTwo.on('end', () => {
  console.log('ended')
})

console.log('still printed first')

// still printed first
// started
// result is 1.5
// ended
```

Les promesses

Le problème des callbacks

- Utile pour de simples opérations
- Continuous Passing Style (CPS)
 - Séquence de fonctions compliquée
 - Parallélisation encore pire
 - Gestion d'erreur catastrophique

CPS en série

```
currentUser.checkAccess('users', function(err, access) {  
  if (err) { return console.error(err) }  
  
  Users.find().exec(function(err, users){  
    if (err) { return console.error(err) }  
  
    response.data = users  
  
    User.count({}, function(err, count){  
      if (err) { return console.error(err) }  
  
      response.meta.count = count  
      console.log(response)  
    })  
  })  
})
```

CPS en parallèle

```
let stuffToBeFinished = 0

stuffToBeFinished += !!Users.find().exec(function(err, users){
  if (err) { return console.error(err) }

  response.data = users
  finishedStuff()
})

stuffToBeFinished += !!Users.count({}, function(err, count){
  if (err) { return console.error(err) }

  response.meta.count = count
  finishedStuff()
})

function finishedStuff () {
  if (--stuffToBeFinished) { return }
  console.log(response)
}
```

Les solutions

Des librairies :

- `async.js`
- `async`
- `chainsaw`
- `each`
- `flow-js`
- ...

ou

Les Promises

La base

CPS (sans Promise)

```
User.find((err, users) => {  
  if (err) {  
    return console.log('Error: ', err)  
  }  
  console.log('All users: ', users)  
})  
console.log('This is written first !')
```

Avec Promise

```
User.find().then((users) => {  
  console.log('All users: ', users)  
}).catch((err) => {  
  console.log('Error: ', err)  
})  
console.log('This is written first !')
```

*(**User** est une classe avec une méthode statique **find**)*

L'objet Promise

Des Promise déjà pré-résolues ou pré-rejetées

```
let promiseOk = Promise.resolve('ok')
let promiseError = Promise.reject('erreur')
```

(Pour tester)

```
promiseOk.then((msg) => {
  console.log('success: ', msg)
}).catch((msg) => {
  console.log('error: ', msg)
})

promiseError.then((msg) => {
  console.log('success: ', msg)
}).catch((msg) => {
  console.log('error: ', msg)
})

console.log('end')
```

Des Promise asynchrone avec **Promise.new**

```
let promiseOk = new Promise((resolve, reject) => {
  resolve('ok')
})

let promiseError = new Promise((resolve, reject) => {
  reject('error')
})
```

Chainer les Promise

1. Un **then** ou un **catch** retournera TOUJOURS une **Promise**
2. Une **Promise** a un état rejeté, résolu ou en cours
3. Les **then** permettent de gérer les promesses acceptées
4. Les **catch** permettent de gérer les promesses rejetées
5. Un **then** sur une promesse rejeté, n'exécute pas la fonction et retourne la même promesse
6. Un **catch** sur une promesse résolue, n'exécute pas la fonction et retourne la même promesse
7. Toute **Promise** retournée par la fonction de callback du **then/catch** sera retournée par le **then/catch**
8. Si la fonction de callback **throw** une erreur alors la **Promise** retournée par le **then/catch** sera implicitement rejetée, dans tous les autres cas elle sera résolue

Chainer les Promise

1. Un **then** ou un **catch** retournera TOUJOURS une **Promise**
2. Une **Promise** a un état rejeté, résolu ou en cours
3. Les **then** permettent de gérer les promesses acceptées
4. Les **catch** permettent de gérer les promesses rejetées
5. Un **then** sur une promesse rejetée, n'exécute pas la fonction et retourne la même promesse
6. Un **catch** sur une promesse résolue, n'exécute pas la fonction et retourne la même promesse
7. Toute **Promise** retournée par la fonction de callback du **then/catch** sera retournée par le **then/catch**
8. Si la fonction de callback **throw** une erreur alors la **Promise** retournée par le **then/catch** sera implicitement rejetée, dans tous les autres cas elle sera résolue

