eBook Gratuit

APPRENEZ Node.js

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#node.js

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: node-js

It is an unofficial and free Node.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Node.js.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Node.js

Remarques

Node.js est un framework d'E / S asynchrones, non bloquant et basé sur des événements, qui utilise le moteur JavaScript V8 de Google. Il est utilisé pour développer des applications faisant largement appel à la possibilité d'exécuter JavaScript à la fois sur le client et sur le serveur, ce qui permet de profiter de la réutilisation du code et de l'absence de changement de contexte. Il est open-source et multi-plateforme. Les applications Node.js sont écrites en JavaScript pur et peuvent être exécutées dans l'environnement Node.js sous Windows, Linux, etc.

Versions

Version	Date de sortie
v8.2.1	2017-07-20
v8.2.0	2017-07-19
v8.1.4	2017-07-11
v8.1.3	2017-06-29
v8.1.2	2017-06-15
v8.1.1	2017-06-13
v8.1.0	2017-06-08
v8.0.0	2017-05-30
v7.10.0	2017-05-02
v7.9.0	2017-04-11
v7.8.0	2017-03-29
v7.7.4	2017-03-21
v7.7.3	2017-03-14
v7.7.2	2017-03-08
v7.7.1	2017-03-02
v7.7.0	2017-02-28

Version	Date de sortie
v7.6.0	2017-02-21
v7.5.0	2017-01-31
v7.4.0	2017-01-04
v7.3.0	2016-12-20
v7.2.1	2016-12-06
v7.2.0	2016-11-22
v7.1.0	2016-11-08
v7.0.0	2016-10-25
v6.11.0	2017-06-06
v6.10.3	2017-05-02
v6.10.2	2017-04-04
v6.10.1	2017-03-21
v6.10.0	2017-02-21
v6.9.5	2017-01-31
v6.9.4	2017-01-05
v6.9.3	2017-01-05
v6.9.2	2016-12-06
v6.9.1	2016-10-19
v6.9.0	2016-10-18
v6.8.1	2016-10-14
v6.8.0	2016-10-12
v6.7.0	2016-09-27
v6.6.0	2016-09-14
v6.5.0	2016-08-26
v6.4.0	2016-08-12

Version	Date de sortie
v6.3.1	2016-07-21
v6.3.0	2016-07-06
v6.2.2	2016-06-16
v6.2.1	2016-06-02
v6.2.0	2016-05-17
v6.1.0	2016-05-05
v6.0.0	2016-04-26
v5.12.0	2016-06-23
v5.11.1	2016-05-05
v5.11.0	2016-04-21
v5.10.1	2016-04-05
v5.10	2016-04-01
v5.9	2016-03-16
v5.8	2016-03-09
v5.7	2016-02-23
v5.6	2016-02-09
v5.5	2016-01-21
v5.4	2016-01-06
v5.3	2015-12-15
v5.2	2015-12-09
v5.1	2015-11-17
v5.0	2015-10-29
v4.4	2016-03-08
v4.3	2016-02-09
v4.2	2015-10-12

Version	Date de sortie
v4.1	2015-09-17
v4.0	2015-09-08
io.js v3.3	2015-09-02
io.js v3.2	2015-08-25
io.js v3.1	2015-08-19
io.js v3.0	2015-08-04
io.js v2.5	2015-07-28
io.js v2.4	2015-07-17
io.js v2.3	2015-06-13
io.js v2.2	2015-06-01
io.js v2.1	2015-05-24
io.js v2.0	2015-05-04
io.js v1.8	2015-04-21
io.js v1.7	2015-04-17
io.js v1.6	2015-03-20
io.js v1.5	2015-03-06
io.js v1.4	2015-02-27
io.js v1.3	2015-02-20
io.js v1.2	2015-02-11
io.js v1.1	2015-02-03
io.js v1.0	2015-01-14
v0.12	2016-02-09
v0.11	2013-03-28
v0.10	2013-03-11
v0.9	2012-07-20

Version	Date de sortie
v0.8	2012-06-22
v0.7	2012-01-17
v0.6	2011-11-04
v0.5	2011-08-26
v0.4	2011-08-26
v0.3	2011-08-26
v0.2	2011-08-26
v0.1	2011-08-26

Examples

Bonjour HTTP server HTTP

Tout d'abord, installez Node.js pour votre plate-forme.

Dans cet exemple, nous allons créer un serveur HTTP écoutant sur le port 1337, qui envoie `Hello, World!` au navigateur. Notez que, au lieu d'utiliser le port 1337, vous pouvez utiliser n'importe quel numéro de port de votre choix qui n'est actuellement utilisé par aucun autre service.

Le module `http` est un ***module de base*** Node.js (un module inclus dans la source de Node.js, qui n'exige pas l'installation de ressources supplémentaires). Le module `http` fournit la fonctionnalité permettant de créer un serveur HTTP à l'aide de la méthode `http.createServer()` . Pour créer l'application, créez un fichier contenant le code JavaScript suivant.

```
const http = require('http'); // Loads the http module

http.createServer((request, response) => {

    // 1. Tell the browser everything is OK (Status code 200), and the data is in plain text
    response.writeHead(200, {
        'Content-Type': 'text/plain'
    });

    // 2. Write the announced text to the body of the page
    response.write('Hello, World!\n');

    // 3. Tell the server that all of the response headers and body have been sent
    response.end();

}).listen(1337); // 4. Tells the server what port to be on
```

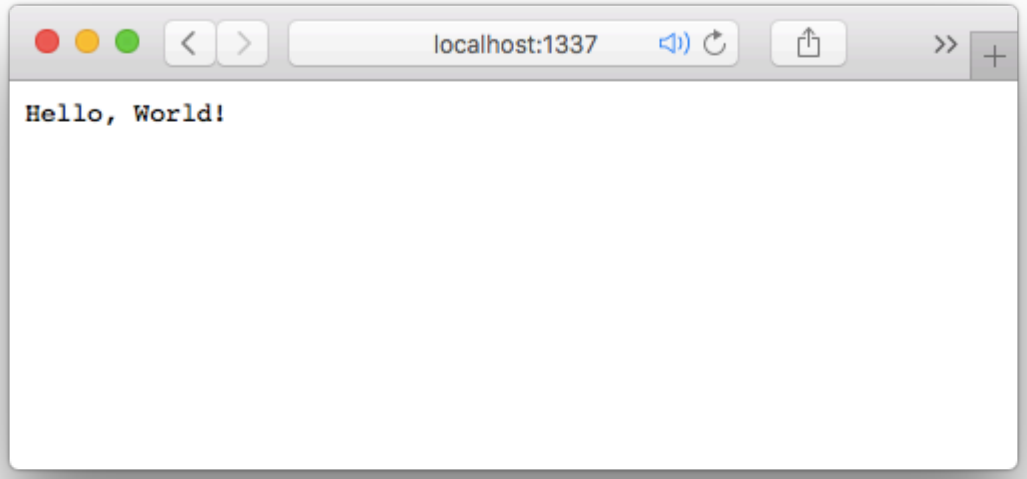
Enregistrez le fichier avec n'importe quel nom de fichier. Dans ce cas, si nous l' `hello.js` nous pouvons exécuter l'application en allant dans le répertoire où se trouve le fichier et en utilisant la

commande suivante:

```
node hello.js
```

Le serveur créé est alors accessible avec l'URL http: // localhost: 1337 ou http://127.0.0.1:1337 dans le navigateur.

Une simple page Web apparaîtra avec un texte «Hello, World!» En haut, comme le montre la capture d'écran ci-dessous.



Exemple en ligne modifiable

Ligne de commande Hello World

Node.js peut également être utilisé pour créer des utilitaires de ligne de commande. L'exemple ci-dessous lit le premier argument de la ligne de commande et imprime un message Hello.

Pour exécuter ce code sur un système Unix:

1. Créez un nouveau fichier et collez le code ci-dessous. Le nom de fichier n'est pas pertinent.
2. Rendre ce fichier exécutable avec `chmod 700 FILE_NAME`
3. Exécutez l'application avec `./APP_NAME David`

Sous Windows, faites l'étape 1 et exécutez-le avec le `node APP_NAME David`

```
#!/usr/bin/env node

'use strict';

/*
    The command line arguments are stored in the `process.argv` array,
    which has the following structure:
```

```
    [0] The path of the executable that started the Node.js process
    [1] The path to this application
    [2-n] the command line arguments

    Example: [ '/bin/node', '/path/to/yourscript', 'arg1', 'arg2', ... ]
    src: https://nodejs.org/api/process.html#process_process_argv
 */

// Store the first argument as username.
var username = process.argv[2];

// Check if the username hasn't been provided.
if (!username) {

    // Extract the filename
    var appName = process.argv[1].split(require('path').sep).pop();

    //  Give the user an example on how to use the app.
    console.error('Missing argument! Example: %s YOUR_NAME', appName);

    // Exit the app (success: 0, error: 1).
    // An error will stop the execution chain. For example:
    //   ./app.js && ls       -> won't execute ls
    //   ./app.js David && ls -> will execute ls
    process.exit(1);
}

// Print the message to the console.
console.log('Hello %s!', username);
```

Installation et exécution de Node.js

Pour commencer, installez Node.js sur votre ordinateur de développement.

Windows: Accédez à la page de téléchargement et téléchargez / exécutez le programme d'installation.

Mac: Accédez à la page de téléchargement et téléchargez / exécutez le programme d'installation. Alternativement, vous pouvez installer Node via Homebrew en utilisant le `brew install node` . Homebrew est un gestionnaire de paquets en ligne de commande pour Macintosh. Vous trouverez plus d'informations à ce sujet sur le site Web Homebrew .

Linux: Suivez les instructions de votre distribution sur la page d'installation de la ligne de commande .

Exécution d'un programme de noeud

Pour exécuter un programme Node.js, exécutez simplement `node app.js` ou `nodejs app.js` , où `app.js` est le nom de fichier du code source de votre application de noeud. Vous n'avez pas besoin d'inclure le suffixe `.js` pour que Node trouve le script que vous souhaitez exécuter.

Sous un système d'exploitation UNIX, un programme Node peut également être exécuté en tant que script de terminal. Pour ce faire, il doit commencer par un pointage vers l'interpréteur de

nœud, tel que le nœud `#!/usr/bin/env node` . Le fichier doit également être défini comme exécutable, ce qui peut être fait en utilisant `chmod` . Maintenant, le script peut être directement exécuté à partir de la ligne de commande.

Déployer votre application en ligne

Lorsque vous déployez votre application dans un environnement hébergé (spécifique à Node.js), cet environnement propose généralement une variable d'environnement `PORT` que vous pouvez utiliser pour exécuter votre serveur. Changer le numéro de port en `process.env.PORT` vous permet d'accéder à l'application.

Par exemple,

```
http.createServer(function(request, response) {
   // your server code
}).listen(process.env.PORT);
```

De plus, si vous souhaitez accéder à ce mode hors connexion pendant le débogage, vous pouvez utiliser ceci:

```
http.createServer(function(request, response) {
  // your server code
}).listen(process.env.PORT || 3000);
```

où `3000` est le numéro de port hors ligne.

Déboguer votre application NodeJS

Vous pouvez utiliser l'inspecteur de noeud. Exécutez cette commande pour l'installer via npm:

```
npm install -g node-inspector
```

Ensuite, vous pouvez déboguer votre application en utilisant

```
node-debug app.js
```

Le dépôt Github peut être trouvé ici: https://github.com/node-inspector/node-inspector

---

Déboguer en mode natif

Vous pouvez également déboguer nativement node.js en le démarrant comme suit:

```
node debug your-script.js
```

Pour cerner votre débogueur exactement dans une ligne de code souhaitée, utilisez ceci:

```
debugger;
```

Pour plus d'informations, voir ici .

Dans node.js 8, utilisez la commande suivante:

```
node --inspect-brk your-script.js
```

Ouvrez ensuite à `about://inspect` une version récente de Google Chrome et sélectionnez votre script Node pour obtenir l'expérience de débogage des outils DevTools de Chrome.

Bonjour tout le monde avec Express

L'exemple suivant utilise Express pour créer un serveur HTTP écoutant sur le port 3000, qui répond par "Hello, World!". Express est un framework Web couramment utilisé qui est utile pour créer des API HTTP.

Créez d'abord un nouveau dossier, par exemple `myApp` . Allez dans `myApp` et créez un nouveau fichier JavaScript contenant le code suivant (`hello.js` le `hello.js` par exemple). Ensuite, installez le module express en utilisant `npm install --save express` depuis la ligne de commande. *Reportez-vous à cette documentation pour plus d'informations sur l'installation des packages* .

```
// Import the top-level function of express
const express = require('express');

// Creates an Express application using the top-level function
const app = express();

// Define port number as 3000
const port = 3000;

// Routes HTTP GET requests to the specified path "/" with the specified callback function
app.get('/', function(request, response) {
  response.send('Hello, World!');
});

// Make the app listen on port 3000
app.listen(port, function() {
  console.log('Server listening on http://localhost:' + port);
});
```

À partir de la ligne de commande, exécutez la commande suivante:

```
node hello.js
```

Ouvrez votre navigateur et accédez à `http://localhost:3000` ou `http://127.0.0.1:3000` pour voir la réponse.

Pour plus d'informations sur le framework Express, vous pouvez consulter la section Web Apps With Express .

Routage de base Hello World

Une fois que vous avez compris comment créer un serveur HTTP avec un noeud, il est important de comprendre comment le faire «faire» des choses en fonction du chemin d'accès auquel l'utilisateur a accédé. Ce phénomène est appelé "routage".

L'exemple le plus fondamental serait de vérifier `if (request.url === 'some/path/here')` , puis d'appeler une fonction qui répond avec un nouveau fichier.

Un exemple de ceci peut être vu ici:

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

function index (request, response) {
    response.writeHead(200);
    response.end('Hello, World!');
}

http.createServer(function (request, response) {

    if (request.url === '/') {
        return index(request, response);
    }

    response.writeHead(404);
    response.end(http.STATUS_CODES[404]);

}).listen(1337);
```

Si vous continuez à définir vos "routes" comme ça, vous vous retrouverez avec une fonction de rappel énorme, et nous ne voulons pas un désordre géant comme ça, alors voyons si nous pouvons le nettoyer.

Tout d'abord, stockons toutes nos routes dans un objet:

```
var routes = {
    '/': function index (request, response) {
        response.writeHead(200);
        response.end('Hello, World!');
    },
    '/foo': function foo (request, response) {
        response.writeHead(200);
        response.end('You are now viewing "foo"');
    }
}
```

Maintenant que nous avons stocké 2 routes dans un objet, nous pouvons maintenant les vérifier dans notre rappel principal:

```
http.createServer(function (request, response) {

    if (request.url in routes) {
        return routes[request.url](request, response);
    }
```

```
    response.writeHead(404);
    response.end(http.STATUS_CODES[404]);

}).listen(1337);
```

Maintenant, chaque fois que vous essayez de naviguer sur votre site Web, il vérifie l'existence de ce chemin dans vos itinéraires et appelle la fonction correspondante. Si aucun itinéraire n'est trouvé, le serveur répondra par un 404 (non trouvé).

Et voilà - le routage avec l'API HTTP Server est très simple.

Socket TLS: serveur et client

Les seules différences majeures entre ceci et une connexion TCP standard sont la clé privée et le certificat public que vous devrez définir dans un objet d'option.

Comment créer une clé et un certificat

La première étape de ce processus de sécurité est la création d'une clé privée. Et quelle est cette clé privée? Fondamentalement, c'est un ensemble de bruits aléatoires utilisés pour chiffrer les informations. En théorie, vous pouvez créer une clé et l'utiliser pour chiffrer ce que vous voulez. Mais il est préférable d'avoir des clés différentes pour des choses spécifiques. Parce que si quelqu'un vole votre clé privée, c'est comme si quelqu'un volait les clés de votre maison. Imaginez si vous utilisiez la même clé pour verrouiller votre voiture, votre garage, votre bureau, etc.

