

TYPE DE FLUX : WRITABLE IMPLEMENTOR

```
const { Writable } = require('stream')
const chalk = require('chalk')
const util = require('util')

// Création de stream "à l'ancienne" (avant ES2015)
function RedLogger(opt) {
  Writable.call(this, opt)
}

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

util.inherits(RedLogger, Writable)

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



TYPE DE FLUX : DUPLEX

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



TYPE DE FLUX : TRANSFORM

- Extension du concept de Duplex
- Prévu pour réaliser des transformations sur les données
- Exemples : zlib, crypto
- Implementor : permet d'éviter d'écrire les méthodes `_read` et `_write` au profit de `_transform` et `_flush`.
- Très utilisée, on a l'habitude de passer par la bibliothèque **`through2`** mais depuis Node.js v1.2.0 la syntaxe simplifiée rend son usage moins justifié.



TYPE DE FLUX : TRANSFORM IMPLEMENTOR

```
const childProcess = require('child_process')
const { Transform } = require('stream')

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

const transform = new Transform({
  transform (chunk, enc, done) {
    done(null, chunk.toString().toUpperCase())
  }
})

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



PIPELINING

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

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

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



OBJECT MODE

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



BIBLIOTHÈQUES

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



LIBRAIRIES : THROUGH2 MAP, FILTER, REDUCE & SPY

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

```
const colors = require('colors')
const map = require('through2-map')

const rainbow = map(chunk => {
  return chunk.toString().rainbow
})

process.stdin.pipe(rainbow).pipe(process.stdout)
```



LIBRAIRIES : CONCAT, DUPLEXER, TRUMPET, COMBINER

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



GULP



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







Lab 8



LIAISON AVEC LA PERSISTANCE DES DONNÉES



PLAN



LIAISON AVEC LA PERSISTANCE DES DONNÉES

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





DRIVERS

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



DRIVERS : EXEMPLE AVEC ORACLE

- Il faut définir des variables d'environnement

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

- Configuration du driver

```
const oracle = require('oracledb');  
  
const connectData = {  
  connectString: 'localhost:1521/sid',  
  user: 'oracle',  
  password: 'oracle'  
}
```



DRIVERS : EXEMPLE AVEC ORACLE

```
(async () => {
  let connection

  try {
    connection = await oracle.getConnection(connectData)

    const result = await connection.execute('SELECT systimestamp FROM dual')

    console.log(result)

  } catch (err) {
    console.error('Error with the db:', err)

  } finally {
    if (connection) {
      try {
        await connection.close()
      } catch (err) {
        console.error(err)
      }
    }
  }
})()
```



DRIVERS : EXEMPLE AVEC POSTGRESQL

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

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

client.connect(err => {
  if (err) throw err

  client.query('SELECT NOW() AS "theTime"', (err, result) => {
    if (err) { throw err }

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



DRIVERS : EXEMPLE AVEC MYSQL

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

const connection = MySQL.createConnection({
  host      : 'localhost',
  user      : 'root',
  password : 'root'
})

connection.connect()

connection.query('SELECT 1 + 1 AS solution',
  (err, rows, fields) => {
    if (err) { throw err }

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

connection.end()
```



DRIVERS : MONGODB

```
const { MongoClient } = require('mongodb')

// Connect to the db
MongoClient.connect('mongodb://url', (err, client) => {
  if (err) { return console.error(err) }

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

  collection.insert(docs, {w:1}, (err, result) => {
    console.log(result)

    client.close();
  });
});
```



DRIVERS : ABSTRACTIONS SQL

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



ORM SQL

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



SEQUELIZE

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

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

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

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



SEQUELIZE : MODEL

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



SEQUELIZE : SYNC, LECTURE, ÉCRITURE

```
sequelize
  .sync({ force: true })
  .complete(err => {
    if (err) throw err
```

User

```
  .create({
    username: 'john',
    password: '1111'
  })
  .complete((err, user1) => {
    if (err) throw err
```

User