```
Promise.reject('error 1').then(() => {
  console.log(1, 'nop')
}).catch((err) => {
  console.log(2, 'yep', err) // error 1
  return 3
}).then((val) => {
  console.log(3, 'yep', val) // 3
  return Promise.resolve('test')
}).then((val) => {
  console.log(4, 'yep', val) // test
  return Promise.reject('whatever')
}).catch((err) => {
  console.log(5, 'yep', err) // whatever
}).then(() => {
  console.log(6, 'yep')
  throw new Error('error 2')
  console.log(7, 'nop')
}).then(() => {
  console.log(8, 'nop')
}).catch((err) => {
  console.log(9, 'yep', err.message) // error 2
})
```

Et donc ?

- Chaînage
- Retours uniformes
- Organisation du code aisée
- Code en série ou en parallèle facile
- Asynchrone
- Erreurs regroupées

Note: le module de promesses Bluebird est plus performant que les Promise ES6 natives, à privilégier

```

class User {
  constructor (pseudo, permissions) {
    this.pseudo = pseudo
    this.permissions = permissions
  }

  checkAccess (resource) {
    return new Promise((resolve, reject) => {
      process.nextTick(() => {
        if (~this.permissions.indexOf(resource)) {
          resolve(true)
        } else {
          reject(new Error('Forbidden'))
        }
      })
    })
  }

  static find () {
    return new Promise((resolve, reject) => {
      process.nextTick(() => {
        resolve([
          new User('Jean', []),
          new User('Michel', []),
        ])
      })
    })
  }
}

```

Pour la suite...

On crée un pseudo model User :
 Normalement les méthodes définies font des appels à une base de donnée, pour simuler l'asynchrone on utilise `process.nextTick()`

```

let asterix = new User('asterix', ['users', 'comments'])

asterix.checkAccess('comments').then((access) => {
  console.log(access)
}).catch((err) => {
  console.error(err)
})

```


Les Promises ES6 en séquentiel

```
const userIndexAction = (params) => {  
  const response = { data: null, meta: {} }  
  
  return params.currentUser.checkAccess('users').then(() => {  
    return User.find()  
  }).then((users) => {  
    response.data = users  
    return User.count()  
  }).then((count) => {  
    response.meta.count = count  
    return response  
  })  
}
```

ATTENTION BONNE PRATIQUE : Un chaîne de promesse doit toujours posséder un **catch final**



Pour tester

```
const params = {  
  currentUser: new User('asterix', ['users']),  
  count: true,  
}  
  
userIndexAction(params).then((resp) => {  
  console.log('Response: ', resp)  
}).catch((err) => {  
  console.error('userIndexAction: ', err)  
})  
  
console.error('ce texte sera écrit en 1er')
```

Encore en séquence

```
const userIndexAction = (params) => {  
  const response = { data: null, meta: {} }  
  
  let promise = Promise.resolve().then(() => {  
    return params.currentUser.checkAccess('users')  
  }).then(() => {  
    return User.find()  
  }).then((users) => {  
    response.data = users  
  })  
  
  if (params.count === true) {  
    promise = promise.then(() => {  
      return User.count()  
    }).then((count) => {  
      response.meta.count = count  
    })  
  }  
  
  return promise.then(() => {  
    return response  
  })  
}
```

Ce code fait la même chose que celui de la page précédente à l'exception de :

- Le nombre d'utilisateur est calculé et envoyé conditionnellement (à la demande)
- On commence par une promesse déjà résolue **Promise.resolve()** pour augmenter la lisibilité

Les Promises en parallèle

```
const userIndexAction = (params) => {  
  let promise = params.currentUser.checkAccess('users')  
  
  return promise.then(() => {  
    const promises = [ User.find() ]  
  
    if (params.count === true) {  
      promises.push(User.count())  
    }  
  
    return Promise.all(promises)  
  }).then((results) => {  
    return {  
      data: results[0],  
      meta: { count: results[1] }  
    }  
  })  
}
```

Pour tester