```
openssl genrsa -out private-key.pem 1024
```

Une fois que nous avons notre clé privée, nous pouvons créer une demande de signature de certificat (CSR), qui est notre demande de faire signer la clé privée par une autorité de fantaisie. C'est pourquoi vous devez saisir des informations relatives à votre entreprise. Cette information sera visible par le signataire autorisé et utilisée pour vous vérifier. Dans notre cas, peu importe ce que vous tapez, puisque nous allons signer notre certificat à l'étape suivante.

```
openssl req -new -key private-key.pem -out csr.pem
```

Maintenant que nous avons rempli nos documents, il est temps de prétendre que nous sommes une autorité de signature géniale.

```
openssl x509 -req -in csr.pem -signkey private-key.pem -out public-cert.pem
```

Maintenant que vous avez la clé privée et le certificat public, vous pouvez établir une connexion sécurisée entre deux applications NodeJS. Et, comme vous pouvez le voir dans l'exemple de code, le processus est très simple.

Important!

Puisque nous avons créé le certificat public nous-mêmes, en toute honnêteté, notre certificat ne

vaut rien, parce que nous ne sommes rien. Le serveur NodeJS ne fera pas confiance à un tel certificat par défaut, et c'est pourquoi nous devons lui demander de faire confiance à notre certificat avec l'option rejectUnauthorized suivante: false. **Très important** : ne définissez jamais cette variable sur true dans un environnement de production.

Serveur Socket TLS

```
'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

var options = {
    key: fs.readFileSync('private-key.pem'),
    cert: fs.readFileSync('public-cert.pem')
};

var server = tls.createServer(options, function(socket) {

    // Send a friendly message
    socket.write("I am the server sending you a message.");

    // Print the data that we received
    socket.on('data', function(data) {

        console.log('Received: %s [it is %d bytes long]',
            data.toString().replace(/(\n)/gm,""),
            data.length);

    });

    // Let us know when the transmission is over
    socket.on('end', function() {

        console.log('EOT (End Of Transmission)');

    });

});

// Start listening on a specific port and address
server.listen(PORT, HOST, function() {

    console.log("I'm listening at %s, on port %s", HOST, PORT);

});

// When an error occurs, show it.
server.on('error', function(error) {

    console.error(error);

    // Close the connection after the error occurred.
    server.destroy();
```

```
});
```

TLS Socket Client

```
'use strict';

var tls = require('tls');
var fs = require('fs');

const PORT = 1337;
const HOST = '127.0.0.1'

// Pass the certs to the server and let it know to process even unauthorized certs.
var options = {
    key: fs.readFileSync('private-key.pem'),
    cert: fs.readFileSync('public-cert.pem'),
    rejectUnauthorized: false
};

var client = tls.connect(PORT, HOST, options, function() {

    // Check if the authorization worked
    if (client.authorized) {
        console.log("Connection authorized by a Certificate Authority.");
    } else {
        console.log("Connection not authorized: " + client.authorizationError)
    }

    // Send a friendly message
    client.write("I am the client sending you a message.");

});

client.on("data", function(data) {

    console.log('Received: %s [it is %d bytes long]',
        data.toString().replace(/(\n)/gm,""),
        data.length);

    // Close the connection after receiving the message
    client.end();

});

client.on('close', function() {

    console.log("Connection closed");

});

// When an error ocoures, show it.
client.on('error', function(error) {

    console.error(error);

    // Close the connection after the error occurred.
    client.destroy();
```

```
});
```

Bonjour tout le monde dans le REPL

Appelé sans arguments, Node.js démarre une REPL (Read-Eval-Print-Loop), également appelée « *shell Node* ».

À l'invite de commandes, tapez `node` .

```
$ node
>
```

À l'invite du shell Node `>` tapez "Hello World!"

```
$ node
> "Hello World!"
'Hello World!'
```

Modules de base

Node.js est un moteur Javascript (moteur V8 de Google pour Chrome, écrit en C ++) qui permet d'exécuter Javascript en dehors du navigateur. Bien que de nombreuses bibliothèques soient disponibles pour étendre les fonctionnalités de Node, le moteur est livré avec un ensemble de *modules* de *base intégrant* des fonctionnalités de base.

Il y a actuellement 34 modules de base inclus dans Node:

```
[ 'assert',
  'buffer',
  'c/c++_addons',
  'child_process',
  'cluster',
  'console',
  'crypto',
  'deprecated_apis',
  'dns',
  'domain',
  'Events',
  'fs',
  'http',
  'https',
  'module',
  'net',
  'os',
  'path',
  'punycode',
  'querystring',
  'readline',
  'repl',
  'stream',
  'string_decoder',
  'timers',
```

```
  'tls_(ssl)',
  'tracing',
  'tty',
  'dgram',
  'url',
  'util',
  'v8',
  'vm',
  'zlib' ]
```

Cette liste a été obtenue à partir de l'API de documentation Node https://nodejs.org/api/all.html (fichier JSON: https://nodejs.org/api/all.json).

Tous les modules de base en un coup d'œil

affirmer

Le module `assert` fournit un ensemble simple de tests d'assertion pouvant être utilisés pour tester les invariants.

tampon

Avant l'introduction de `TypedArray` dans ECMAScript 2015 (ES6), le langage JavaScript ne disposait d'aucun mécanisme pour lire ou manipuler des flux de données binaires. La classe `Buffer` été introduite dans le cadre de l'API Node.js pour permettre l'interaction avec les flux d'octets dans le cadre d'activités telles que les flux TCP et les opérations de système de fichiers.

Maintenant que `TypedArray` a été ajouté à ES6, la classe `Buffer` implémente l'API `Uin t8Array` de manière plus optimisée et adaptée aux cas d'utilisation de Node.js.

c / c ++ _ addons

Node.js Les Addons sont des objets partagés liés dynamiquement, écrits en C ou C ++, qui peuvent être chargés dans Node.js en utilisant la fonction `require()` , et utilisés comme s'ils étaient un module Node.js ordinaire. Ils servent principalement à fournir une interface entre JavaScript s'exécutant dans les bibliothèques Node.js et C / C ++.

child_process

Le module `child_process` permet de générer des processus enfant de manière similaire, mais pas identique, à popen (3).

grappe

Une seule instance de Node.js s'exécute dans un seul thread. Pour tirer parti des systèmes multi-core, l'utilisateur voudra parfois lancer un cluster de processus Node.js pour gérer la charge. Le module de cluster vous permet de créer facilement des processus enfants partageant tous des ports de serveur.

console

Le module de `console` fournit une console de débogage simple, similaire au mécanisme de console JavaScript fourni par les navigateurs Web.

crypto

Le `crypto` module fournit des fonctionnalités de chiffrement qui comprend un ensemble d'emballages pour le hachage de OpenSSL, HMAC, chiffrement, déchiffrement, signer et vérifier les fonctions.

deprecated_apis

Node.js peut rendre obsolètes les API lorsque: (a) l'utilisation de l'API est considérée comme non sécurisée, (b) une autre API améliorée a été mise à disposition, ou (c) des modifications de l'API sont attendues dans une prochaine version majeure .

dns

Le module `dns` contient des fonctions appartenant à deux catégories différentes:

1. Fonctions qui utilisent les fonctionnalités du système d'exploitation sous-jacent pour effectuer la résolution de noms et n'effectuent pas nécessairement de communication réseau. Cette catégorie ne contient qu'une seule fonction: `dns.lookup()` .
2. Fonctions qui se connectent à un serveur DNS réel pour effectuer la résolution de noms et qui utilisent *toujours* le réseau pour effectuer des requêtes DNS. Cette catégorie contient toutes les fonctions du module `dns` *exception de* `dns.lookup()` .

domaine

Ce module est en attente de dépréciation . Une fois qu'une API de remplacement a été finalisée, ce module sera complètement obsolète. La plupart des utilisateurs finaux **ne** devraient **pas** avoir de raison d'utiliser ce module. Les utilisateurs qui doivent absolument disposer de la fonctionnalité fournie par les domaines peuvent, pour le moment, compter sur elle, mais doivent s'attendre à devoir migrer vers une solution différente à l'avenir.

Événements

Une grande partie de l'API de base de Node.js est construite autour d'une architecture asynchrone pilotée par des événements idiomatiques dans laquelle certains types d'objets (appelés "émetteurs") émettent périodiquement des événements nommés provoquant l'appel des objets Fonction ("écouteurs").

fs

Les E / S sur fichiers sont fournies par des wrappers simples autour des fonctions POSIX standard. Pour utiliser ce module, `require('fs')` . Toutes les méthodes ont des formes asynchrones et synchrones.

http

Les interfaces HTTP dans Node.js sont conçues pour prendre en charge de nombreuses

fonctionnalités du protocole qui étaient traditionnellement difficiles à utiliser. En particulier, des messages volumineux, éventuellement codés en bloc. L'interface prend soin de ne jamais mettre en mémoire tampon des demandes ou des réponses entières - l'utilisateur peut diffuser des données.

https

HTTPS est le protocole HTTP sur TLS / SSL. Dans Node.js, ceci est implémenté en tant que module séparé.

module

Node.js a un système de chargement de module simple. Dans Node.js, les fichiers et les modules sont en correspondance directe (chaque fichier est traité comme un module distinct).

net

Le module `net` vous fournit un wrapper réseau asynchrone. Il contient des fonctions pour créer à la fois des serveurs et des clients (appelés flux). Vous pouvez inclure ce module avec `require('net');` .

os

Le module `os` fournit un certain nombre de méthodes utilitaires liées au système d'exploitation.

chemin

Le module `path` fournit des utilitaires pour travailler avec des chemins de fichiers et de répertoires.

punycode

La version du module punycode intégrée à Node.js est obsolète .

chaîne de requête

Le module `querystring` fournit des utilitaires pour analyser et formater les chaînes de requête URL.

readline

Le module `readline` fournit une interface pour lire les données à partir d'un flux lisible (tel que `process.stdin`) une ligne à la fois.

repl

Le module `repl` fournit une implémentation REPL (Read-Eval-Print-Loop) disponible à la fois en tant que programme autonome ou dans d'autres applications.

courant

Un flux est une interface abstraite pour travailler avec des données en continu dans Node.js. Le module de `stream` fournit une API de base qui facilite la création d'objets qui implémentent

l'interface de flux.

Il y a beaucoup d'objets de flux fournis par Node.js. Par exemple, une requête sur un serveur HTTP et `process.stdout` sont des instances de flux.

string_decoder

Le module `string_decoder` fournit une API pour décoder les objets `Buffer` en chaînes de manière à préserver les caractères codés sur plusieurs octets UTF-8 et UTF-16.

minuteries

Le module de `timer` expose une API globale pour les fonctions de planification à appeler ultérieurement. Comme les fonctions du minuteur sont des globales, il n'est pas nécessaire d'appeler `require('timers')` pour utiliser l'API.

Les fonctions du minuteur dans Node.js implémentent une API similaire à celle de l'API de temporisation fournie par les navigateurs Web, mais utilisent une implémentation interne différente basée sur la boucle d'événement Node.js.

tls_ (ssl)

Le module `tls` fournit une implémentation des protocoles TLS (Transport Layer Security) et SSL (Secure Socket Layer) basés sur OpenSSL.

tracé

Trace Event fournit un mécanisme permettant de centraliser les informations de traçage générées par V8, Node core et le code de l'espace utilisateur.

Le suivi peut être activé en passant l' `--trace-events-enabled` lors du démarrage d'une application Node.js.

tty

Le module `tty` fournit les classes `tty.ReadStream` et `tty.WriteStream` . Dans la plupart des cas, il ne sera pas nécessaire ou possible d'utiliser ce module directement.

dgram

Le module `dgram` fournit une implémentation des sockets UDP Datagram.

URL

Le module `url` fournit des utilitaires pour la résolution et l'analyse syntaxique des URL.

util

Le module `util` est principalement conçu pour répondre aux besoins des API internes de Node.js. Cependant, de nombreux utilitaires sont également utiles pour les développeurs d'applications et de modules.

v8

Le module `v8` expose les API spécifiques à la version de V8 intégrée au binaire Node.js.

Remarque : Les API et l'implémentation sont susceptibles d'être modifiées à tout moment.

vm

Le module `vm` fournit des API pour compiler et exécuter du code dans les contextes de machine virtuelle V8. Le code JavaScript peut être compilé et exécuté immédiatement ou compilé, enregistré et exécuté plus tard.

Remarque : Le module vm n'est pas un mécanisme de sécurité. ***Ne l'utilisez pas pour exécuter du code non fiable*** .

zlib

Le module `zlib` fournit des fonctionnalités de compression implémentées avec Gzip et Deflate / Inflate.

Comment faire fonctionner un serveur Web HTTPS de base!

Une fois que node.js est installé sur votre système, vous pouvez simplement suivre la procédure ci-dessous pour obtenir un serveur Web de base compatible avec HTTP et HTTPS!

Étape 1: créer une autorité de certification

1. Créez le dossier dans lequel vous souhaitez stocker votre clé et votre certificat:

```
mkdir conf
```

---

2. allez dans ce répertoire:

```
cd conf
```

---

3. récupérer ce fichier `ca.cnf` à utiliser comme raccourci de configuration:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

---

4. créer une nouvelle autorité de certification en utilisant cette configuration:

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

---

5. Maintenant que nous avons notre autorité de certification dans `ca-key.pem` et `ca-cert.pem` , générons une clé privée pour le serveur:

```
openssl genrsa -out key.pem 4096
```

---

6. récupérer ce fichier `server.cnf` à utiliser comme raccourci de configuration:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/server.cnf
```

---

7. générer la demande de signature de certificat en utilisant cette configuration:

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

---

8. signer la demande:

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

Étape 2: installez votre certificat en tant que certificat racine

1. copiez votre certificat dans le dossier de vos certificats racine:

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

---

2. mettre à jour le magasin CA:

```
sudo update-ca-certificates
```

Étape 3: Démarrer votre serveur de noeud

Tout d'abord, vous voulez créer un fichier `server.js` contenant votre code de serveur actuel.