```
  .find({ username: 'john' })
  .complete((err, user2) => {
    if (err) throw err
    console.log(user1.values, user2.values)
  })
})
})
```



SEQUELIZE : SEQUELIZE-AUTO

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

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

- Ces fichiers peuvent être chargés par la fonction import

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



MONGOOSE

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

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

- Définition du mapping des objets

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



MONGOOSE : MAPPING

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



MONGOOSE : API

- Créer des objets et les sauvegarder

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

- Requêtes

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

```
User.find((err, users) => {
  if (err) throw err
  console.log('users', users)
})
```







Lab 9



OUTILLAGE ET USINE LOGICIELLE



PLAN



ÉCRIRE UNE APPLICATION NODE.JS

De nombreux outils existent pour développer du JavaScript :

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



DÉBOGAGE D'UNE APPLICATION NODE.JS

- Déboguer est une nécessité, cela permet :
 - de comprendre pourquoi le code n'a pas le comportement attendu
 - d'inspecter les différentes variables présentes dans le code JS à un instant donné
- Pour cela, il existe plusieurs méthodes :
 - Utilisation des logs
 - Utilisation d'un IDE (c.f. documentation de l'IDE)
 - Utilisation du débogueur de Node.js
 - Utilisation de Node Inspector



DÉBOGUER AVEC LES LOGS

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

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

Sortie :

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

Cas d'utilisation :

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



DÉBOGUER AVEC LE DÉBOGUEUR NODE.JS

Embarqué nativement dans Node.js

```
| node debug server.js
```

Le code utilisé est le suivant :

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

Les commandes disponibles sont :

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



DÉBOGUER AVEC LE DÉBOGUEUR NODE.JS : UNE SÉQUENCE

Déroulement d'un débogage :

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



DÉBOGUER AVEC LE DÉBOGUEUR NODE.JS : UNE SÉQUENCE

Déroulement d'un débogage (suite) :

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



DÉBOGUER AVEC LE DÉBOGUEUR NODE.JS

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



DÉBOGUER AVEC NODE INSPECTOR

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



TESTS

- Node.js dispose de très bons outils de test
- La nature souple du langage permet des notations élégantes
- Un des frameworks les plus populaires est Jest
 - Fournit une API simple et efficace
 - Il fonctionne également côté navigateur



ILLUSTRATION

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

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



ILLUSTRATION

```
describe('Test using Jest', () => {
  let rh
  beforeEach(() => {
    // Avant chaque test : réinitialisation du RequestHandler
    rh = new RequestHandler()
  })

  // Premier test avec une syntaxe lisible
  it('should handle the request and return a number between 0 and 1', () => {
    const result = rh.handleRequest()
    expect(result).toBeGreaterThanOrEqual(0)
    expect(result).toBeLessThan(1)
  })
}

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



ILLUSTRATION

Utilisation des spies :

```
describe('Test using spies', () => {
  let rh
  beforeEach(function() {
    rh = new RequestHandler()
  })

  it('should call the onSuccess function once', () => {
    let spy = jest.spyOn(rh, 'onSuccess')

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

    // Check we called it once
    expect(spy).toHaveBeenCalledTimes(1)
  })
})

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



ILLUSTRATION

Utilisation des stubs :

```
describe('Test using stubs', function() {
  let rh
  beforeEach(function() {
    rh = new RequestHandler()
  })

  it('should return one through mock request handler', () => {
    rh.onSuccess = jest.fn().mockReturnValue(1)

    let result = rh.handleRequest()

    // Assert that onSuccess returns 1
    expect(result).toEqual(1)

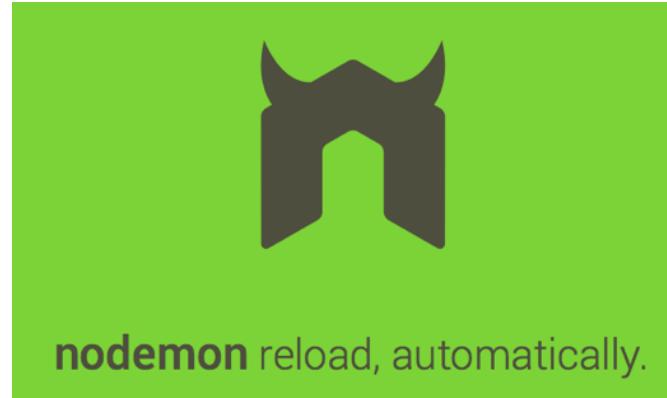
    // Assert that the method is called once
    expect(rh.onSuccess).toHaveBeenCalledTimes(1)
  })
})
```



AUTRES OUTILS DE TEST



NODEMON



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

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

