

# PACKAGE.JSON : FONCTIONNEMENT

- `engines` : définit les moteurs et leur version

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

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

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

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



# MODULES : PATTERN

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



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



# MODULES

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



# MODULES : REQUIRE

- Modules core : fournis par Node.js, ils sont toujours accessibles
  - | `const fs = require('fs');`
- Modules npm : sont accessibles par leur nom s'ils se trouvent dans le répertoire `./node_modules`
  - | `const yargs = require('yargs');`
- N'importe quel fichier `.js` : tout fichier `.js` est un module.
  - | `const submodule = require('./dir/submodule.js');`
    - Il existe différentes astuces (ex : le `.js` peut être omis)
    - C'est cette fonctionnalité qu'on utilise pour découper un programme Node.js en plusieurs fichiers.



# MODULES : REQUIRE

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



# MODULES : EXPORTS

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

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

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

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



# MODULES : EXPORTS

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



# MODULES : EXEMPLE

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

fs.readFile('./lorem.txt', 'utf8', (error, data) => {
  const upperCased = upperCase(data)
  console.log(chalk.blue(upperCased))
})

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

//ou exports.upperCase = function...
```



# CORE MODULES - LES MODULES INCLUS DE NODEJS

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



# MODULES NOYAUTS

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



# MODULES NOYAUX

- Les autres modules noyaux sont à charger avec `require`
- `os` : permet d'avoir des informations sur le système d'exploitation sur lequel s'exécute le processus
- `path` : outils de manipulation des chemins d'accès aux fichiers
- `util` : méthodes utilitaires (`format`, `is*`, `inspect...`)
- `fs` : accès au système de fichiers
  - `readFile`, `writeFile` : lire et écrire dans un fichier
  - `rename`, `chmod`, `rmdir`, ... : manipulation de fichiers
- `net` : fournit des APIs pour l'accès au réseau
- Et bien d'autres : `http`, `http2`, `https`, `dns`, `child_process...`



# GESTION DES DÉPENDANCES

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

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



# PUBLIER UN MODULE NPM

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







# Lab 4



# NODE ET LE WEB : HTTP, CONNECT & EXPRESS



# PLAN



# NODE.JS ET LE WEB

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

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

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('hello world');
})

server.listen(1234);
```



# MODULE HTTP

- Node.js propose un module noyau **HTTP** (et aussi **HTTP/2** et **HTTPS**)
- Il fournit un certain nombre d'APIs de bas niveau :
  - Pour communiquer avec un Serveur HTTP :  
`| const request = http.request(options, callback);`
  - Pour créer un serveur HTTP :  
`| const server = http.createServer(requestListener).listen(port);`
- Il supporte tout le protocole HTTP/1.1
  - Le cache : ETag, If-Modified-Since
  - Réseau: Connexions persistantes, Pipelining, Chunk Transfer Encoding...
  - Un nouveau verbe : UPGRADE



# MODULE HTTP : CLIENT

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

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

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

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



# MODULE HTTP : CLIENT

envoi

envoyé

Code HTTP : 200

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

Contenu : {"db\_name": "registry", "doc\_count": 75380, ...}



# MODULE HTTP : CLIENT

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



# MODULE HTTP : SERVEUR

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



# MODULE HTTP : SERVEUR

## 1. Charger le module HTTP

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

## 2. Créer un serveur HTTP

```
| const server = http.createServer()
```

## 3. Définir un callback pour traiter les requêtes

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

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

```
| server.listen(port)
```



# MODULE HTTP : SERVEUR

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



# SERVEUR WEB MONO-THREADÉ

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



# MODULE HTTP : LIMITES

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



# CONNECT

- Le module **Connect**

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

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



# CONNECT : LES MIDDLEWARES

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



# CONNECT : EXEMPLE

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

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



# EXPRESS

- Le module **Express**

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

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

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

app.listen(3000)
```



# EXPRESS : GENERATOR

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

```
$ express -h
```

```
Usage: express [options] [dir]
```

```
Options:
```

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



# EXPRESS : STRUCTURE

- La structure d'un serveur Express ressemble à cela

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

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

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

app.listen(8080)
```