La configuration minimale pour un serveur HTTPS dans Node.js serait la suivante:

```
var https = require('https');
var fs = require('fs');

var httpsOptions = {
    key: fs.readFileSync('path/to/server-key.pem'),
    cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

https.createServer(httpsOptions, app).listen(4433);
```

Si vous souhaitez également prendre en charge les requêtes http, vous devez apporter cette petite modification:

```
var http = require('http');
var https = require('https');
var fs = require('fs');

var httpsOptions = {
    key: fs.readFileSync('path/to/server-key.pem'),
    cert: fs.readFileSync('path/to/server-crt.pem')
};

var app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

1. allez dans le répertoire où se trouve votre `server.js` :

   ```
   cd /path/to
   ```

---

2. lancez `server.js` :

   ```
   node server.js
   ```

Lire Démarrer avec Node.js en ligne: https://riptutorial.com/fr/node-js/topic/340/demarrer-avec-node-js

Chapitre 2: Analyse des arguments de ligne de commande

Examples

Action de passage (verbe) et valeurs

```
const options = require("commander");

options
  .option("-v, --verbose", "Be verbose");

options
  .command("convert")
  .alias("c")
  .description("Converts input file to output file")
  .option("-i, --in-file <file_name>", "Input file")
  .option("-o, --out-file <file_name>", "Output file")
  .action(doConvert);

options.parse(process.argv);

if (!options.args.length) options.help();

function doConvert(options){
    //do something with options.inFile and options.outFile
};
```

Interrupteurs booléens

```
const options = require("commander");

options
  .option("-v, --verbose")
  .parse(process.argv);

if (options.verbose){
  console.log("Let's make some noise!");
}
```

Lire Analyse des arguments de ligne de commande en ligne: https://riptutorial.com/fr/node-js/topic/6174/analyse-des-arguments-de-ligne-de-commande

Chapitre 3: analyseur csv dans le noeud js

Introduction

La lecture des données à partir d'un CSV peut être traitée de plusieurs manières. Une solution consiste à lire le fichier `csv` dans un tableau. De là, vous pouvez travailler sur le tableau.

Examples

Utiliser FS pour lire dans un fichier CSV

fs est l' API du système de fichiers dans le noeud. Nous pouvons utiliser la méthode readFile sur notre variable fs, lui transmettre un fichier `data.csv` , un format et une fonction qui lisent et divisent le `csv` pour un traitement ultérieur.

Cela suppose que vous avez un fichier nommé `data.csv` dans le même dossier.

```
'use strict'

const fs = require('fs');

fs.readFile('data.csv', 'utf8', function (err, data) {
  var dataArray = data.split(/\r?\n/);
  console.log(dataArray);
});
```

Vous pouvez maintenant utiliser le tableau comme n'importe quel autre pour y travailler.

Lire analyseur csv dans le noeud js en ligne: https://riptutorial.com/fr/node-js/topic/9162/analyseur-csv-dans-le-noeud-js

Chapitre 4: API CRUD simple basée sur REST

Examples

API REST pour CRUD dans Express 3+

```
var express = require("express"),
    bodyParser = require("body-parser"),
    server = express();

//body parser for parsing request body
server.use(bodyParser.json());
server.use(bodyParser.urlencoded({ extended: true }));

//temperary store for `item` in memory
var itemStore = [];

//GET all items
server.get('/item', function (req, res) {
    res.json(itemStore);
});

//GET the item with specified id
server.get('/item/:id', function (req, res) {
    res.json(itemStore[req.params.id]);
});

//POST new item
server.post('/item', function (req, res) {
    itemStore.push(req.body);
    res.json(req.body);
});

//PUT edited item in-place of item with specified id
server.put('/item/:id', function (req, res) {
    itemStore[req.params.id] = req.body
    res.json(req.body);
});

//DELETE item with specified id
server.delete('/item/:id', function (req, res) {
    itemStore.splice(req.params.id, 1)
    res.json(req.body);
});

//START SERVER
server.listen(3000, function () {
    console.log("Server running");
})
```

Lire API CRUD simple basée sur REST en ligne: https://riptutorial.com/fr/node-js/topic/5850/api-crud-simple-basee-sur-rest

Chapitre 5: Applications Web avec Express

Introduction

Express est une infrastructure d'application Web Node.js minimale et flexible, fournissant un ensemble robuste de fonctionnalités pour la création d'applications Web.

Le site officiel d'Express est expressjs.com . La source peut être trouvée sur GitHub .

Syntaxe

- app.get (chemin [, middleware], callback [, callback ...])
- app.put (chemin [, middleware], callback [, callback ...])
- app.post (chemin [, middleware], callback [, callback ...])
- app ['delete'] (chemin [, middleware], callback [, callback ...])
- app.use (chemin [, middleware], callback [, callback ...])
- app.use (rappel)

Paramètres

Paramètre	Détails
`path`	Spécifie la portion de chemin ou l'URL que le rappel donné va gérer.
`middleware`	Une ou plusieurs fonctions qui seront appelées avant le rappel. Essentiellement un chaînage de plusieurs fonctions de `callback` . Utile pour une manipulation plus spécifique, par exemple une autorisation ou un traitement des erreurs.
`callback`	Une fonction qui sera utilisée pour gérer les demandes sur le `path` spécifié. Il sera appelé comme `callback(request, response, next)` , où `request` , `response` et `next` sont décrits ci-dessous.
`request` *rappel*	Un objet encapsulant des détails sur la requête HTTP que le rappel est appelé à gérer.
`response`	Un objet utilisé pour spécifier comment le serveur doit répondre à la demande.
`next`	Un rappel qui passe le contrôle au prochain itinéraire correspondant. Il accepte un objet d'erreur facultatif.

Examples

Commencer

Vous devez d'abord créer un répertoire, y accéder dans votre shell et installer Express en utilisant npm en exécutant `npm install express --save`

Créez un fichier et nommez-le `app.js` et ajoutez le code suivant qui crée un nouveau serveur Express et lui ajoute un point de terminaison (`/ping`) avec la méthode `app.get` :

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

const app = express();

app.get('/ping', (request, response) => {
    response.send('pong');
});

app.listen(8080, 'localhost');
```

Pour exécuter votre script, utilisez la commande suivante dans votre shell:

```
> node app.js
```

Votre application acceptera les connexions sur le port localhost 8080. Si l'argument hostname de `app.listen` est omis, le serveur acceptera les connexions sur l'adresse IP de la machine et sur localhost. Si la valeur du port est 0, le système d'exploitation attribuera un port disponible.

Une fois que votre script est en cours d'exécution, vous pouvez le tester dans un shell pour confirmer que vous obtenez la réponse attendue, "pong", du serveur:

```
> curl http://localhost:8080/ping
pong
```

Vous pouvez également ouvrir un navigateur Web, accédez à l'URL http: // localhost: 8080 / ping pour afficher la sortie

Routage de base

Commencez par créer une application express:

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

Ensuite, vous pouvez définir des itinéraires comme celui-ci:

```
app.get('/someUri', function (req, res, next) {})
```

Cette structure fonctionne pour toutes les méthodes HTTP et attend un chemin comme premier argument et un gestionnaire pour ce chemin qui reçoit les objets de requête et de réponse. Donc, pour les méthodes HTTP de base, ce sont les routes

```
// GET www.domain.com/myPath
```

```
app.get('/myPath', function (req, res, next) {})

// POST www.domain.com/myPath
app.post('/myPath', function (req, res, next) {})

// PUT www.domain.com/myPath
app.put('/myPath', function (req, res, next) {})

// DELETE www.domain.com/myPath
app.delete('/myPath', function (req, res, next) {})
```

Vous pouvez vérifier la liste complète des verbes pris en charge ici . Si vous souhaitez définir le même comportement pour une route et toutes les méthodes HTTP, vous pouvez utiliser:

```
app.all('/myPath', function (req, res, next) {})
```

ou

```
app.use('/myPath', function (req, res, next) {})
```

ou

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

// * wildcard will route for all paths
```

Vous pouvez enchaîner vos définitions de route pour un chemin unique

```
app.route('/myPath')
  .get(function (req, res, next) {})
  .post(function (req, res, next) {})
  .put(function (req, res, next) {})
```

Vous pouvez également ajouter des fonctions à toute méthode HTTP. Ils s'exécuteront avant le rappel final et prendront les paramètres (req, res, next) comme arguments.

```
// GET www.domain.com/myPath
app.get('/myPath', myFunction, function (req, res, next) {})
```

Vos derniers rappels peuvent être stockés dans un fichier externe pour éviter de mettre trop de code dans un fichier:

```
// other.js
exports.doSomething = function(req, res, next) {/* do some stuff */};
```

Et puis dans le fichier contenant vos itinéraires:

```
const other = require('./other.js');
app.get('/someUri', myFunction, other.doSomething);
```

Cela rendra votre code beaucoup plus propre.

Obtenir des informations à partir de la demande

Pour obtenir des informations de la part de l'url requérante (notez que `req` est l'objet de requête dans la fonction de gestionnaire des itinéraires). Considérez cette définition **`/settings/:user_id`** et cet exemple particulier **`/settings/32135?field=name`**

```
// get the full path
req.originalUrl // => /settings/32135?field=name

// get the user_id param
req.params.user_id // => 32135

// get the query value of the field
req.query.field // => 'name'
```

Vous pouvez également obtenir les en-têtes de la requête, comme ceci

```
req.get('Content-Type')
// "text/plain"
```

Pour simplifier l'obtention d'autres informations, vous pouvez utiliser des middlewares. Par exemple, pour obtenir les informations sur le corps de la requête, vous pouvez utiliser le middleware analyseur de corps , qui transformera le corps de la requête brute en un format utilisable.

```
var app = require('express')();
var bodyParser = require('body-parser');

app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-
urlencoded
```

Maintenant, supposons une requête comme celle-ci

```
PUT /settings/32135
{
  "name": "Peter"
}
```

Vous pouvez accéder au nom affiché comme ceci

```
req.body.name
// "Peter"
```

De la même manière, vous pouvez accéder aux cookies de la requête, vous avez également besoin d'un middleware comme cookie-parser

```
req.cookies.name
```

Application express modulaire

Pour rendre les applications de modulaires d'application web express modulaires:

Module:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
    const router = express.Router();

    router.get('/greet', (req, res, next) => {
        res.end(options.greeting);
    });

    return router;
};
```

Application:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

express()
    .use('/api/v1/', greetMiddleware({ greeting:'Hello world' }))
    .listen(8080);
```

Cela rendra votre application modulable, personnalisable et votre code réutilisable.

Lorsque vous accédez à `http://<hostname>:8080/api/v1/greet` le résultat sera `Hello world`

Exemple plus compliqué

Exemple avec des services qui montrent les avantages d'une usine middleware.

Module:

```
// greet.js
const express = require('express');

module.exports = function(options = {}) { // Router factory
    const router = express.Router();
    // Get controller
    const {service} = options;

    router.get('/greet', (req, res, next) => {
        res.end(
            service.createGreeting(req.query.name || 'Stranger')
        );
    });
```

```
    return router;
};
```

Application:

```
// app.js
const express = require('express');
const greetMiddleware = require('./greet.js');

class GreetingService {
    constructor(greeting = 'Hello') {
        this.greeting = greeting;
    }

    createGreeting(name) {
        return `${this.greeting}, ${name}!`;
    }
}

express()
    .use('/api/v1/service1', greetMiddleware({
        service: new GreetingService('Hello'),
    }))
    .use('/api/v1/service2', greetMiddleware({
        service: new GreetingService('Hi'),
    }))
    .listen(8080);
```

Lorsque vous accédez à `http://<hostname>:8080/api/v1/service1/greet?name=World` le résultat sera `Hello, World` et vous accéderez à `http://<hostname>:8080/api/v1/service2/greet?name=World` La sortie sera `Hi, World` .

Utiliser un moteur de template

Utiliser un moteur de template

Le code suivant va configurer Jade comme moteur de template. (Remarque: Jade a été renommé `pug` en décembre 2015.)

```
const express = require('express');  //Imports the express module
const app = express();  //Creates an instance of the express module

const PORT = 3000; //Randomly chosen port

app.set('view engine','jade'); //Sets jade as the View Engine / Template Engine
app.set('views','src/views'); //Sets the directory where all the views (.jade files) are
stored.

//Creates a Root Route
app.get('/',function(req, res){
    res.render('index');  //renders the index.jade file into html and returns as a response.
The render function optionally takes the data to pass to the view.
});

//Starts the Express server with a callback
```

```
app.listen(PORT, function(err) {
    if (!err) {
        console.log('Server is running at port', PORT);
    } else {
        console.log(JSON.stringify(err));
    }
});
```

De même, d'autres moteurs de gabarit pourraient être utilisés, tels que des `Handlebars` (`hbs`) ou des `ejs` . N'oubliez pas de `npm install` le moteur de template aussi. Pour les guidons, nous utilisons un paquetage `hbs` , pour Jade, nous avons un paquet `jade` et pour EJS, nous avons un paquet `ejs` .

Exemple de modèle EJS

Avec EJS (comme les autres modèles express), vous pouvez exécuter du code serveur et accéder à vos variables serveur à partir de votre code HTML.
Dans EJS, on utilise " `<%` " comme balise de début et " `%>` " comme balise de fin, les variables transmises comme les paramètres de rendu sont accessibles avec `<%=var_name%>`
Par exemple, si vous avez une baie de consommables dans votre code serveur
vous pouvez le parcourir en utilisant

```
<h1><%= title %></h1>
   <ul>
<% for(var i=0; i<supplies.length; i++) { %>
    <li>
        <a href='supplies/<%= supplies[i] %>'>
            <%= supplies[i] %>
        </a>
    </li>
<% } %>
```

Comme vous pouvez le voir dans l'exemple chaque fois que vous passez le code côté serveur et HTML que vous devez fermer la balise EJS actuelle et ouvrir un nouveau plus tard, nous voulions créer `li` l' intérieur de la `for` commande si nous devions fermer notre étiquette EJS à la fin du `for` et créer une nouvelle balise juste pour les accolades
un autre exemple
si nous voulons mettre en entrée la version par défaut pour être une variable du côté serveur, nous utilisons `<%=`
par exemple:

```
 Message:<br>
<input type="text" value="<%= message %>" name="message" required>
```

Ici, la variable de message transmise de votre côté serveur sera la valeur par défaut de votre saisie. Notez que si vous ne transmettez pas la variable de message depuis votre serveur, EJS lancera une exception. Vous pouvez passer des paramètres à l'aide de `res.render('index', {message: message});` (pour le fichier ejs appelé index.ejs).

Dans les balises EJS, vous pouvez également utiliser `if` , `while` ou toute autre commande

javascript souhaitée.

API JSON avec ExpressJS