```
# npx nodemon script
```





BUILD

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



AUTOMATISATION DE LA PRODUCTION DES LIVRABLES

Avec Node.js nous pouvons désormais :

- Tester nos applications
- Automatiser le lancement des tests lors de la production de livrables
- La prochaine étape serait donc de :
 - Builder automatiquement les livrables et lancer les tests à chaque nouvelle version sur le VCS.
Une solution bien connue dans le monde Java pour cette problématique s'appelle Jenkins.



INTÉGRATION CONTINUE

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

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



INTÉGRATION AVEC JENKINS

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

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



INTÉGRATION AVEC JENKINS

Configuration d'une tâche Grunt qui sera appelée par Jenkins

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

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



INTÉGRATION AVEC JENKINS

Création du job Jenkins :

- Créer un nouveau projet free-style
- Configurer la récupération des sources par CVS, Git ou SVN
- Ajouter une étape de build sous forme de script shell
 - | npm install
 - | grunt jenkins
- Ajouter une étape post-build **Publier le rapport des résultats des tests JUnit**
 - La valeur XML des rapports de tests correspond à celle du Gruntfile : **report/test-reports.xml**
- Sauvegarder le job et le lancer.



INTÉGRATION AVEC JENKINS

Grâce à Jenkins, il est possible de

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







Lab 10



NODE.JS EN MODE CLUSTER



PLAN



NODE.JS EN CLUSTER

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