```
const params = {  
  currentUser: new User('asterix', ['users']),  
  count: true,  
}  
  
userIndexAction(params).then((resp) => {  
  console.log('Response: ', resp)  
}).catch((err) => {  
  console.error('userIndexAction: ', err)  
})  
  
console.error('ce texte sera écrit en 1er')
```

Lisibilité accrue avec async / await

```
function defer(x, t = 1000) {  
  return new Promise((resolve) => {  
    setTimeout(() => {  
      resolve(x)  
    }, t)  
  })  
}  
  
const addParallel = async (a, b) => {  
  const promiseA = defer(a, 2000)  
  const promiseB = defer(b, 3000)  
  
  return await promiseA + await promiseB  
}  
  
const addSequential = async (a, b) => {  
  const resultA = await defer(a, 2000)  
  const resultB = await defer(b, 3000)  
  
  return resultA + resultB  
}
```

```
const run = () => {  
  console.time('parallel time')  
  console.time('sequential time')  
  
  addSequential(2, 3).then((result) => {  
    console.log('Sequential', result)  
    console.timeEnd('sequential time')  
  })  
  
  addParallel(4, 5).then((result) => {  
    console.log('Parallel', result)  
    console.timeEnd('parallel time')  
  })  
}  
  
run()
```

Async et Await permettent d'utiliser les promesses avec beaucoup plus de simplicité

En parallèle avec async/ await

```
const userIndexAction = async (params) => {  
  await params.currentUser.checkAccess('users')  
  
  const dataPromise = User.find()  
  let metaPromise = Promise.resolve({})  
  
  if (params.count) {  
    metaPromise = User.count()  
  }  
  
  return { data: await dataPromise, meta: await metaPromise }  
}
```

Pour tester

```
const params = {  
  currentUser: new User('asterix', ['users']),  
  count: true,  
}  
  
userIndexAction(params).then((resp) => {  
  console.log('Response: ', resp)  
}).catch((err) => {  
  console.error('userIndexAction: ', err)  
})  
  
console.error('ce texte sera écrit en 1er')
```

Export / Import

Les modules

- Il est indispensable d'exposer certaines méthodes à partir de modules ou de fichiers
- On doit pouvoir importer et utiliser des fonctions de modules ou de fichiers

Import de modules

ATTENTION : Le support de la syntaxe ES6 n'est pas encore gérée par nodeJS et nécessite l'utilisation d'un transpiler tel que Babel ou TypeScript

CommonJS

```
const _ = require('lodash')

const ages = _.compact(_.map([
  { username: 'Fool' },
  { username: 'Fitz', age: 50 },
  { username: 'Bee', age: 14 },
], 'age'))

console.log(ages)
```

ES6

```
import * as _ from 'lodash'

const ages = _.compact(_.map([
  { username: 'Fool' },
  { username: 'Fitz', age: 50 },
  { username: 'Bee', age: 14 },
], 'age'))

console.log(ages)
```

ES6

```
import { map, compact } from 'lodash'

const ages = compact(map([
  { username: 'Fool' },
  { username: 'Fitz', age: 50 },
  { username: 'Bee', age: 14 },
], 'age'))

console.log(ages)
```

*Nécessite de **npm install lodash** pour que ce code fonctionne*

Exporter des modules CommonJS

models/user.js

```
module.exports = class User {  
  constructor () {  
    console.log('user created')  
  }  
}
```

models/message.js

```
module.exports = class Message {  
  constructor () {  
    console.log('message created')  
  }  
}
```

models/index.js

```
module.exports = {  
  Message: require('./message')  
  User: require('./user')  
}
```

Pour utiliser notre classe User :

app.js

```
const User = require('./models/user')  
new User()
```

Ou grâce à **models/index.js** :

app.js

```
const { User } = require('./models')  
new User()
```

Exporter des modules ES6

models/user.js

```
export default class User {  
  constructor () {  
    console.log('user created')  
  }  
}
```

models/message.js

```
export class Message {  
  constructor () {  
    console.log('message created')  
  }  
}
```

models/index.js