```
var express = require('express');
var cors = require('cors'); // Use cors module for enable Cross-origin resource sharing

var app = express();
app.use(cors()); // for all routes

var port = process.env.PORT || 8080;

app.get('/', function(req, res) {
    var info = {
        'string_value': 'StackOverflow',
        'number_value': 8476
    }
    res.json(info);

    // or
    /* res.send(JSON.stringify({
        string_value: 'StackOverflow',
        number_value: 8476
    })) */

  //you can add a status code to the json response
   /* res.status(200).json(info) */
})

app.listen(port, function() {
    console.log('Node.js listening on port ' + port)
})
```

Sur `http://localhost:8080/` output object

```
{
    string_value: "StackOverflow",
    number_value: 8476
}
```

Servant des fichiers statiques

Lors de la création d'un serveur Web avec Express, il est souvent nécessaire de fournir une combinaison de contenu dynamique et de fichiers statiques.

Par exemple, vous pouvez avoir index.html et script.js qui sont des fichiers statiques conservés dans le système de fichiers.

Il est courant d'utiliser le dossier nommé 'public' pour avoir des fichiers statiques. Dans ce cas, la structure des dossiers peut ressembler à ceci:

```
project root
├── server.js
├── package.json
└── public
```

```
    ├── index.html
    └── script.js
```

Voici comment configurer Express pour servir des fichiers statiques:

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

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

Remarque: une fois le dossier configuré, index.html, script.js et tous les fichiers du dossier "public" seront disponibles dans le chemin racine (vous ne devez pas spécifier `/public/` dans l'URL). En effet, express recherche les fichiers relatifs au dossier statique configuré. Vous pouvez spécifier *le préfixe de chemin virtuel* comme indiqué ci-dessous:

```
app.use('/static', express.static('public'));
```

rendra les ressources disponibles sous le préfixe `/static/` .

Plusieurs dossiers

Il est possible de définir plusieurs dossiers en même temps:

```
app.use(express.static('public'));
app.use(express.static('images'));
app.use(express.static('files'));
```

Lors de la diffusion des ressources, Express examinera le dossier dans l'ordre de définition. Dans le cas de fichiers portant le même nom, celui du premier dossier correspondant sera servi.

Routes nommées dans le style Django

Un gros problème est que les itinéraires nommés de valeur ne sont pas pris en charge par Express. La solution consiste à installer un package tiers pris en charge, par exemple express-reverse :

```
npm install express-reverse
```

Branchez-le dans votre projet:

```
var app = require('express')();
require('express-reverse')(app);
```

Ensuite, utilisez-le comme:

```
app.get('test', '/hello', function(req, res) {
  res.end('hello');
});
```

L'inconvénient de cette approche est que vous ne pouvez pas utiliser le module `route` Express comme indiqué dans Utilisation avancée du routeur . La solution consiste à transmettre votre `app` tant que paramètre à votre fabrique de routeurs:

```
require('./middlewares/routing')(app);
```

Et l'utiliser comme:

```
module.exports = (app) => {
    app.get('test', '/hello', function(req, res) {
      res.end('hello');
    });
};
```

Vous pouvez désormais comprendre comment définir des fonctions pour le fusionner avec les espaces de noms personnalisés spécifiés et pointer vers les contrôleurs appropriés.

La gestion des erreurs

Gestion des erreurs de base

Par défaut, Express recherchera une vue "erreur" dans le répertoire `/views` pour effectuer le rendu. Créez simplement la vue 'error' et placez-la dans le répertoire views pour gérer les erreurs. Les erreurs sont écrites avec le message d'erreur, l'état et la trace de la pile, par exemple:

views / error.pug

```
html
  body
      h1= message
      h2= error.status
      p= error.stack
```

Gestion avancée des erreurs

Définissez les fonctions de gestion des erreurs à la toute fin de la pile de fonctions du middleware. Celles-ci ont quatre arguments au lieu de trois `(err, req, res, next)` par exemple:

app.js

```
// catch 404 and forward to error handler
app.use(function(req, res, next) {
    var err = new Error('Not Found');
    err.status = 404;

    //pass error to the next matching route.
    next(err);
});

// handle error, print stacktrace
app.use(function(err, req, res, next) {
    res.status(err.status || 500);
```

```
    res.render('error', {
        message: err.message,
        error: err
    });
});
```

Vous pouvez définir plusieurs fonctions de middleware de gestion des erreurs, comme vous le feriez avec des fonctions de middleware standard.

Utiliser le middleware et le prochain rappel

Express transmet un rappel `next` à chaque fonction de gestionnaire de routage et de middleware qui peut être utilisée pour rompre la logique des itinéraires uniques entre plusieurs gestionnaires. L'appel de `next()` sans arguments indique à express de continuer vers le middleware ou le gestionnaire de route suivant. L'appel à `next(err)` avec une erreur déclenchera tout middleware de gestionnaire d'erreurs. L'appel `next('route')` contournera tout middleware suivant sur l'itinéraire actuel et passera à l'itinéraire suivant. Cela permet de découpler la logique de domaine en composants réutilisables, autonomes, plus simples à tester et plus faciles à gérer et à modifier.

Plusieurs itinéraires correspondants

Les demandes à `/api/foo` ou à `/api/bar` exécuteront le gestionnaire initial pour rechercher le membre, puis passer le contrôle au gestionnaire réel pour chaque route.

```
app.get('/api', function(req, res, next) {
  // Both /api/foo and /api/bar will run this
  lookupMember(function(err, member) {
    if (err) return next(err);
    req.member = member;
    next();
  });
});

app.get('/api/foo', function(req, res, next) {
  // Only /api/foo will run this
  doSomethingWithMember(req.member);
});

app.get('/api/bar', function(req, res, next) {
  // Only /api/bar will run this
  doSomethingDifferentWithMember(req.member);
});
```

Gestionnaire d'erreur

Les gestionnaires d'erreurs sont des middlewares avec la `function(err, req, res, next)` signature `function(err, req, res, next)` . Ils peuvent être configurés par route (par exemple, `app.get('/foo', function(err, req, res, next)`) mais généralement, un seul gestionnaire d’erreur qui affiche une page d’erreur suffit.

```
app.get('/foo', function(req, res, next) {
  doSomethingAsync(function(err, data) {
```

```
    if (err) return next(err);
    renderPage(data);
  });
});

// In the case that doSomethingAsync return an error, this special
// error handler middleware will be called with the error as the
// first parameter.
app.use(function(err, req, res, next) {
  renderErrorPage(err);
});
```

Middleware

Chacune des fonctions ci-dessus est en fait une fonction de middleware exécutée à chaque fois qu'une requête correspond à la route définie, mais un nombre quelconque de fonctions de middleware peut être défini sur une seule route. Cela permet de définir le middleware dans des fichiers séparés et de réutiliser la logique commune sur plusieurs routes.

```
app.get('/bananas', function(req, res, next) {
  getMember(function(err, member) {
    if (err) return next(err);
    // If there's no member, don't try to look
    // up data. Just go render the page now.
    if (!member) return next('route');
    // Otherwise, call the next middleware and fetch
    // the member's data.
    req.member = member;
    next();
  });
}, function(req, res, next) {
  getMemberData(req.member, function(err, data) {
    if (err) return next(err);
    // If this member has no data, don't bother
    // parsing it. Just go render the page now.
    if (!data) return next('route');
    // Otherwise, call the next middleware and parse
    // the member's data. THEN render the page.
    req.member.data = data;
    next();
  });
}, function(req, res, next) {
  req.member.parsedData = parseMemberData(req.member.data);
  next();
});

app.get('/bananas', function(req, res, next) {
  renderBananas(req.member);
});
```

Dans cet exemple, chaque fonction de middleware serait soit dans son propre fichier, soit dans une variable ailleurs dans le fichier, de manière à pouvoir être réutilisée dans d'autres itinéraires.

La gestion des erreurs

Les documents de base peuvent être trouvés **ici**

```
app.get('/path/:id(\\d+)', function (req, res, next) { // please note: "next" is passed
    if (req.params.id == 0) // validate param
        return next(new Error('Id is 0')); // go to first Error handler, see below

    // Catch error on sync operation
    var data;
    try {
        data = JSON.parse('/file.json');
    } catch (err) {
        return next(err);
    }

    // If some critical error then stop application
    if (!data)
        throw new Error('Smth wrong');

    // If you need send extra info to Error handler
    // then send custom error (see Appendix B)
    if (smth)
        next(new MyError('smth wrong', arg1, arg2))

    // Finish request by res.render or res.end
    res.status(200).end('OK');
});

// Be sure: order of app.use have matter
// Error handler
app.use(function(err, req, res, next)) {
    if (smth-check, e.g. req.url != 'POST')
        return next(err); // go-to Error handler 2.

    console.log(req.url, err.message);

    if (req.xhr) // if req via ajax then send json else render error-page
        res.json(err);
    else
        res.render('error.html', {error: err.message});
});

// Error handler 2
app.use(function(err, req, res, next)) {
    // do smth here e.g. check that error is MyError
    if (err instanceof MyError) {
        console.log(err.message, err.arg1, err.arg2);
    }
    ...
    res.end();
});
```

Annexe A

```
// "In Express, 404 responses are not the result of an error,
// so the error-handler middleware will not capture them."
// You can change it.
app.use(function(req, res, next) {
    next(new Error(404));
});
```

Appendice B

```
// How to define custom error
var util = require('util');
...
function MyError(message, arg1, arg2) {
    this.message = message;
    this.arg1 = arg1;
    this.arg2 = arg2;
    Error.captureStackTrace(this, MyError);
}
util.inherits(MyError, Error);
MyError.prototype.name = 'MyError';
```

Hook: Comment exécuter du code avant toute demande et après toute res

`app.use()` et le middleware peuvent être utilisés pour "before" et une combinaison des événements close et finish peut être utilisée pour "after".

```
app.use(function (req, res, next) {
    function afterResponse() {
        res.removeListener('finish', afterResponse);
        res.removeListener('close', afterResponse);

        // actions after response
    }
    res.on('finish', afterResponse);
    res.on('close', afterResponse);

    // action before request
    // eventually calling `next()`
    next();
});
...
app.use(app.router);
```

Un exemple de ceci est le middleware de l' enregistreur , qui sera ajouté au journal après la réponse par défaut.

Assurez-vous simplement que ce "middleware" est utilisé avant `app.router` car l'ordre compte.

Le message original est **ici**

Gestion des requêtes POST

Tout comme vous gérez les demandes d'obtention dans Express avec la méthode app.get, vous pouvez utiliser la méthode app.post pour gérer les demandes de publication.

Mais avant de pouvoir traiter les requêtes POST, vous devrez utiliser le middleware `body-parser` . Il analyse simplement le corps des requêtes `POST` , `PUT` , `DELETE` et autres.

`Body-Parser` middleware `Body-Parser` analyse le corps de la requête et le transforme en objet disponible dans `req.body`

```
var bodyParser = require('body-parser');

const express = require('express');

const app = express();

// Parses the body for POST, PUT, DELETE, etc.
app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.post('/post-data-here', function(req, res, next){

    console.log(req.body); // req.body contains the parsed body of the request.

});

app.listen(8080, 'localhost');
```

Définition de cookies avec un cookie-parser

Voici un exemple de configuration et de lecture de cookies à l'aide du module cookie-parser :

```
var express = require('express');
var cookieParser = require('cookie-parser'); // module for parsing cookies
var app = express();
app.use(cookieParser());

app.get('/setcookie', function(req, res){
    // setting cookies
    res.cookie('username', 'john doe', { maxAge: 900000, httpOnly: true });
    return res.send('Cookie has been set');
});

app.get('/getcookie', function(req, res) {
    var username = req.cookies['username'];
    if (username) {
        return res.send(username);
    }

    return res.send('No cookie found');
});

app.listen(3000);
```

Middleware personnalisé dans Express

Dans Express, vous pouvez définir des middlewares pouvant être utilisés pour vérifier les requêtes ou définir des en-têtes en réponse.

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

Exemple

Le code suivant ajoute l' `user` à l'objet de requête et le transmet au prochain itinéraire

correspondant.

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

//each request will pass through it
app.use(function(req, res, next){
    req.user = 'testuser';
    next();    // it will pass the control to next matching route
});

app.get('/', function(req, res){
    var user = req.user;
    console.log(user); // testuser
    return res.send(user);
});

app.listen(3000);
```

Gestion des erreurs dans Express

Dans Express, vous pouvez définir un gestionnaire d'erreurs unifié pour la gestion des erreurs survenues dans l'application. Définissez ensuite le gestionnaire à la fin de toutes les routes et du code logique.

Exemple

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

//GET /names/john
app.get('/names/:name', function(req, res, next){
    if (req.params.name == 'john'){
        return res.send('Valid Name');
    } else{
        next(new Error('Not valid name'));    //pass to error handler
    }
});

//error handler
app.use(function(err, req, res, next){
    console.log(err.stack);    // e.g., Not valid name
    return res.status(500).send('Internal Server Occured');
});

app.listen(3000);
```

Ajout de middleware

Les fonctions de middleware sont des fonctions qui ont accès à l'objet de requête (req), à l'objet de réponse (res) et à la fonction de middleware suivante dans le cycle demande-réponse de l'application.

Les fonctions middleware peuvent exécuter n'importe quel code, apporter des modifications aux objets `res` et `req` , mettre fin au cycle de réponse et appeler le prochain middleware.

Un exemple très courant de middleware est le module `cors` . Pour ajouter le support CORS, installez-le simplement, exigez-le et mettez cette ligne:

```
app.use(cors());
```

avant tout routeurs ou fonctions de routage.

Bonjour le monde

Ici, nous créons un serveur de base hello world en utilisant Express. Itinéraires:

- '/'
- '/ wiki'

Et pour le reste donnera "404", c'est-à-dire la page introuvable.

```
'use strict';

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

var app = require('express')();
    app.listen(port);

app.get('/',(req,res)=>res.send('HelloWorld!'));
app.get('/wiki',(req,res)=>res.send('This is wiki page.'));
app.use((req,res)=>res.send('404-PageNotFound'));
```

Remarque: Nous avons placé 404 route comme dernier itinéraire car Express stocke les itinéraires dans l'ordre et les traite pour chaque requête de manière séquentielle.

Lire Applications Web avec Express en ligne: https://riptutorial.com/fr/node-js/topic/483/applications-web-avec-express

Chapitre 6: Arrêt gracieux

Examples

Arrêt gracieux - SIGTERM

En utilisant **server.close ()** et **process.exit ()** , nous pouvons intercepter l'exception du serveur et procéder à un arrêt en douceur.

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

var server = http.createServer(function (req, res) {
  setTimeout(function () { //simulate a long request
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
  }, 4000);
}).listen(9090, function (err) {
  console.log('listening http://localhost:9090/');
  console.log('pid is ' + process.pid);
});

process.on('SIGTERM', function () {
  server.close(function () {
    process.exit(0);
  });
});
```

Lire Arrêt gracieux en ligne: https://riptutorial.com/fr/node-js/topic/5996/arret-gracieux

Chapitre 7: Async / En attente

Introduction

Async / wait est un ensemble de mots-clés permettant d'écrire du code asynchrone de manière procédurale sans avoir à se fier aux callbacks (*callback hell*) ou `.then().then().then()` -chaining (`.then().then().then()`).