# EXPRESS : ROUTER

- Il existe de nombreuses façons de réagir aux requêtes

- Sous forme de middleware

```
| app.use((req, res, next) => {});
```

- Avec la méthode HTTP et l'URL

```
| app.get('/resource', (req, res) => {});
```

- Répondre à toutes les méthodes

```
| app.all('/resource', (req, res) => {});
```

- Capturer des paramètres dans l'URL

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



# EXPRESS : LA REQUÊTE

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



# EXPRESS : LA RÉPONSE

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



# EXPRESS : TEMPLATES

- Express propose l'intégration d'un moteur de templates

1. Définir le dossier contenant les templates

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

2. Choisir le moteur de templates

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

3. Créer le template

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

4. L'utiliser dans une route

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



# EXPRESS : TEMPLATES

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



# EXPRESS : TEMPLATES

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

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



# ALLER PLUS LOIN

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



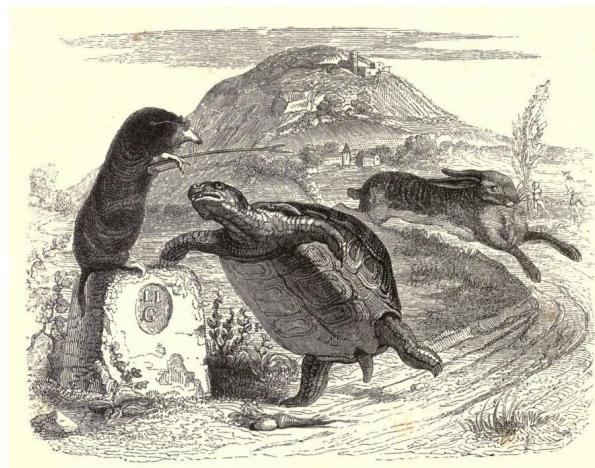




# Lab 5



# L'ASYNCHRONE EN DÉTAILS



# PLAN



# ASYNCHRONISME

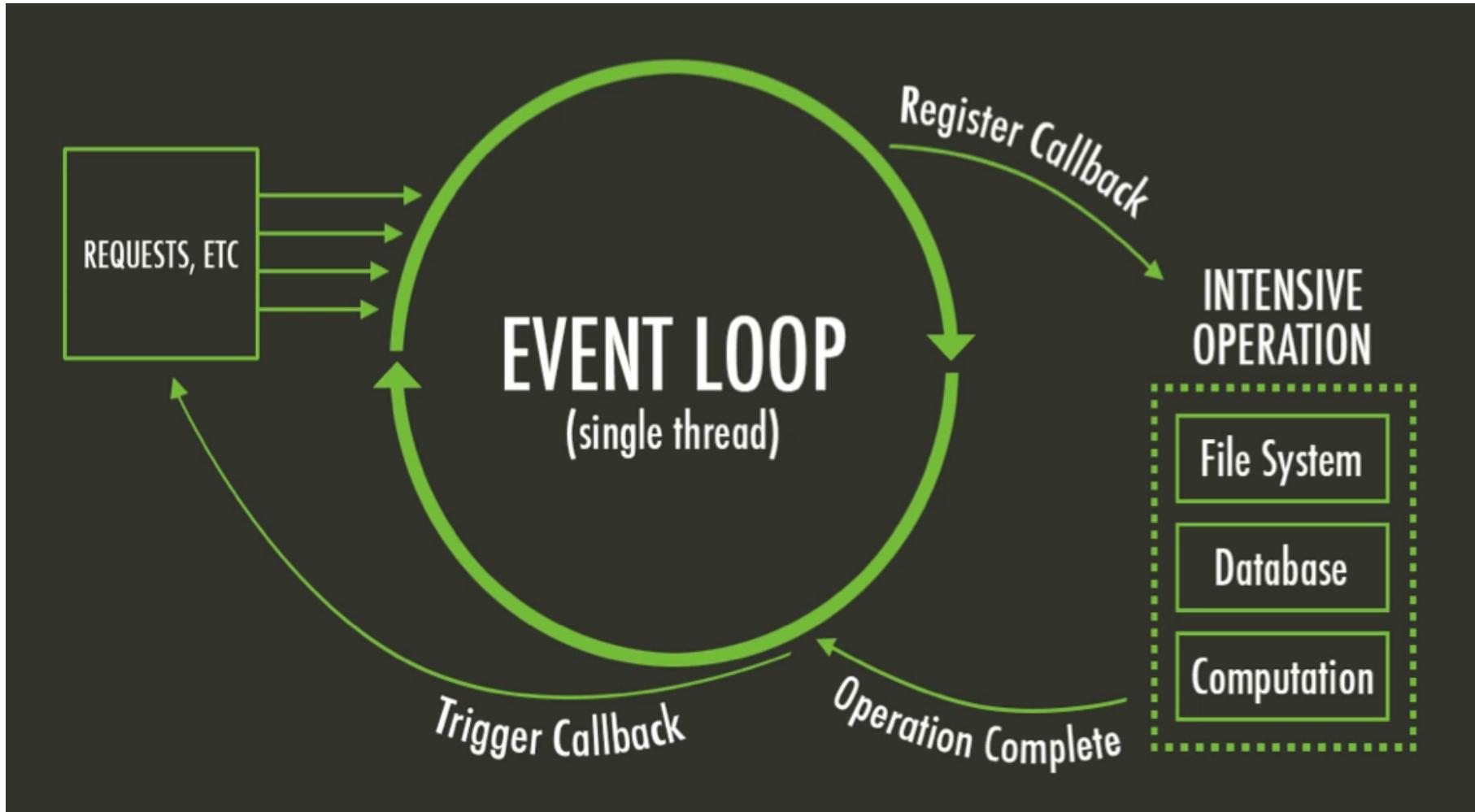
- Node.js est **monothreadé** mais **non bloquant pour les entrées/sorties (I/O)** comme la lecture d'un fichier, le requêtage en base de données ou la réponse à une requête HTTP.