```
export * from './message'  
export * from './user'
```

Différents moyens d'importer nos classes :

```
import { User } from './models/user'  
new User()
```

```
import * as User from './models/user'  
new User()
```

Ou grâce au mot clé **default** qui permet d'avoir une valeur d'export par défaut :

```
import LaClasseUser from './models/user'  
new LaClasseUser()
```

Ou grâce à **models/index.js** :

app.js

```
import { User, Message } from './models'  
new User()
```


TP: Chat

Objectifs

- Familiarisation avec le langage
- Familiarisation avec la création d'un serveur Web sans framework
- Utilisation des promesses
- Utilisation des classes
- Utilisation des événements
- Utilisation de ES6

Ce TP permet une remise à niveau grâce à l'implémentation d'un serveur Web très simple et d'une connection à une base de donnée.

Si les concepts sont familiers, ce TP est facultatif.

TP : Chat

Serveur Web

Afficher une simple page avec :

- *Une liste de messages*
 - *Pseudo*
 - *Message*
 - *Heure*
- *Un formulaire*
 - *Pseudo*
 - *Message*
 - *Bouton Envoyer*

*On s'attarde au côté fonctionnel
et qualité du code, pas de style*

```
const http = require('http')
const qs = require('querystring')

// Simple serveur WEB
http.createServer((req, res) => {
  console.log(req.method) // GET ou POST
  console.log(req.url) // '/chat'

  // SOIT une page HTML
  res.writeHead(200, {'Content-Type': 'text/html'})

  // SOIT une redirection vers "/chat"
  res.writeHead(302, {'Location': '/chat'})
  // Écriture d'une page HTML
  res.write(`
    <html>
      <body>HTML Page</body>
    </html>
  `)
  res.end() // Toutes les données ont été envoyées
}).listen(8081)
```

TP : Chat (aide)

Récupération données POST du formulaire lors d'une requête

```
// Il suffit d'écouter deux évènements sur l'objet req :
// - 'data' => Renvoie les données du formulaire au fur à mesure qu'elles arrivent
// - 'end' => Appelé une fois que toutes les données sont arrivées
const body = null
const dataBuffers = []

req.on('data', (data) => { dataBuffers.push(data) })
req.on('end', () => {
  body = Buffer.concat(dataBuffers).toString()
  console.log('All data has been received and can now be used: ', body)
})

// qs permet de parser des données :
let params = qs.parse('foo=3&bar=5')
console.log(params.foo) // => 3
```

TP : Chat (aide)

Utilisation d'un moteur de template (pug, ejs, handlebars, ...)

1. Pré-requis (ne pas oublier `npm install pug`)

```
const pug = require('pug')

const tplIndexPath = './views/index.pug'
const renderIndex = pug.compileFile(tplIndexPath)
```

2. `views/index.pug`

```
doctype html
html
  head
    title #{title}
  body
    h1 Voici les fruits
    ul
      each fruit in fruits
        li #{fruit}
```

3. Puis lors d'une requête cliente pour render `views/index.pug`

```
const html = renderIndex({
  title: 'Hello world',
  fruits: [ 'banana', 'apple' ]
})

res.writeHead(200, { 'Content-Type': 'text/html' } )
res.write(html)
res.end()
```

Résultat :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Les fruits</title>
  </head>
  <body>
    <h1>Voici les fruits</h1>
    <ul>
      <li>banana</li>
      <li>apple</li>
    </ul>
  </body>
</html>
```

TP : Chat (aide)

Stockage des données avec Sequelize :

<http://docs.sequelizejs.com/manual/installation/getting-started.html#installation>

Sequelize utilise les Promesses !!