Cela fonctionne en utilisant le mot-clé `await` pour suspendre l'état d'une fonction asynchrone, jusqu'à la résolution d'une promesse, et en utilisant le mot-clé `async` pour déclarer ces fonctions asynchrones, qui renvoient une promesse.

Async / waiting est disponible par défaut sur node.js 8 ou 7 en utilisant le drapeau `--harmony-async-await` .

Examples

Fonctions asynchrones avec gestion des erreurs de try-catch

L'une des meilleures caractéristiques de la syntaxe asynchrone / d'attente est qu'un style de codage try-catch standard est possible, tout comme vous écrivez du code synchrone.

```
const myFunc = async (req, res) => {
  try {
    const result = await somePromise();
  } catch (err) {
    // handle errors here
  }
});
```

Voici un exemple avec Express et promise-mysql:

```
router.get('/flags/:id', async (req, res) => {

  try {

    const connection = await pool.createConnection();

    try {
      const sql = `SELECT f.id, f.width, f.height, f.code, f.filename
                   FROM flags f
                   WHERE f.id = ?
                   LIMIT 1`;
      const flags = await connection.query(sql, req.params.id);
      if (flags.length === 0)
        return res.status(404).send({ message: 'flag not found' });

      return res.send({ flags[0] });

    } finally {
```

```
      pool.releaseConnection(connection);
    }

  } catch (err) {
    // handle errors here
  }
});
```

Comparaison entre promesses et async / en attente

Fonction utilisant des promesses:

```
function myAsyncFunction() {
    return aFunctionThatReturnsAPromise()
           // doSomething is a sync function
           .then(result => doSomething(result))
           .catch(handleError);
}
```

Alors, voici quand Async / Await entre en action pour que notre fonction soit plus propre:

```
async function myAsyncFunction() {
  let result;

  try {
      result = await aFunctionThatReturnsAPromise();
  } catch (error) {
      handleError(error);
  }

  // doSomething is a sync function
  return doSomething(result);
}
```

Le mot-clé `async` serait donc similaire à write `return new Promise((resolve, reject) => {...}`.

Et `await` même façon pour obtenir votre résultat, `then` rappel.

Je laisse ici un gif assez bref qui ne laissera aucun doute après l'avoir vu:

GIF

Progression des rappels

Au début, il y avait des rappels, et les rappels étaient corrects:

```
const getTemperature = (callback) => {
  http.get('www.temperature.com/current', (res) => {
    callback(res.data.temperature)
  })
}

const getAirPollution = (callback) => {
  http.get('www.pollution.com/current', (res) => {
```

```
    callback(res.data.pollution)
  });
}

getTemperature(function(temp) {
  getAirPollution(function(pollution) {
    console.log(`the temp is ${temp} and the pollution is ${pollution}.`)
    // The temp is 27 and the pollution is 0.5.
  })
})
```

Mais il y avait quelques problèmes vraiment frustrants avec les rappels, nous avons donc tous commencé à utiliser des promesses.

```
const getTemperature = () => {
  return new Promise((resolve, reject) => {
    http.get('www.temperature.com/current', (res) => {
      resolve(res.data.temperature)
    })
  })
}

const getAirPollution = () => {
  return new Promise((resolve, reject) => {
    http.get('www.pollution.com/current', (res) => {
      resolve(res.data.pollution)
    })
  })
}

getTemperature()
.then(temp => console.log(`the temp is ${temp}`))
.then(() => getAirPollution())
.then(pollution => console.log(`and the pollution is ${pollution}`))
// the temp is 32
// and the pollution is 0.5
```

C'était un peu mieux. Enfin, nous avons trouvé async / waiting. Qui utilise encore des promesses sous le capot.

```
const temp = await getTemperature()
const pollution = await getAirPollution()
```

Arrête l'exécution à l'attente

Si la promesse ne renvoie rien, la tâche asynchrone peut être terminée en utilisant `await` .

```
try{
    await User.findByIdAndUpdate(user._id, {
        $push: {
            tokens: token
        }
    }).exec()
}catch(e){
    handleError(e)
}
```

Lire Async / En attente en ligne: https://riptutorial.com/fr/node-js/topic/6729/async---en-attente

Chapitre 8: async.js

Syntaxe

- **Chaque rappel doit être écrit avec cette syntaxe:**
- fonction callback (err, result [, arg1 [, ...]])
- **De cette façon, vous êtes obligé de renvoyer l'erreur en premier et vous ne pouvez pas ignorer leur traitement ultérieurement. `null` est la convention en l'absence d'erreurs**
- callback (null, myResult);
- **Vos rappels peuvent contenir plus d'arguments que d' *erreurs* et de *résultats* , mais ne sont utiles que pour un ensemble spécifique de fonctions (cascade, seq, ...)**
- callback (null, myResult, myCustomArgument);
- **Et, bien sûr, envoyer des erreurs. Vous devez le faire et gérer les erreurs (ou au moins les enregistrer).**
- rappel (err);

Examples

Parallèle: multi-tâches

async.parallel (tâches, afterTasksCallback) exécutera un ensemble de tâches en parallèle et **attendra la fin de toutes les tâches** (signalées par l'appel de la fonction de **rappel**).

Lorsque les tâches sont terminées, *async* appelle le rappel principal avec toutes les erreurs et tous les résultats des tâches.

```
function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfLongTime');
  }, 1000);
```

```
}

async.parallel([
    shortTimeFunction,
    mediumTimeFunction,
    longTimeFunction
  ],
  function(err, results) {
    if (err) {
      return console.error(err);
    }

    console.log(results);
  });
```

Résultat: `["resultOfShortTime", "resultOfMediumTime", "resultOfLongTime"]` .

Appelez `async.parallel()` avec un objet

Vous pouvez remplacer le paramètre tableau de *tâches* par un objet. Dans ce cas, les résultats seront également un objet **avec les mêmes clés que les tâches** .

C'est très utile pour calculer certaines tâches et trouver facilement chaque résultat.

```
async.parallel({
    short: shortTimeFunction,
    medium: mediumTimeFunction,
    long: longTimeFunction
  },
  function(err, results) {
    if (err) {
      return console.error(err);
    }

    console.log(results);
  });
```

Résultat: `{short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"}` .

Résolution de plusieurs valeurs

Chaque fonction parallèle reçoit un rappel. Ce rappel peut renvoyer une erreur comme premier argument ou des valeurs de réussite après cela. Si un rappel est transmis à plusieurs valeurs de réussite, ces résultats sont renvoyés sous forme de tableau.

```
async.parallel({
    short: function shortTimeFunction(callback) {
      setTimeout(function() {
        callback(null, 'resultOfShortTime1', 'resultOfShortTime2');
      }, 200);
    },
    medium: function mediumTimeFunction(callback) {
```

```
      setTimeout(function() {
        callback(null, 'resultOfMediumTime1', 'resultOfMeiumTime2');
      }, 500);
    }
  },
  function(err, results) {
    if (err) {
      return console.error(err);
    }

    console.log(results);
  });
```

Résultat :

```
{
    short: ["resultOfShortTime1", "resultOfShortTime2"],
    medium: ["resultOfMediumTime1", "resultOfMediumTime2"]
}
```

.

Série: mono-tâche indépendante

async.series (tasks, afterTasksCallback) exécutera un ensemble de tâches. Chaque tâche est exécutée **après l'autre** . **Si une tâche échoue, *async* arrête immédiatement l'exécution et saute dans le rappel principal** .

Lorsque les tâches sont terminées avec succès, *async* appelle le rappel "maître" avec toutes les erreurs et tous les résultats des tâches.

```
function shortTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfShortTime');
  }, 200);
}

function mediumTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfMediumTime');
  }, 500);
}

function longTimeFunction(callback) {
  setTimeout(function() {
    callback(null, 'resultOfLongTime');
  }, 1000);
}

async.series([
    mediumTimeFunction,
    shortTimeFunction,
    longTimeFunction
  ],
  function(err, results) {
    if (err) {
```

```
      return console.error(err);
    }

    console.log(results);
  });
```

Résultat: `["resultOfMediumTime", "resultOfShortTime", "resultOfLongTime"]` .

Appelez `async.series()` avec un objet

Vous pouvez remplacer le paramètre tableau de *tâches* par un objet. Dans ce cas, les résultats seront également un objet **avec les mêmes clés que les tâches** .

C'est très utile pour calculer certaines tâches et trouver facilement chaque résultat.

```
async.series({
    short: shortTimeFunction,
    medium: mediumTimeFunction,
    long: longTimeFunction
  },
  function(err, results) {
    if (err) {
      return console.error(err);
    }

    console.log(results);
  });
```

Résultat: `{short: "resultOfShortTime", medium: "resultOfMediumTime", long: "resultOfLongTime"}` .

Cascade: mono-tâche dépendante

async.waterfall (tasks, afterTasksCallback) exécutera un ensemble de tâches. Chaque tâche est exécutée **après l'autre et le résultat d'une tâche est transmis à la tâche suivante** . En tant que *async.series ()* , si une tâche échoue, *async* arrête l'exécution et appelle immédiatement le rappel principal.

Lorsque les tâches sont terminées avec succès, *async* appelle le rappel "maître" avec toutes les erreurs et tous les résultats des tâches.

```
function getUserRequest(callback) {
  // We simulate the request with a timeout
  setTimeout(function() {
    var userResult = {
      name : 'Aamu'
    };

    callback(null, userResult);
  }, 500);
}

function getUserFriendsRequest(user, callback) {
```

```
  // Another request simulate with a timeout
  setTimeout(function() {
    var friendsResult = [];

    if (user.name === "Aamu"){
        friendsResult = [{
          name : 'Alice'
        }, {
          name: 'Bob'
        }];
    }

    callback(null, friendsResult);
  }, 500);
}

async.waterfall([
    getUserRequest,
    getUserFriendsRequest
  ],
  function(err, results) {
    if (err) {
      return console.error(err);
    }

    console.log(JSON.stringify(results));
  });
```

Résultat: le `results` contient le deuxième paramètre de rappel de la dernière fonction de la cascade, qui est `friendsResult` dans ce cas.

async.times (gérer mieux la boucle)

Pour exécuter une fonction dans une boucle en Node.js, il est bon d'utiliser une `for` boucle pour les boucles courtes. Mais la boucle est longue, utiliser `for` loop augmentera le temps de traitement, ce qui pourrait entraîner le blocage du processus de noeud. Dans de tels scénarios, vous pouvez utiliser: **asycn.times**

```
function recursiveAction(n, callback)
{
    //do whatever want to do repeatedly
    callback(err, result);
}
async.times(5, function(n, next) {
    recursiveAction(n, function(err, result) {
        next(err, result);
    });
}, function(err, results) {
    // we should now have 5 result
});
```

Ceci est appelé en parallèle. Lorsque nous voulons l'appeler un à la fois, utilisez: **async.timesSeries**

async.each (pour gérer efficacement le tableau de données)

Lorsque nous voulons gérer un tableau de données, il est préférable d'utiliser **async.each** . Lorsque nous voulons effectuer quelque chose avec toutes les données et que nous voulons obtenir le rappel final une fois que tout est fait, cette méthode sera utile. Ceci est géré en parallèle.

```
function createUser(userName, callback)
{
    //create user in db
    callback(null)//or error based on creation
}

var arrayOfData = ['Ritu', 'Sid', 'Tom'];
async.each(arrayOfData, function(eachUserName, callback) {

    // Perform operation on each user.
    console.log('Creating user '+eachUserName);
    //Returning callback is must. Else it wont get the final callback, even if we miss to
return one callback
    createUser(eachUserName, callback);

}, function(err) {
    //If any of the user creation failed may throw error.
    if( err ) {
      // One of the iterations produced an error.
      // All processing will now stop.
      console.log('unable to create user');
    } else {
      console.log('All user created successfully');
    }
});
```

Pour faire un à la fois, vous pouvez utiliser **async.eachSeries**

async.series (Pour gérer les événements un par un)

/ Dans async.series, toutes les fonctions sont exécutées en série et les sorties consolidées de chaque fonction sont passées au rappel final. par exemple

var async = require ('async'); async.series ([function (callback) {console.log ('First Execute ..'); callback (null, 'userPersonalData');}, function (callback) {console.log ('Second Execute ..'); callback (null, 'userDependentData');}], fonction (err, result) {console.log (result);});

//Sortie:

First Execute .. Second Execute .. ['userPersonalData', 'userDependentData'] // résultat

Lire async.js en ligne: https://riptutorial.com/fr/node-js/topic/3972/async-js

Chapitre 9: Authentification Windows sous node.js

Remarques

Il existe plusieurs autres APIS Active Directory, tels que `activedirectory2` et `adldap` .

Examples

Utiliser activedirectory

L'exemple ci-dessous provient des documents complets, disponibles ici (GitHub) ou ici (NPM) .

Installation

```
npm install --save activedirectory
```

Usage

```
// Initialize
var ActiveDirectory = require('activedirectory');
var config = {
    url: 'ldap://dc.domain.com',
    baseDN: 'dc=domain,dc=com'
};
var ad = new ActiveDirectory(config);
var username = 'john.smith@domain.com';
var password = 'password';
// Authenticate
ad.authenticate(username, password, function(err, auth) {
    if (err) {
        console.log('ERROR: '+JSON.stringify(err));
        return;
    }
    if (auth) {
        console.log('Authenticated!');
    }
    else {
        console.log('Authentication failed!');
    }
});
```

Lire Authentification Windows sous node.js en ligne: https://riptutorial.com/fr/node-js/topic/10612/authentification-windows-sous-node-js

Chapitre 10: Base de données (MongoDB avec Mongoose)

Examples

Connexion Mongoose

Assurez-vous d'avoir mongodb en premier! `mongod --dbpath data/`

package.json

```
"dependencies": {
    "mongoose": "^4.5.5",
}
```

server.js (ECMA 6)

```
import mongoose from 'mongoose';

mongoose.connect('mongodb://localhost:27017/stackoverflow-example');
const db = mongoose.connection;
db.on('error', console.error.bind(console, 'DB connection error!'));
```

server.js (ECMA 5.1)

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/stackoverflow-example');
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'DB connection error!'));
```

Modèle

Définissez votre (vos) modèle (s):

app / models / user.js (ECMA 6)

```
import mongoose from 'mongoose';

const userSchema = new mongoose.Schema({
    name: String,
    password: String
});

const User = mongoose.model('User', userSchema);

export default User;
```

app / model / user.js (ECMA 5.1)

```
var mongoose = require('mongoose');

var userSchema = new mongoose.Schema({
    name: String,
    password: String
});

var User = mongoose.model('User', userSchema);

module.exports = User
```

Insérer des données

ECMA 6:

```
const user = new User({
   name: 'Stack',
   password: 'Overflow',
   }) ;

user.save((err) => {
    if (err) throw err;

    console.log('User saved!');
});
```

ECMA5.1:

```
var user = new User({
   name: 'Stack',
   password: 'Overflow',
   }) ;

user.save(function (err) {
    if (err) throw err;

    console.log('User saved!');
});
```

Lire des données

ECMA6:

```
User.findOne({
    name: 'stack'
}, (err, user) => {
    if (err) throw err;

    if (!user) {
        console.log('No user was found');
    } else {
        console.log('User was found');
    }
```

```
});
```

ECMA5.1:

```
User.findOne({
    name: 'stack'
}, function (err, user) {
    if (err) throw err;

    if (!user) {
        console.log('No user was found');
    } else {
        console.log('User was found');
    }
});
```

Lire Base de données (MongoDB avec Mongoose) en ligne: https://riptutorial.com/fr/node-js/topic/6411/base-de-donnees--mongodb-avec-mongoose-

Chapitre 11: Bibliothèque Mongoose

Examples

Connectez-vous à MongoDB en utilisant Mongoose

Tout d'abord, installez Mongoose avec:

```
npm install mongoose
```

Ensuite, ajoutez-le à `server.js` tant que dépendances:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
```

Ensuite, créez le schéma de base de données et le nom de la collection:

```
var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});
```

Créez un modèle et connectez-vous à la base de données:

```
var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');
```

Ensuite, démarrez MongoDB et exécutez `server.js` utilisant `node server.js`

Pour vérifier si nous avons réussi à nous connecter à la base de données, nous pouvons utiliser les événements `open` , `error` de l'objet `mongoose.connection` .