MODULE CLUSTER : CONTEXTE

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



MODULE CLUSTER : OBJECTIF

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



MASTER & WORKERS

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



MASTER & WORKERS : API

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



MODULE CLUSTER : EXEMPLE

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

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

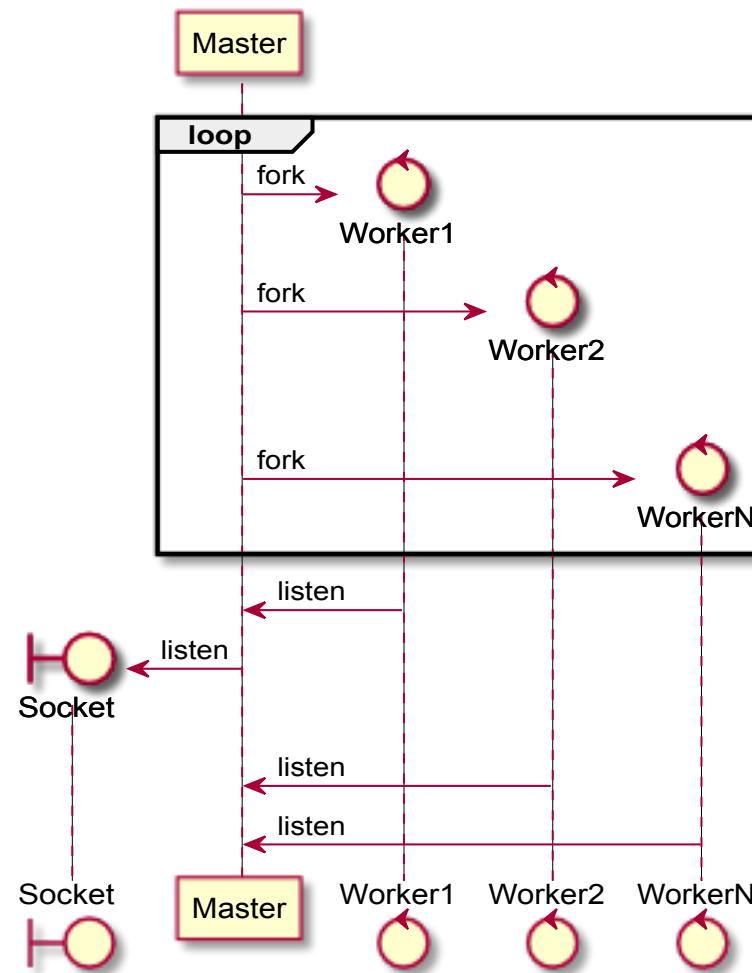


CYCLE DE VIE

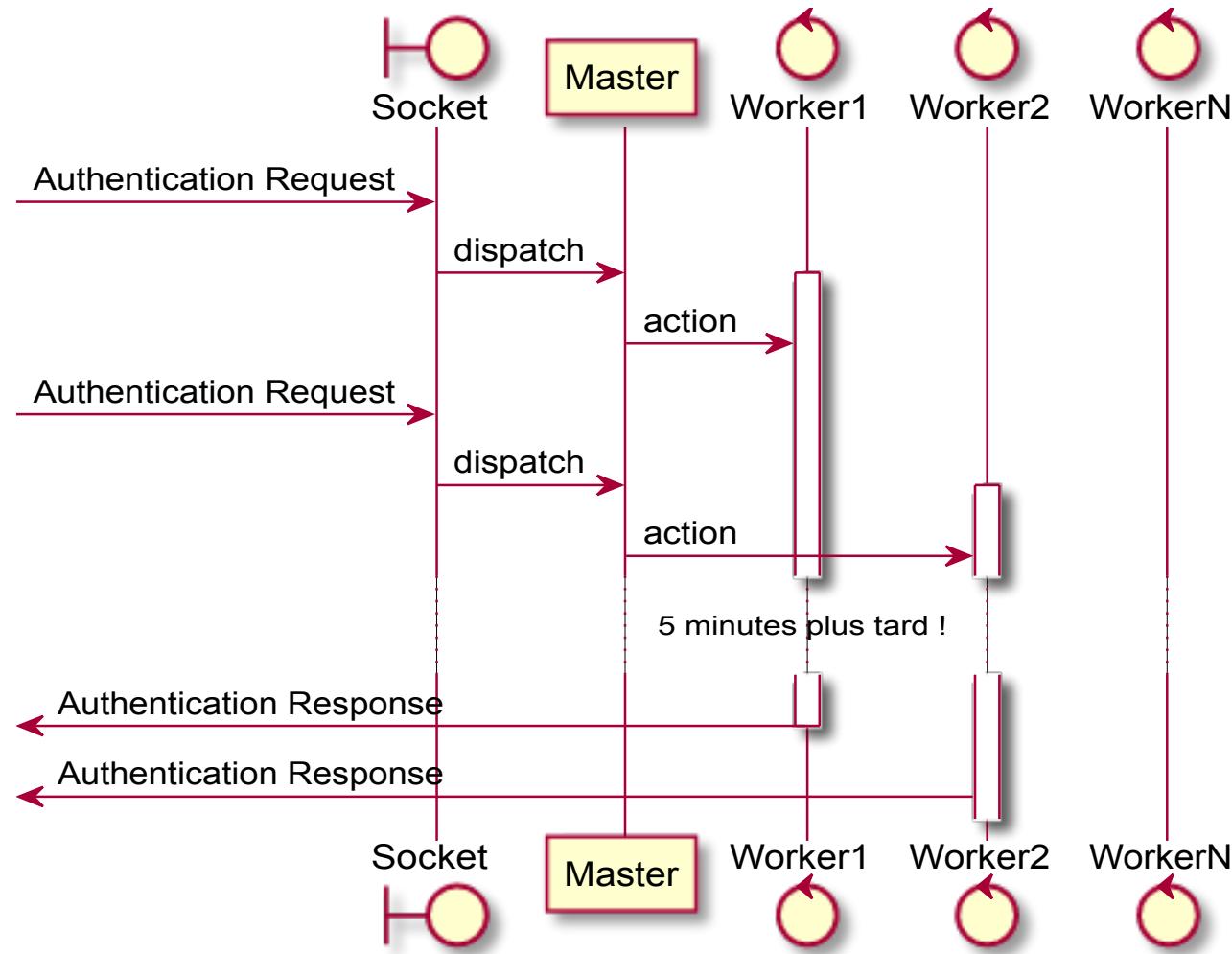
- Le worker peut s'appuyer sur des évènements
 - `setup`, `fork`, `online`, `listening`, ...
- L'objet worker fournit des informations
 - `worker.id` : contient un id unique du process
C'est la clef du worker dans workers
 - `worker.process` : le processus du worker
 - `worker.exitedAfterDisconnect`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



CYCLE DE VIE : DÉMARRAGE



PARTAGE DE CONNEXION





MESSAGING

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



MESSAGING : API

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



MESSAGING : EXEMPLE

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

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

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

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

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

Object.keys(cluster.workers).map(id => cluster.workers[id])
  .forEach(worker => {
    worker.on('message', data => {
      console.log('Master received message from worker',
        worker.id, ':', data);
    });
    worker.send('Hi!');
  });
}
```