TP : Chat (aide)

Exemple de HTML en sortie :

```
<html>
  <body>
    <h1>Chat</h1>
    <div class="messages">
      <p class="message">
        Jean le 10/05/2024 : Oui ça marche bien
      </p>
      <p class="message">
        Emilie le 09/05/2024 : Bonjour, c'est super !
      </p>
    </div>
    <form method="post">
      <label for="pseudo">
        Pseudo :
        <input type="text" id="pseudo" name="pseudo" />
      </label>
      <br />
      <label for="message">
        Commentaire :
        <textarea id="message" name="message"></textarea>
      </label>
      <br />
      <input type="submit" value="Envoyer" />
    </form>
  </body>
</html>
```

TP : Chat

Objectifs

- Familiarisation avec le langage
- Familiarisation avec la création d'un serveur Web sans framework
- Utilisation des promesses
- Utilisation des classes
- Utilisation des évènements
- Utilisation de ES6

TP : Chat (Guide)

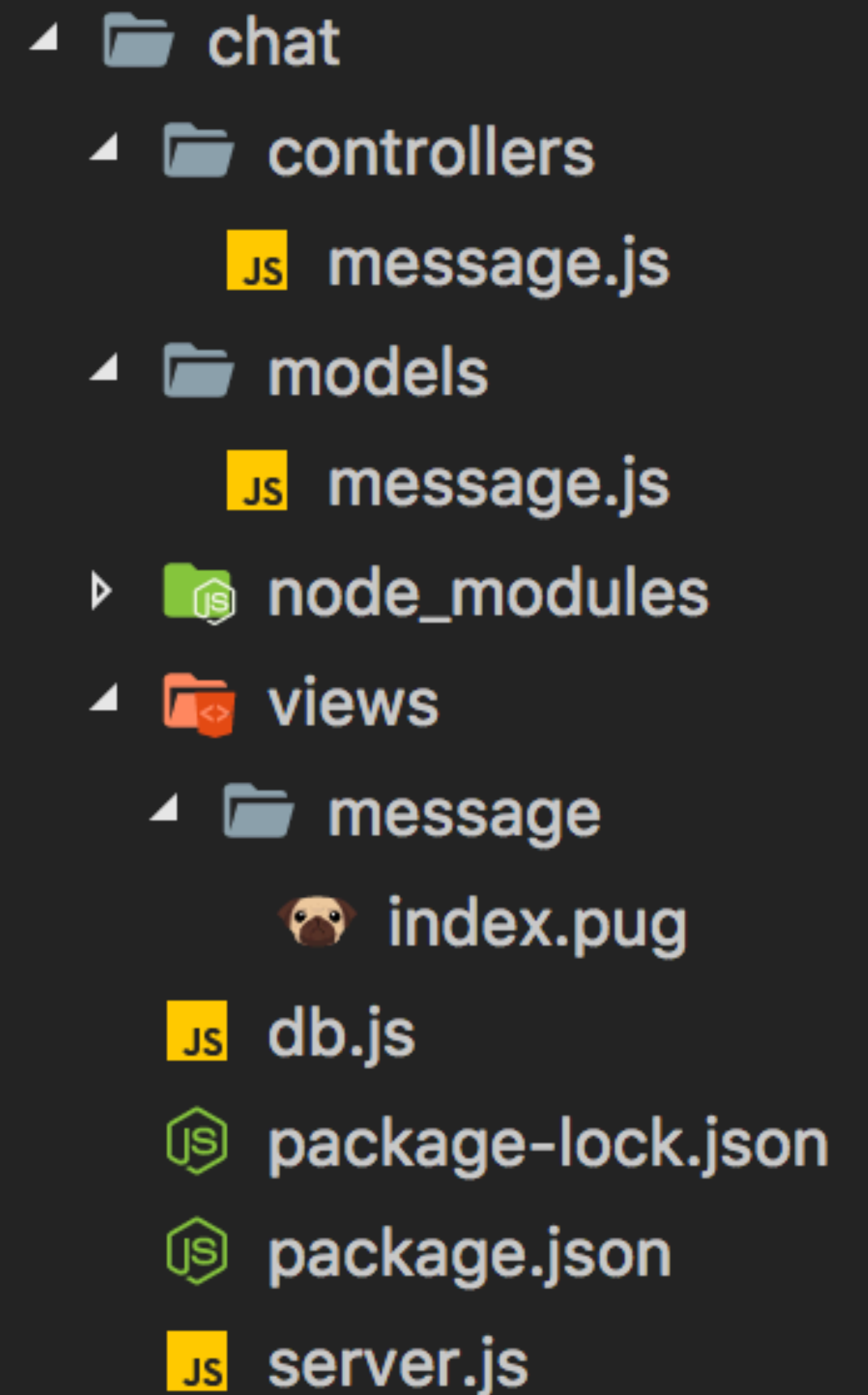
Pour un soucis de lisibilité, il est bien de prendre l'habitude de séparer notre code en différents fichiers.