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));
db.once('open', function() {
  // we're connected!
});
```

Enregistrer les données sur MongoDB en utilisant les routes Mongoose et Express.js

Installer

D'abord, installez les paquets nécessaires avec:

```
npm install express cors mongoose
```

Code

Ensuite, ajoutez des dépendances à votre fichier `server.js` , créez le schéma de base de données et le nom de la collection, créez un serveur Express.js et connectez-vous à MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
    console.log('Node.js listening on port ' + port);
});
```

Ajoutez maintenant les routes Express.js que nous utiliserons pour écrire les données:

```
app.get('/save/:query', cors(), function(req, res) {
    var query = req.params.query;

    var savedata = new Model({
        'request': query,
        'time': Math.floor(Date.now() / 1000) // Time of save the data in unix timestamp
format
    }).save(function(err, result) {
        if (err) throw err;

        if(result) {
            res.json(result)
        }
    })
})
```

Ici, la variable de `query` sera le paramètre `<query>` de la requête HTTP entrante, qui sera enregistrée dans MongoDB:

```
var savedata = new Model({
    'request': query,
    //...
```

Si une erreur survient lors de la tentative d'écriture sur MongoDB, vous recevrez un message d'erreur sur la console. Si tout est réussi, vous verrez les données enregistrées au format JSON sur la page.

```
//...
}).save(function(err, result) {
    if (err) throw err;

    if(result) {
        res.json(result)
    }
})
//...
```

Maintenant, vous devez démarrer MongoDB et exécuter votre fichier `server.js` en utilisant `node server.js` .

Usage

Pour l'utiliser pour enregistrer des données, accédez à l'URL suivante dans votre navigateur:

```
http://localhost:8080/save/<query>
```

Où `<query>` est la nouvelle requête que vous souhaitez enregistrer.

Exemple:

```
http://localhost:8080/save/JavaScript%20is%20Awesome
```

Sortie au format JSON:

```
{
    __v: 0,
    request: "JavaScript is Awesome",
    time: 1469411348,
    _id: "57957014b93bc8640f2c78c4"
}
```

Rechercher des données dans MongoDB en utilisant les routes Mongoose et Express.js

Installer

D'abord, installez les paquets nécessaires avec:

```
npm install express cors mongoose
```

Code

Ensuite, ajoutez des dépendances à `server.js` , créez le schéma de base de données et le nom de la collection, créez un serveur Express.js et connectez-vous à MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
    console.log('Node.js listening on port ' + port);
});
```

Ajoutez maintenant les routes Express.js que nous utiliserons pour interroger les données:

```
app.get('/find/:query', cors(), function(req, res) {
    var query = req.params.query;

    Model.find({
        'request': query
    }, function(err, result) {
        if (err) throw err;
        if (result) {
            res.json(result)
        } else {
            res.send(JSON.stringify({
                error : 'Error'
            }))
        }
    })
})
```

Supposons que les documents suivants figurent dans la collection du modèle:

```
{
        "_id" : ObjectId("578abe97522ad414b8eeb55a"),
        "request" : "JavaScript is Awesome",
        "time" : 1468710551
}
{
        "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
```

```
        "request" : "JavaScript is Awesome",
        "time" : 1468710555
}
{
        "_id" : ObjectId("578abea0522ad414b8eeb55c"),
        "request" : "JavaScript is Awesome",
        "time" : 1468710560
}
```

Et le but est de trouver et d'afficher tous les documents contenant `"JavaScript is Awesome"` sous la clé `"request"` .

Pour cela, démarrez MongoDB et exécutez `server.js` avec `node server.js` :

Usage

Pour l'utiliser pour rechercher des données, accédez à l'URL suivante dans un navigateur:

```
http://localhost:8080/find/<query>
```

Où `<query>` est la requête de recherche.

Exemple:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Sortie:

```
[{
    _id: "578abe97522ad414b8eeb55a",
    request: "JavaScript is Awesome",
    time: 1468710551,
    __v: 0
},
{
    _id: "578abe9b522ad414b8eeb55b",
    request: "JavaScript is Awesome",
    time: 1468710555,
    __v: 0
},
{
    _id: "578abea0522ad414b8eeb55c",
    request: "JavaScript is Awesome",
    time: 1468710560,
    __v: 0
}]
```

Recherche de données dans MongoDB à l'aide de Mongoose, Express.js Routes et $ text Operator

Installer

D'abord, installez les paquets nécessaires avec:

```
npm install express cors mongoose
```

Code

Ensuite, ajoutez des dépendances à `server.js` , créez le schéma de base de données et le nom de la collection, créez un serveur Express.js et connectez-vous à MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
    console.log('Node.js listening on port ' + port);
});
```

Ajoutez maintenant les routes Express.js que nous utiliserons pour interroger les données:

```
app.get('/find/:query', cors(), function(req, res) {
    var query = req.params.query;

    Model.find({
        'request': query
    }, function(err, result) {
        if (err) throw err;
        if (result) {
            res.json(result)
        } else {
            res.send(JSON.stringify({
                error : 'Error'
            }))
        }
    })
})
```

Supposons que les documents suivants figurent dans la collection du modèle:

```
{
        "_id" : ObjectId("578abe97522ad414b8eeb55a"),
        "request" : "JavaScript is Awesome",
        "time" : 1468710551
}
{
        "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
        "request" : "JavaScript is Awesome",
        "time" : 1468710555
}
{
        "_id" : ObjectId("578abea0522ad414b8eeb55c"),
        "request" : "JavaScript is Awesome",
        "time" : 1468710560
}
```

Et que le but est de trouver et d'afficher tous les documents ne contenant que le mot `"JavaScript"` sous la clé `"request"` .

Pour ce faire, créez d'abord un *index de texte* pour `"request"` dans la collection. Pour cela, ajoutez le code suivant à `server.js` :

```
schemaName.index({ request: 'text' });
```

Et remplacer:

```
    Model.find({
        'request': query
    }, function(err, result) {
```

Avec:

```
Model.find({
    $text: {
        $search: query
    }
}, function(err, result) {
```

Ici, nous utilisons `$search` opérateurs `$text` et `$search` MongoDB pour rechercher tous les documents de la collection `collectionName` qui contient au moins un mot de la requête de recherche spécifiée.

Usage

Pour l'utiliser pour rechercher des données, accédez à l'URL suivante dans un navigateur:

```
http://localhost:8080/find/<query>
```

Où `<query>` est la requête de recherche.

Exemple:

```
http://localhost:8080/find/JavaScript
```

Sortie:

```
[{
    _id: "578abe97522ad414b8eeb55a",
    request: "JavaScript is Awesome",
    time: 1468710551,
    __v: 0
},
{
    _id: "578abe9b522ad414b8eeb55b",
    request: "JavaScript is Awesome",
    time: 1468710555,
    __v: 0
},
{
    _id: "578abea0522ad414b8eeb55c",
    request: "JavaScript is Awesome",
    time: 1468710560,
    __v: 0
}]
```

Index dans les modèles.

MongoDB prend en charge les index secondaires. Dans Mongoose, nous définissons ces index dans notre schéma. La définition d'index au niveau du schéma est nécessaire lorsque vous devez créer des index composés.

Connexion Mongoose

```
var strConnection = 'mongodb://localhost:27017/dbName';
var db = mongoose.createConnection(strConnection)
```

Créer un schéma de base

```
var Schema = require('mongoose').Schema;
var usersSchema = new Schema({
    username: {
        type: String,
        required: true,
        unique: true
    },
    email: {
        type: String,
        required: true
    },
    password: {
        type: String,
        required: true
```

```
    },
    created: {
        type: Date,
        default: Date.now
    }
});

var usersModel = db.model('users', usersSchema);
module.exports = usersModel;
```

Par défaut, mongoose ajoute deux nouveaux champs à notre modèle, même s'ils ne sont pas définis dans le modèle. Ces champs sont:

_id

Mongoose attribue à chacun de vos schémas un champ _id par défaut si l'un d'entre eux n'est pas transmis au constructeur de schéma. Le type attribué est un ObjectId qui coïncide avec le comportement par défaut de MongoDB. Si vous ne voulez pas ajouter d'id à votre schéma, vous pouvez le désactiver en utilisant cette option.

```
var usersSchema = new Schema({
    username: {
        type: String,
        required: true,
        unique: true
    }, {
        _id: false
});
```

__v ou versionKey

La versionKey est une propriété définie sur chaque document lors de sa création par Mongoose. Cette valeur de clé contient la révision interne du document. Le nom de cette propriété de document est configurable.

Vous pouvez facilement désactiver ce champ dans la configuration du modèle:

```
var usersSchema = new Schema({
    username: {
        type: String,
        required: true,
        unique: true
    }, {
    versionKey: false
});
```

Index composés

Nous pouvons créer d’autres index que ceux créés par Mongoose.

```
usersSchema.index({username: 1 });
usersSchema.index({email: 1 });
```

Dans ce cas, notre modèle a deux autres index, un pour le nom d'utilisateur du champ et un autre pour le champ email. Mais nous pouvons créer des index composés.

```
usersSchema.index({username: 1, email: 1 });
```

Impact sur la performance de l'index

Par défaut, mongoose appelle toujours séquentiellement assureIndex pour chaque index et émet un événement 'index' sur le modèle lorsque tous les appels à efficientIndex ont abouti ou en cas d'erreur.

Dans MongoDB EnsureIndex est obsolète depuis la version 3.0.0, est maintenant un alias pour createIndex.

Il est recommandé de désactiver le comportement en définissant l'option autoIndex de votre schéma sur false ou globalement sur la connexion en définissant l'option config.autoIndex sur false.

```
usersSchema.set('autoIndex', false);
```

Fonctions utiles de Mongoose

Mongoose contient des fonctions intégrées basées sur `find()` .

```
doc.find({'some.value':5},function(err,docs){
    //returns array docs
});

doc.findOne({'some.value':5},function(err,doc){
    //returns document doc
});

doc.findById(obj._id,function(err,doc){
    //returns document doc
});
```

trouver des données dans mongodb en utilisant des promesses

Installer

D'abord, installez les paquets nécessaires avec:

```
npm install express cors mongoose
```

Code

Ensuite, ajoutez des dépendances à `server.js` , créez le schéma de base de données et le nom

de la collection, créez un serveur Express.js et connectez-vous à MongoDB:

```
var express = require('express');
var cors = require('cors'); // We will use CORS to enable cross origin domain requests.
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var app = express();

var schemaName = new Schema({
    request: String,
    time: Number
}, {
    collection: 'collectionName'
});

var Model = mongoose.model('Model', schemaName);
mongoose.connect('mongodb://localhost:27017/dbName');

var port = process.env.PORT || 8080;
app.listen(port, function() {
    console.log('Node.js listening on port ' + port);
});

app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});

app.use(function(req, res, next) {
  res.status(404).send('Sorry cant find that!');
});
```

Ajoutez maintenant les routes Express.js que nous utiliserons pour interroger les données:

```
app.get('/find/:query', cors(), function(req, res, next) {
    var query = req.params.query;

    Model.find({
        'request': query
    })
    .exec() //remember to add exec, queries have a .then attribute but aren't promises
    .then(function(result) {
        if (result) {
            res.json(result)
        } else {
            next() //pass to 404 handler
        }
    })
    .catch(next) //pass to error handler
})
```

Supposons que les documents suivants figurent dans la collection du modèle:

```
{
        "_id" : ObjectId("578abe97522ad414b8eeb55a"),
        "request" : "JavaScript is Awesome",
        "time" : 1468710551
```

```
}
{
        "_id" : ObjectId("578abe9b522ad414b8eeb55b"),
        "request" : "JavaScript is Awesome",
        "time" : 1468710555
}
{
        "_id" : ObjectId("578abea0522ad414b8eeb55c"),
        "request" : "JavaScript is Awesome",
        "time" : 1468710560
}
```

Et le but est de trouver et d'afficher tous les documents contenant `"JavaScript is Awesome"` sous la clé `"request"` .

Pour cela, démarrez MongoDB et exécutez `server.js` avec `node server.js` :

Usage

Pour l'utiliser pour rechercher des données, accédez à l'URL suivante dans un navigateur:

```
http://localhost:8080/find/<query>
```

Où `<query>` est la requête de recherche.