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

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



# ASYNCHRONISME : EVENT LOOP



# CALLBACK HELL OU PYRAMID OF DOOM

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

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

myCallbackHell({ firstname: 'Georges' }, (err, result) => {
    console.log(result);
});
```



# CALLBACK HELL OU PYRAMID OF DOOM

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



# GESTION DE L'ASYNCHRONISME

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



# LES PROMESSES

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



# LES PROMESSES : LES IMPLÉMENTATIONS



# UTILISER UNE PROMESSE

- Pour utiliser une API utilisant les promesses

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

```

- Ou en version complète avec gestion des erreurs

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

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

```



# CHAÎNAGE DES PROMESSES

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

```
readFileWithPromise('file1.txt')
  .then(data => {
    // data vaut le contenu de file1
    return readFileWithPromise('file2.txt');
  })
  .then(data => {
    // data vaut le contenu de file2
    return readFileWithPromise('file3.txt');
  })
  .then(data => {
    // data vaut le contenu de file3
  })
  .catch(err => {
    console.log('Erreur', err);
  })
}
```



# PARALLÈLE ET PROMESSES

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

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



# NODE.JS ET LES PROMESSES

- L'API de Node.js ne propose pas de retourner une promesse directement
- Il faut encapsuler une fonction avec callback
- Méthode `promisify` du module core `util` apparu en v8.0.0
- propriété `promises` du module core `fs` apparu en v10.0.0
- Pour des versions de Node.js plus anciennes, des librairies existent, exemple : `denodeify`



# NODE.JS ET LES PROMESSES

- transformer un API avec callback en promesse

```
const util = require('util')
const action = util.promisify(module.action)

action(param1, ..., param[n])
  .then(result => { /* ... */ })

/* Avec Node 8.0.0 */
const fs = require('fs')
const util = require('util')

const readFileWithPromise = util.promisify(fs.readFile)

readFileWithPromise('file1.txt')
  .then(fileContent => console.log('file content = ', fileContent))

/* Avec Node 10.0.0 */
const { readFile } = require('fs').promises

readFile('file1.txt')
  .then(console.log)
```



# CRÉER UNE PROMESSE

- Un callback classique

```
function funcCallback (param1, ..., paramn, callback) {  
  /* ... */  
  if (condition) {  
    return callback(new Error('example error'));  
  }  
  
  callback(null, result);  
}
```

- Une promesse en détail

```
function funcPromise (param1, ..., paramn) {  
  return new Promise((resolve, reject) => {  
    /* ... */  
    if (condition) {  
      return reject(new Error('example error'));  
    }  
  
    resolve(result);  
  });  
}
```



# CRÉER UNE PROMESSE

- Une promesse un peu plus idiomatique