MESSAGING : EXEMPLE

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

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

console.log(`Hello I'm the worker ${worker.id}`);

process.on('message', (data) => {
  process.send(`#${data} from ${worker.id}`);
});

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



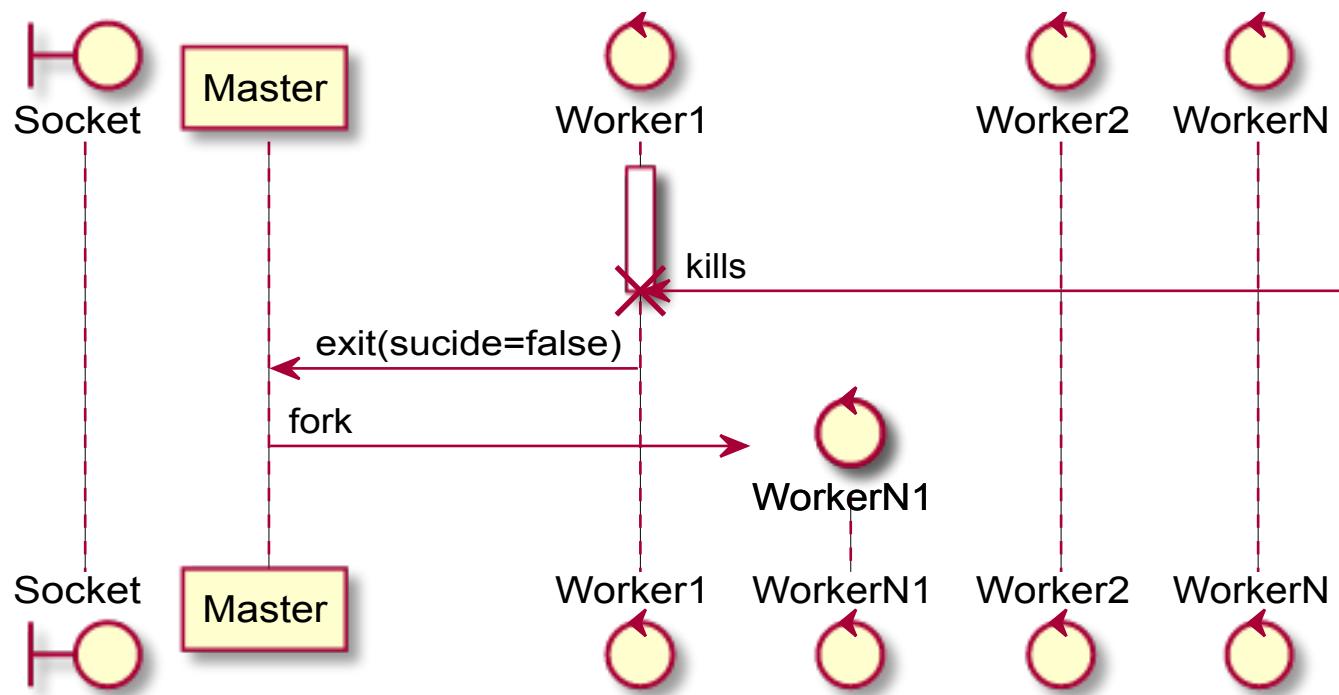
MESSAGING : APPLICATIF

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



GESTION DES ERREURS

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



GESTION DES ERREURS : EXEMPLE

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

/* ... */

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







Lab 11



AU DELÀ DE NODEJS



PLAN



MEAN / MERN



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



MEAN / MERN

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



MEAN / MERN

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





CLOUD

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



HEROKU

- Presque aussi simple qu'un `git push`
- Créer un fichier `Procfile`
 - | `web: node index.js`
- Le port HTTP se trouve dans les variables d'environnement
 - | `const port = process.env.PORT || 3000;`
- Ajouter Heroku comme remote dans git
 - | `heroku git:remote -a identifiant-git-heroku`
- ***Pusher*** sur Heroku
 - | `git push heroku master`



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



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



- Installer l'environnement sur la machine

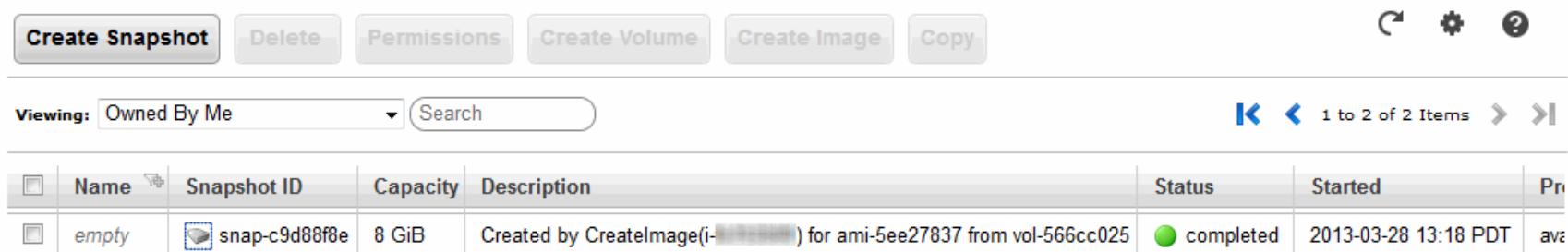
```
$ sudo yum install gcc-c++ make  
$ sudo yum install openssl-devel  
$ sudo yum install git  
$ git clone git://github.com/joyent/node.git  
$ cd node
```

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

```
$ git checkout v12.17.0 # ou une autre version  
$ ./configure  
$ make  
$ sudo make install
```