Exemple:

```
http://localhost:8080/find/JavaScript%20is%20Awesome
```

Sortie:

```
[{
    _id: "578abe97522ad414b8eeb55a",
    request: "JavaScript is Awesome",
    time: 1468710551,
    __v: 0
},
{
    _id: "578abe9b522ad414b8eeb55b",
    request: "JavaScript is Awesome",
    time: 1468710555,
    __v: 0
},
{
    _id: "578abea0522ad414b8eeb55c",
    request: "JavaScript is Awesome",
    time: 1468710560,
    __v: 0
}]
```

Lire Bibliothèque Mongoose en ligne: https://riptutorial.com/fr/node-js/topic/3486/bibliotheque-mongoose

Chapitre 12: Bluebird Promises

Examples

Conversion de la bibliothèque nodeback en promises

```
const Promise = require('bluebird'),
      fs = require('fs')

Promise.promisifyAll(fs)

// now you can use promise based methods on 'fs' with the Async suffix
fs.readFileAsync('file.txt').then(contents => {
  console.log(contents)
}).catch(err => {
  console.error('error reading', err)
})
```

Promesses fonctionnelles

Exemple de carte:

```
Promise.resolve([ 1, 2, 3 ]).map(el => {
   return Promise.resolve(el * el) // return some async operation in real world
})
```

Exemple de filtre:

```
Promise.resolve([ 1, 2, 3 ]).filter(el => {
  return Promise.resolve(el % 2 === 0) // return some async operation in real world
}).then(console.log)
```

Exemple de réduire:

```
Promise.resolve([ 1, 2, 3 ]).reduce((prev, curr) => {
  return Promise.resolve(prev + curr) // return some async operation in real world
}).then(console.log)
```

Coroutines (Générateurs)

```
const promiseReturningFunction = Promise.coroutine(function* (file) {
  const data = yield fs.readFileAsync(file) // this returns a Promise and resolves to the file
contents

  return data.toString().toUpperCase()
})

promiseReturningFunction('file.txt').then(console.log)
```

Élimination automatique des ressources (Promise.using)

```
function somethingThatReturnsADisposableResource() {
  return getSomeResourceAsync(...).disposer(resource => {
    resource.dispose()
  })
}

Promise.using(somethingThatReturnsADisposableResource(), resource => {
  // use the resource here, the disposer will automatically close it when Promise.using exits
})
```

Exécution en série

```
Promise.resolve([1, 2, 3])
  .mapSeries(el => Promise.resolve(el * el)) // in real world, use Promise returning async
function
  .then(console.log)
```

Lire Bluebird Promises en ligne: https://riptutorial.com/fr/node-js/topic/6728/bluebird-promises

Chapitre 13: Bon style de codage

Remarques

Je recommanderais à un débutant de commencer avec ce style de codage. Et si quelqu'un peut suggérer un meilleur moyen (ps j'ai opté pour cette technique et travaille efficacement pour moi dans une application utilisée par plus de 100 000 utilisateurs), n'hésitez pas à faire des suggestions. TIA.

Examples

Programme de base pour l'inscription

Dans cet exemple, il sera expliqué de diviser le code **node.js** en différents **modules / dossiers** pour une meilleure compréhension. Suivre cette technique permet aux autres développeurs de mieux comprendre le code, car il peut directement se référer au fichier concerné au lieu de parcourir tout le code. L'utilisation principale est que lorsque vous travaillez en équipe et qu'un nouveau développeur se joint ultérieurement, il sera plus facile pour lui de se familiariser avec le code lui-même.

index.js : - Ce fichier va gérer la connexion au serveur.

```
//Import Libraries
var express = require('express'),
    session = require('express-session'),
    mongoose = require('mongoose'),
    request = require('request');

//Import custom modules
var userRoutes = require('./app/routes/userRoutes');
var config = require('./app/config/config');

//Connect to Mongo DB
mongoose.connect(config.getDBString());

//Create a new Express application and Configure it
var app = express();

//Configure Routes
app.use(config.API_PATH, userRoutes());

//Start the server
app.listen(config.PORT);
console.log('Server started at - '+ config.URL+ ":" +config.PORT);
```

config.js : - Ce fichier va gérer tous les paramètres liés à la configuration qui resteront les mêmes tout au long.

```
var config = {
VERSION: 1,
```

```
BUILD: 1,
URL: 'http://127.0.0.1',
API_PATH : '/api',
PORT : process.env.PORT || 8080,
DB : {
    //MongoDB configuration
    HOST : 'localhost',
    PORT : '27017',
    DATABASE : 'db'
},

/*
 * Get DB Connection String for connecting to MongoDB database
 */
getDBString : function(){
    return 'mongodb://'+ this.DB.HOST +':'+ this.DB.PORT +'/'+ this.DB.DATABASE;
},

/*
 * Get the http URL
 */
getHTTPUrl : function(){
    return 'http://' + this.URL + ":" + this.PORT;
}

module.exports = config;
```

user.js : - Fichier modèle où le schéma est défini

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

//Schema for User
var UserSchema = new Schema({
    name: {
        type: String,
    //    required: true
    },
    email: {
        type: String
    },
    password: {
        type: String,
        //required: true
    },
    dob: {
        type: Date,
        //required: true
    },
    gender: {
        type: String, // Male/Female
    //    required: true
    }
});

//Define the model for User
var User;
if(mongoose.models.User)
    User = mongoose.model('User');
else
```

```
    User = mongoose.model('User', UserSchema);

//Export the User Model
module.exports = User;
```

userController : - Ce fichier contient la fonction de connexion de l'utilisateur

```
var User = require('../models/user');
var crypto = require('crypto');

//Controller for User
var UserController = {

    //Create a User
    create: function(req, res){
        var repassword = req.body.repassword;
        var password = req.body.password;
        var userEmail = req.body.email;

        //Check if the email address already exists
        User.find({"email": userEmail}, function(err, usr){
            if(usr.length > 0){
                //Email Exists

                res.json('Email already exists');
                return;
            }
            else
            {
                //New Email

                //Check for same passwords
                if(password != repassword){
                    res.json('Passwords does not match');
                    return;
                }

                //Generate Password hash based on sha1
                var shasum = crypto.createHash('sha1');
                shasum.update(req.body.password);
                var passwordHash = shasum.digest('hex');

                //Create User
                var user = new User();
                user.name = req.body.name;
                user.email = req.body.email;
                user.password = passwordHash;
                user.dob = Date.parse(req.body.dob) || "";
                user.gender = req.body.gender;

                //Validate the User
                user.validate(function(err){
                    if(err){
                        res.json(err);
                        return;
                    }else{
                        //Finally save the User
                        user.save(function(err){
                            if(err)
                            {
```

```
                                res.json(err);
                                return;
                            }

                            //Remove Password before sending User details
                            user.password = undefined;
                            res.json(user);
                            return;
                        });
                    }
                });
            }
        });
    }

}

module.exports = UserController;
```

userRoutes.js : - Ceci la route pour userController

```
var express = require('express');
var UserController = require('../controllers/userController');

//Routes for User
var UserRoutes = function(app)
{
    var router = express.Router();

router.route('/users')
    .post(UserController.create);

return router;

}

module.exports = UserRoutes;
```

L'exemple ci-dessus peut sembler trop grand mais si un débutant chez node.js avec un petit mélange de connaissances expresses tente de passer à travers cela, il le trouvera facile et vraiment utile.

Lire Bon style de codage en ligne: https://riptutorial.com/fr/node-js/topic/6489/bon-style-de-codage

Chapitre 14: Cadres de modèles

Examples

Nunjucks

Moteur côté serveur avec héritage de bloc, mise en cache automatique, macros, contrôle asynchrone, etc. Fortement inspiré par jinja2, très similaire à Twig (php).

Docs - http://mozilla.github.io/nunjucks/
Installer - `npm i nunjucks`

Utilisation de base avec *Express* ci-dessous.

app.js

```
var express = require ('express');
var nunjucks  = require('nunjucks');

var app = express();
app.use(express.static('/public'));

// Apply nunjucks and add custom filter and function (for example).
var env = nunjucks.configure(['views/'], { // set folders with templates
    autoescape: true,
    express: app
});
env.addFilter('myFilter', function(obj, arg1, arg2) {
    console.log('myFilter', obj, arg1, arg2);
    // Do smth with obj
    return obj;
});
env.addGlobal('myFunc', function(obj, arg1) {
    console.log('myFunc', obj, arg1);
    // Do smth with obj
    return obj;
});

app.get('/', function(req, res){
    res.render('index.html', {title: 'Main page'});
});

app.get('/foo', function(req, res){
    res.locals.smthVar = 'This is Sparta!';
    res.render('foo.html', {title: 'Foo page'});
});

app.listen(3000, function() {
    console.log('Example app listening on port 3000...');
});
```

/views/index.html

```
<html>
<head>
    <title>Nunjucks example</title>
</head>
<body>
{% block content %}
{{title}}
{% endblock %}
</body>
</html>
```

/views/foo.html

```
{% extends "index.html" %}

{# This is comment #}
{% block content %}
    <h1>{{title}}</h1>
    {# apply custom function and next build-in and custom filters #}
    {{ myFunc(smthVar) | lower | myFilter(5, 'abc') }}
{% endblock %}
```

Lire Cadres de modèles en ligne: https://riptutorial.com/fr/node-js/topic/5885/cadres-de-modeles

Chapitre 15: Cas d'utilisation de Node.js

Examples

Serveur HTTP

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

console.log('Starting server...');
var config = {
    port: 80,
    contentType: 'application/json; charset=utf-8'
};
// JSON-API server on port 80

var server = http.createServer();
server.listen(config.port);
server.on('error', (err) => {
    if (err.code == 'EADDRINUSE') console.error('Port '+ config.port +' is already in use');
    else console.error(err.message);
});
server.on('request', (request, res) => {
    var remoteAddress = request.headers['x-forwarded-for'] ||
request.connection.remoteAddress; // Client address
    console.log(remoteAddress +' '+ request.method +' '+ request.url);

    var out = {};
    // Here you can change output according to `request.url`
    out.test = request.url;
    res.writeHead(200, {
        'Content-Type': config.contentType
    });
    res.end(JSON.stringify(out));
});
server.on('listening', () => {
    c.info('Server is available: http://localhost:'+ config.port);
});
```

Console avec invite de commande

```
const process = require('process');
const rl = require('readline').createInterface(process.stdin, process.stdout);

rl.pause();
console.log('Something long is happening here...');

var cliConfig = {
    promptPrefix: ' > '
}

/*
    Commands recognition
    BEGIN
*/
var commands = {
```

```
    eval: function(arg) { // Try typing in console: eval 2 * 10 ^ 3 + 2 ^ 4
        arg = arg.join(' ');
        try { console.log(eval(arg)); }
        catch (e) { console.log(e); }
    },
    exit: function(arg) {
        process.exit();
    }
};
rl.on('line', (str) => {
    rl.pause();
    var arg = str.trim().match(/([^"]+)|("(?:[^"\\]|\\.)+")/g); // Applying regular expression
for removing all spaces except for what between double quotes:
http://stackoverflow.com/a/14540319/2396907
    if (arg) {
        for (let n in arg) {
            arg[n] = arg[n].replace(/^\"|\"$/g, '');
        }
        var commandName = arg[0];
        var command = commands[commandName];
        if (command) {
            arg.shift();
            command(arg);
        }
        else console.log('Command "'+ commandName +'" doesn\'t exist');
    }
    rl.prompt();
});
/*
    END OF
    Commands recognition
*/

rl.setPrompt(cliConfig.promptPrefix);
rl.prompt();
```

Lire Cas d'utilisation de Node.js en ligne: https://riptutorial.com/fr/node-js/topic/7703/cas-d-utilisation-de-node-js

Chapitre 16: Chargement automatique des modifications

Examples

Autoreload sur les modifications du code source à l'aide de nodemon

Le paquet nodemon permet de recharger automatiquement votre programme lorsque vous modifiez un fichier du code source.

Installer nodemon globalement

`npm install -g nodemon` (ou `npm i -g nodemon`)

Installer nodemon localement

Au cas où vous ne voulez pas l'installer globalement

`npm install --save-dev nodemon` (ou `npm i -D nodemon`)

Utiliser nodemon

Exécutez votre programme avec `nodemon entry.js` (ou `nodemon entry`)

Cela remplace l'utilisation habituelle de `node entry.js` (ou `node entry`).

Vous pouvez également ajouter votre démarrage nodemon en tant que script npm, ce qui peut être utile si vous souhaitez fournir des paramètres et ne pas les taper à chaque fois.

Ajouter **package.json:**

```
"scripts": {
  "start": "nodemon entry.js -devmode -something 1"
}
```

De cette façon, vous pouvez simplement utiliser `npm start` depuis votre console.

Browsersync

Vue d'ensemble

Browsersync est un outil qui permet de regarder des fichiers en direct et de recharger un navigateur. Il est disponible en paquet NPM .

Installation

Pour installer Browsersync, vous devez d'abord installer Node.js et NPM. Pour plus d'informations, consultez la documentation SO relative à l' installation et à l'exécution de Node.js.

Une fois votre projet configuré, vous pouvez installer Browsersync avec la commande suivante:

```
$ npm install browser-sync -D
```

Cela va installer Browsersync dans le `node_modules` local `node_modules` et l'enregistrer dans les dépendances de votre développeur.

Si vous préférez l'installer globalement, utilisez l' `-g` à la place de l' `-D` .

Utilisateurs Windows

Si vous ne parvenez pas à installer Browsersync sous Windows, vous devrez peut-être installer Visual Studio pour pouvoir accéder aux outils de génération pour installer Browsersync. Vous devrez ensuite spécifier la version de Visual Studio que vous utilisez comme suit:

```
$ npm install browser-sync --msvs_version=2013 -D
```

Cette commande spécifie la version 2013 de Visual Studio.

Utilisation de base

Pour recharger automatiquement votre site chaque fois que vous modifiez un fichier JavaScript dans votre projet, utilisez la commande suivante:

```
$ browser-sync start --proxy "myproject.dev" --files "**/*.js"
```

Remplacez `myproject.dev` par l'adresse Web que vous utilisez pour accéder à votre projet. Browsersync affichera une autre adresse pouvant être utilisée pour accéder à votre site via le proxy.

Utilisation avancée

Outre l'interface de ligne de commande décrite ci-dessus, Browsersync peut également être utilisé avec Grunt.js et Gulp.js.

Grunt.js

L'utilisation de Grunt.js nécessite un plugin qui peut être installé comme ceci:

```
$ npm install grunt-browser-sync -D
```

Ensuite, vous allez ajouter cette ligne à votre `gruntfile.js` :

```
grunt.loadNpmTasks('grunt-browser-sync');
```

Gulp.js

Browsersync fonctionne comme un module CommonJS , donc pas besoin d'un plugin Gulp.js. Simplement besoin du module comme ceci:

```
var browserSync = require('browser-sync').create();
```

Vous pouvez maintenant utiliser l' API Browsersync pour le configurer selon vos besoins.

API

L'API Browsersync peut être trouvée ici: https://browsersync.io/docs/api

Lire Chargement automatique des modifications en ligne: https://riptutorial.com/fr/node-js/topic/1743/chargement-automatique-des-modifications

Chapitre 17: CLI

Syntaxe

- noeud [options] [options v8] [script.js | -e "script"] [arguments]

Examples

Options de ligne de commande

```
-v, --version
```

Ajouté dans: v0.1.3 Version du noeud d'impression.

```
-h, --help
```

Ajouté dans: v0.1.3 Options de ligne de commande du noeud d'impression. La sortie de cette option est moins détaillée que ce document.

```
-e, --eval "script"
```

Ajouté dans: v0.5.2 Évaluez l'argument suivant en tant que JavaScript. Les modules prédéfinis dans le REPL peuvent également être utilisés en script.

```
-p, --print "script"
```

Ajouté dans: v0.6.4 Identique à -e mais imprime le résultat.

```
-c, --check
```

Ajouté dans: v5.0.0 Syntaxe vérifier le script sans exécuter.

```
-i, --interactive
```

Ajouté dans: v0.7.7 Ouvre le REPL même si stdin ne semble pas être un terminal.

```
-r, --require module
```

Ajouté dans: v1.6.0 Précharger le module spécifié au démarrage.