```
function funcPromise(param1, ..., paramn) {  
    return Promise.resolve().then(() => {  
        /* ... */  
        if (condition) {  
            throw new Error('example error')  
        }  
  
        return results  
    });  
}
```

- On verra une méthode encore plus simple avec `async/await` ci-après



# TRAITEMENT DES ERREURS ASYNCHRONES

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

```
try {
  // Emission d'un erreur asynchrone
  setTimeout(() => {
    throw new Error('example error');
  }, 0);
} catch (error) {
  console.log('Une erreur a été interceptée:', error.message);
}
```

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



# TRAITEMENT DES ERREURS ASYNCHRONES : PROMESSES

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

```
aPromise.then(  
  (resolvedValue) => { /* ... */ },  
  (error) => { /* ... */ }  
)
```

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

```
aPromise.catch((error) => { /* ... */ });
```

- Avec **finally()** (optionnel, à partir de la v10.0.0) execute une action quelque soit l'état final de la promesse.



# ASYNC / AWAIT

- Depuis ES2017 (Node.js v7.6.0), il est possible d'avoir une écriture plus proche des appels synchrones avec les Promises grâce à la syntaxe `async/await`

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

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

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







# Lab 6



# COMMUNICATION TEMPS RÉEL



# PLAN



# WEB TEMPS RÉEL

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



# WEB TEMPS RÉEL

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



# WEB TEMPS RÉEL : WEBSOCKETS

- Spécification normalisée par l'IETF (Internet Engineering Task Force) dans le RFC 6455 et fait partie d'**HTML5**
- <https://html.spec.whatwg.org/multipage/web-sockets.html>
- Principe
  - Connexion permanente entre le client et serveur
  - Possibilité de communication dans les deux sens
- Le protocole est constitué de deux étapes
  - Handshake
  - Transfert de données



# SOCKET.IO



- C'est un framework pour faire des applications temps réel
- Il propose une API très simple
  - Pour établir la connexion avec les clients
  - Transmettre des messages dans les deux sens
- Il existe depuis très longtemps (V0.1 en 2010, V1 en 2014, V2 en 2017)
- Mis en cause pour ses performances
  - Mais seulement dans des contextes TRÈS sollicités



# SOCKET.IO : TRANSPORTS

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



# SOCKET.IO : API

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



# SOCKET.IO : API SERVEUR

- Connection d'un client

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

- Émission d'un message à tous les clients

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

- Émission d'un message

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

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

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

- Écoute de la réception d'un message

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



# SOCKET.IO : API CLIENTE

- Connexion au serveur

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

- Écoute de la réception d'un message

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

- Émission d'un message

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

- Attention à ne pas mélanger les noms des messages

- Qui vont du serveur vers le client

- Qui vont du client vers le serveur

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



# ► SOCKET.IO : EXEMPLE

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



# ► SOCKET.IO : EXEMPLE CÔTÉ SERVEUR

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

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

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

io.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' })
  socket.on('my other event', function (data) {
    console.log(data)
  })
})
server.listen(8000)
```