Ci-contre, un exemple d'architecture pour cette application très basique.

models : Ce qui gère les données

views : Les templates (qui seront "render" en html)

controllers : En fonction de la requête utilisateur, génère une réponse



```
chat
├── controllers
│   └── message.js
├── models
│   └── message.js
├── node_modules
├── views
│   └── message
│       ├── index.pug
│       ├── db.js
│       ├── package-lock.json
│       ├── package.json
│       └── server.js
```


TP : Chat (Guide)

Exemple d'implémentation de `server.js`

```
const http = require('http')
const messageController = require('./controllers/message')

http.createServer((req, res) => {
  // On redirige vers la bonne action en fonction de l'url et de la
  // méthode HTTP
  if(req.url === '/messages') {
    if (req.method === 'GET') {
      return messageController.index(req, res)
    }

    if(req.method === 'POST') {
      return messageController.create(req, res)
    }
  }

  // Dans tous les autres cas, on redirige vers /messages
  res.writeHead(302, {'Location': '/messages'})
  res.end()
}).listen(8081)
```

Dans cet exemple, on vérifie l'URL et on appelle des actions qui sont dans `controllers/messages.js`

GET et POST sont des verbes HTTP, un navigateur émet habituellement ces requêtes en GET, mais on peut définir un envoi du formulaire en POST.

La nomenclature suivante est appelée REST:

- * GET /messages -> index (liste tous les messages)
- * GET /messages/:id -> show (récupère un message)
- * POST /messages -> create (créé un message)
- * PUT /messages/:id -> replace un message
- * PATCH / messages/:id -> modifie des champs d'un message
- * DELETE /messages/:id -> supprime un message

TP : Chat (Guide)

db.js

```
const Sequelize = require('sequelize')  
  
const db = new Sequelize(...)  
  
module.exports = db
```

Compléter ce fichier, grâce à la doc de Sequelize : <http://docs.sequelizejs.com/manual/installation/getting-started.html#setting-up-a-connection>

Ce fichier permet d'initialiser la connection à la base de donnée.

En CommonJS grâce à `module.exports` et `require`, le `require` met en cache le fichier, c'est à dire qu'à chaque fois qu'on fera un `require` sur ce fichier, c'est toujours la même instance de `db` qu'on retrouvera.

TP : Chat (Guide)

models/message.js

```
const db = require('../db')

const Message = db.define('message', {
  ...
})

module.exports = Message
```

Compléter ce fichier grâce à la documentation de Sequelize : <http://docs.sequelizejs.com/manual/tutorial/models-definition.html>

Il permet de définir la forme d'un Message en base de donnée.

Attention, il faudra créer la table dans la base de donnée en appelant avant la première utilisation du model : **Message.sync()**

TP : Chat (Guide)

controllers/message.js

```
const Message = require('../models/message')

function index(req, res) {
  res.write('Index action -> à compléter')
  res.end()
}

function create(req, res) {
  res.write('Create action -> à compléter')
  res.end()
}

module.exports = { create, index }
```

Compléter ce fichier :

La fonction **index** doit permettre d'afficher la vue de liste de message (**views/message/index.pug**) avec en paramètre la liste de message

La fonction **create** doit permettre de rajouter un message depuis les données d'un formulaire et rediriger vers la liste des messages

Message est le model, et permet grâce à **Sequelize** de récupérer des messages depuis une base de donnée

Félicitations !!

Cours WIK-NJS-301 burned :)