- Sauvegarder votre instance comme modèle (AMI)



The screenshot shows the AWS Management Console interface for managing snapshots. At the top, there are several buttons: 'Create Snapshot' (highlighted in blue), 'Delete', 'Permissions', 'Create Volume', 'Create Image', and 'Copy'. To the right of these are navigation icons: a left arrow, a gear icon, and a question mark icon. Below the buttons, there is a dropdown menu labeled 'Viewing: Owned By Me' and a search bar. On the far right, there are more navigation icons: a left arrow, a double-left arrow, a double-right arrow, and a right arrow. The main area displays a table of snapshots:

	Name	Snapshot ID	Capacity	Description	Status	Started	Pr...
<input type="checkbox"/>	empty	 snap-c9d88f8e	8 GiB	Created by CreateImage(i-...) for ami-5ee27837 from vol-566cc025	completed	2013-03-28 13:18 PDT	ava...

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



MONITORING

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



MONITORING : PM2

- L'installation se fait par la commande suivante :
| npm install pm2 -g
- **ping** : vérifie la présence du master et peut le relancer
- **web** : démarre un serveur web accessible sur le port 9615
- **start script** : démarre l'application en mode cluster monitoré
- **stop script** : arrête l'application
- **monit** : présente un écran de surveillance des process



MONITORING : NEWRELIC

- Il faut passer par la création d'un compte (<http://newrelic.com/>)
- Installer le module NewRelic
 - | `npm install newrelic`
- Copier `newrelic.js` du dossier `node_modules/newrelic` dans le dossier racine de l'application
- Editer le fichier `newrelic.js` pour remplacer la clef de licence
 - | `license_key : 'license key here'`,
- rajouter la commande suivante en première ligne de l'application
 - | `require('newrelic');`
- Lancer l'application puis aller sur le site de NewRelic pour **monitorer l'application**