# SOCKET.IO : EXEMPLE CÔTÉ CLIENT

```
<!DOCTYPE html>
<title>Socket</title>

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







# Lab 7



# LA GESTION DES STREAMS



# PLAN



# DÉFINITION

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

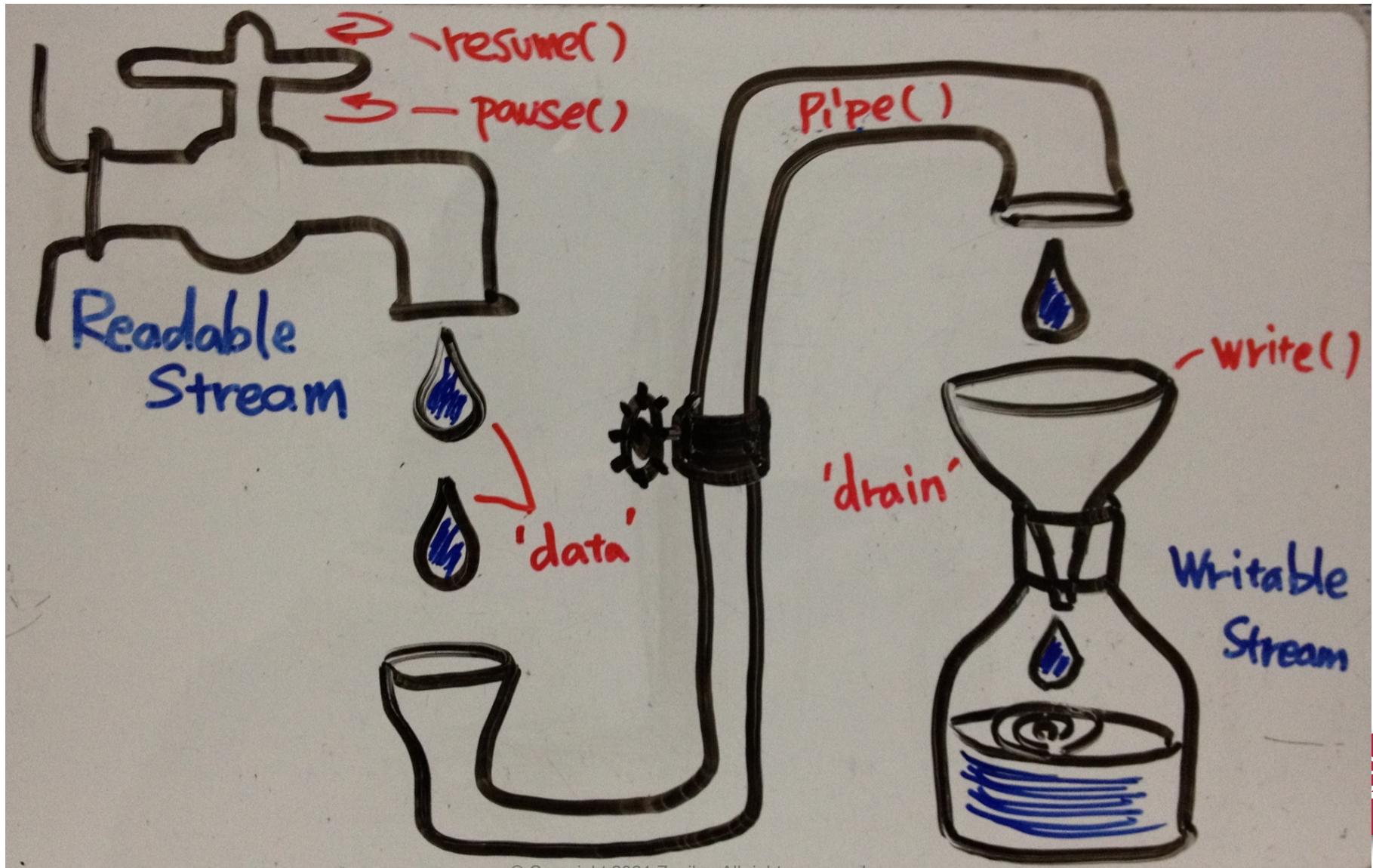


# DÉFINITION

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



# DÉFINITION : EN UNE IMAGE



# POURQUOI ?

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



# STREAMS : UN EXEMPLE

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

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

const transform = through2((chunk, enc, done) => {
  this.push(chunk.toString().toUpperCase())
  done()
})

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



# BUFFERS

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



# TYPE DE FLUX

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



# TYPE DE FLUX : READABLE

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



# TYPE DE FLUX : READABLE EXEMPLE

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

const readable = fs.createReadStream('../images/stream-image.jpg')

readable.on('readable', () => {
  let chunk
  while (null !== (chunk = readable.read())) {
    console.log('got %d bytes of data', chunk.length)
  }
})

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



# TYPE DE FLUX : READABLE IMPLEMENTOR

```
const { Readable } = require('stream')

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

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

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



# TYPE DE FLUX : WRITABLE

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



# TYPE DE FLUX : WRITABLE EXEMPLE

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

const server = http.createServer((req, res) => {
  res.write('hello, ')
  res.end('world!')
})
```

```
server.listen(1234)
```

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

```
hello, world!
```