Suit les règles de résolution du module require (). Le module peut être soit un chemin d'accès à un fichier, soit un nom de module de noeud.

```
--no-deprecation
```

Ajouté dans: v0.8.0 Silence des avertissements de dépréciation.

```
--trace-deprecation
```

Ajouté dans: v0.8.0 Impression des traces de pile pour les dépréciations.

```
--throw-deprecation
```

Ajouté dans: v0.11.14 Jeter les erreurs pour les dépréciations.

```
--no-warnings
```

Ajouté dans: v6.0.0 Désactiver tous les avertissements de processus (y compris les dépréciations).

```
--trace-warnings
```

Ajouté dans: v6.0.0 Imprimer des traces de pile pour les avertissements de processus (y compris les dépréciations).

```
--trace-sync-io
```

Ajouté dans: v2.1.0 Imprime une trace de pile chaque fois qu'une E / S synchrone est détectée après le premier tour de la boucle d'événement.

```
--zero-fill-buffers
```

Ajouté à: v6.0.0 Remplit automatiquement toutes les nouvelles instances de Buffer et SlowBuffer.

```
--preserve-symlinks
```

Ajouté dans: v6.3.0 Demande au chargeur de module de préserver les liens symboliques lors de la résolution et de la mise en cache des modules.

Par défaut, lorsque Node.js charge un module à partir d'un chemin d'accès symboliquement lié à un autre emplacement sur disque, Node.js déréférencera le lien et utilisera le "chemin réel" sur disque du module comme identifiant. et comme chemin racine pour localiser d'autres modules de dépendance. Dans la plupart des cas, ce comportement par défaut est acceptable. Cependant, lorsque vous utilisez des dépendances d'homologues liées symboliquement, comme illustré dans l'exemple ci-dessous, le comportement par défaut provoque la levée d'une exception si moduleA tente d'exiger moduleB comme dépendance homologue:

```
{appDir}
 ├── app
 │   ├── index.js
 │   └── node_modules
 │       ├── moduleA -> {appDir}/moduleA
 │       └── moduleB
```

```
│           ├── index.js
│           └── package.json
└── moduleA
    ├── index.js
    └── package.json
```

L'indicateur de ligne de commande --preserve-symlinks indique à Node.js d'utiliser le chemin de lien symbolique pour les modules, par opposition au chemin réel, ce qui permet de trouver des dépendances d'homologues liées symboliquement.

Notez toutefois que l'utilisation de --preserve-symlinks peut avoir d'autres effets secondaires. Plus précisément, les modules natifs liés symboliquement peuvent ne pas se charger si ceux-ci sont liés depuis plusieurs emplacements dans l'arborescence des dépendances (Node.js les verrait comme deux modules distincts et tenterait de charger le module plusieurs fois, provoquant la levée d'une exception)).

```
--track-heap-objects
```

Ajouté dans: v2.4.0 Suivi des allocations d'objets de tas pour les instantanés de tas.

```
--prof-process
```

Ajouté dans: v6.0.0 Processus Sortie du profileur Process v8 générée à l'aide de l'option v8 --prof.

```
--v8-options
```

Ajouté dans: v0.1.3 Options de ligne de commande Print v8.

Remarque: les options v8 permettent de séparer les mots par des tirets (-) ou des traits de soulignement (_).

Par exemple, --stack-trace-limit est équivalent à --stack_trace_limit.

```
--tls-cipher-list=list
```

Ajouté dans: v4.0.0 Spécifiez une autre liste de chiffrement TLS par défaut. (Nécessite Node.js pour être construit avec un support cryptographique. (Par défaut))

```
--enable-fips
```

Ajouté dans: v6.0.0 Activer la crypto compatible FIPS au démarrage. (Nécessite Node.js pour être construit avec ./configure --openssl-fips)

```
--force-fips
```

Ajouté à: v6.0.0 Force crypto compatible FIPS au démarrage. (Ne peut pas être désactivé à partir du code du script.) (Même exigence que --enable-fips)

```
--icu-data-dir=file
```

Ajouté dans: v0.11.15 Spécifiez le chemin de chargement des données ICU. (remplace NODE_ICU_DATA)

```
Environment Variables

NODE_DEBUG=module[,…]
```

Ajouté dans: v0.1.32 ',' - liste séparée des principaux modules devant imprimer les informations de débogage.

```
NODE_PATH=path[:…]
```

Ajouté dans: v0.1.32 ':' - liste séparée des répertoires précédés du chemin de recherche du module.

Remarque: sous Windows, il s'agit d'une liste séparée par un ';'

```
NODE_DISABLE_COLORS=1
```

Ajouté dans: v0.3.0 Lorsqu'il est défini sur 1, les couleurs ne seront pas utilisées dans le REPL.

```
NODE_ICU_DATA=file
```

Ajouté dans: v0.11.15 Chemin de données pour les données ICU (Intl object). Étendra les données liées lors de la compilation avec la prise en charge de small-icu.

```
NODE_REPL_HISTORY=file
```

Ajouté dans: v5.0.0 Chemin d'accès au fichier utilisé pour stocker l'historique persistant de la REPL. Le chemin par défaut est ~ / .node_repl_history, qui est remplacé par cette variable. Définir la valeur sur une chaîne vide ("" ou "") désactive l'historique persistant de REPL.

Lire CLI en ligne: https://riptutorial.com/fr/node-js/topic/6013/cli

Chapitre 18: Comment les modules sont chargés

Examples

Mode global

Si vous avez installé Node en utilisant le répertoire par défaut, alors que NPM installe les packages dans `/usr/local/lib/node_modules` . Si vous tapez ce qui suit dans le shell, NPM recherche, télécharge et installe la dernière version du package nommé sax dans le répertoire `/usr/local/lib/node_modules/express` :

```
$ npm install -g express
```

Assurez-vous de disposer de droits d'accès suffisants au dossier. Ces modules seront disponibles pour tous les processus de noeud qui seront exécutés sur cette machine.

En installation en mode local. Npm charge et installe les modules dans les dossiers de travail actuels en créant un nouveau dossier appelé `node_modules` par exemple si vous êtes dans `/home/user/apps/my_app` un nouveau dossier sera créé appelé `node_modules` `/home/user/apps/my_app/node_modules` s'il n'existe pas déjà

Chargement des modules

Lorsque l' on parle du module dans le code, premier noeud recherche le `node_module` dossier dans le dossier référencé dans l' instruction requise Si le nom du module n'est pas relatif et n'est pas un module de base, le nœud essaiera de le trouver à l' intérieur du `node_modules` dossier dans le courant annuaire. Par exemple, si vous procédez comme suit, Node essaiera de rechercher le fichier `./node_modules/myModule.js` :

```
var myModule = require('myModule.js');
```

Si Node ne parvient pas à trouver le fichier, il examinera le dossier parent nommé `../node_modules/myModule.js` . S'il échoue à nouveau, il essaiera le dossier parent et continuera à descendre jusqu'à ce qu'il atteigne la racine ou trouve le module requis.

Vous pouvez également omettre l'extension `.js` si vous le souhaitez, auquel cas le nœud ajoutera l'extension `.js` et recherchera le fichier.

Chargement d'un module de dossier

Vous pouvez utiliser le chemin d'accès d'un dossier pour charger un module comme celui-ci:

```
var myModule = require('./myModuleDir');
```

Si vous le faites, Node effectuera une recherche dans ce dossier. Node présume que ce dossier est un package et essaiera de rechercher une définition de package. Cette définition de package doit être un fichier nommé `package.json` . Si ce dossier ne contient pas de fichier de définition de package nommé `package.json` , le point d'entrée du package prendra la valeur par défaut `index.js` et Node recherchera, dans ce cas, un fichier sous le chemin d'accès `./myModuleDir/index.js` .

Le dernier recours si le module est introuvable dans l'un des dossiers est le dossier d'installation du module global.

Lire Comment les modules sont chargés en ligne: https://riptutorial.com/fr/node-js/topic/7738/comment-les-modules-sont-charges

Chapitre 19: Communication Arduino avec nodeJs

Introduction

Façon de montrer comment Node.Js peut communiquer avec Arduino Uno.

Examples

Node Js communication avec Arduino via serialport

Node js code

Un exemple pour démarrer cette rubrique est le serveur Node.js communiquant avec Arduino via serialport.

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

Exemple d'application.js:

```
const express = require('express');
const app = express();
var SerialPort = require("serialport");

var port = 3000;

var arduinoCOMPort = "COM3";

var arduinoSerialPort = new SerialPort(arduinoCOMPort, {
 baudrate: 9600
});

arduinoSerialPort.on('open',function() {
  console.log('Serial Port ' + arduinoCOMPort + ' is opened.');
});

app.get('/', function (req, res) {

    return res.send('Working');

})

app.get('/:action', function (req, res) {

   var action = req.params.action || req.param('action');

    if(action == 'led'){
        arduinoSerialPort.write("w");
        return res.send('Led light is on!');
```

```
    }
    if(action == 'off') {
        arduinoSerialPort.write("t");
        return res.send("Led light is off!");
    }

    return res.send('Action: ' + action);

});

app.listen(port, function () {
  console.log('Example app listening on port http://0.0.0.0:' + port + '!');
});
```

Exemple de serveur express:

```
node app.js
```

Code Arduino

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.

  Serial.begin(9600); // Begen listening on port 9600 for serial

  pinMode(LED_BUILTIN, OUTPUT);

  digitalWrite(LED_BUILTIN, LOW);
}

// the loop function runs over and over again forever
void loop() {

   if(Serial.available() > 0) // Read from serial port
    {
      char ReaderFromNode; // Store current character
      ReaderFromNode = (char) Serial.read();
      convertToState(ReaderFromNode); // Convert character to state
    }
  delay(1000);
}

void convertToState(char chr) {
  if(chr=='o'){
    digitalWrite(LED_BUILTIN, HIGH);
    delay(100);
  }
  if(chr=='f'){
    digitalWrite(LED_BUILTIN, LOW);
    delay(100);
  }
}
```

Démarrage

1. Connectez l'arduino à votre machine.
2. Démarrer le serveur

Contrôler la construction en led via le serveur js noeud express.

Allumer la led:

```
http://0.0.0.0:3000/led
```

Pour éteindre la led:

```
http://0.0.0.0:3000/off
```

Lire Communication Arduino avec nodeJs en ligne: https://riptutorial.com/fr/node-js/topic/10509/communication-arduino-avec-nodejs

Chapitre 20: Communication client-serveur

Examples

/ w Express, jQuery et Jade

```
//'client.jade'

//a button is placed down; similar in HTML
button(type='button', id='send_by_button') Modify data

    #modify Lorem ipsum Sender

    //loading jQuery; it can be done from an online source as well
    script(src='./js/jquery-2.2.0.min.js')

    //AJAX request using jQuery
    script
        $(function () {
            $('#send_by_button').click(function (e) {
                e.preventDefault();

                //test: the text within brackets should appear when clicking on said button
                //window.alert('You clicked on me.  - jQuery');

                //a variable and a JSON initialized in the code
                var predeclared = "Katamori";
                var data = {
                    Title: "Name_SenderTest",
                    Nick: predeclared,
                    FirstName: "Zoltan",
                    Surname: "Schmidt"
                };

                //an AJAX request with given parameters
                $.ajax({
                    type: 'POST',
                    data: JSON.stringify(data),
                    contentType: 'application/json',
                    url: 'http://localhost:7776/domaintest',

                    //on success, received data is used as 'data' function input
                    success: function (data) {
                        window.alert('Request sent; data received.');

                        var jsonstr = JSON.stringify(data);
                        var jsonobj = JSON.parse(jsonstr);

                        //if the 'nick' member of the JSON does not equal to the predeclared
string (as it was initialized), then the backend script was executed, meaning that
communication has been established
                        if(data.Nick != predeclared){
                            document.getElementById("modify").innerHTML = "JSON changed!\n" +
jsonstr;
                        };

                    }
```

```
                });
            });
            });

//'domaintest_route.js'

var express = require('express');
var router = express.Router();

//an Express router listening to GET requests - in this case, it's empty, meaning that nothing
is displayed when you reach 'localhost/domaintest'
router.get('/', function(req, res, next) {
});

//same for POST requests - notice, how the AJAX request above was defined as POST
router.post('/', function(req, res) {
    res.setHeader('Content-Type', 'application/json');

    //content generated here
    var some_json = {
        Title: "Test",
        Item: "Crate"
    };

    var result = JSON.stringify(some_json);

    //content got 'client.jade'
    var sent_data = req.body;
    sent_data.Nick = "ttony33";

    res.send(sent_data);

});


module.exports = router;
```

// basé sur un élément personnel utilisé: https://gist.github.com/Katamori/5c9850f02e4baf6e9896

Lire Communication client-serveur en ligne: https://riptutorial.com/fr/node-js/topic/6222/communication-client-serveur

Chapitre 21: Conception d'API reposante: meilleures pratiques

Examples

Gestion des erreurs: GET all resources

Comment gérez-vous les erreurs plutôt que de les connecter à la console?

Mauvaise manière:

```
Router.route('/')
  .get((req, res) => {
    Request.find((err, r) => {
      if(err){
        console.log(err)
      } else {
        res.json(r)
      }
    })
  })
  .post((req, res) => {
    const request = new Request({
      type: req.body.type,
      info: req.body.info
    });
    request.info.user = req.user._id;
    console.log("ABOUT TO SAVE REQUEST", request);
    request.save((err, r) => {
      if (err) {
        res.json({ message: 'there was an error saving your r' });
      } else {
        res.json(r);
      }
    });
  });
```

Meilleure façon:

```
Router.route('/')
 .get((req, res) => {
  Request.find((err, r) => {
   if(err){
    console.log(err)
  } else {
    return next(err)
  }
})
})
.post((req, res) => {
const request = new Request({
  type: req.body.type,
  info: req.body.info
```

```
});
request.info.user = req.user._id;
console.log("ABOUT TO SAVE REQUEST", request);
request.save((err, r) => {
  if (err) {
    return next(err)
  } else {
    res.json(r);
  }
});
});
```

Lire Conception d'API reposante: meilleures pratiques en ligne: https://riptutorial.com/fr/node-js/topic/6490/conception-d-api-reposante--meilleures-pratiques

Chapitre 22: Connectez-vous à Mongodb

Introduction

MongoDB est un programme de base de données multi-plateformes gratuit et open-source. Classée comme un programme de base de données NoSQL, MongoDB utilise des documents de type JSON avec des schémas.

Pour plus de détails, visitez https://www.mongodb.com/

Syntaxe

- MongoClient.connect ('mongodb: //127.0.0.1: 27017 / crud', fonction (err, db) {// faites quelque chose ici});

Examples

Exemple simple pour connecter mongoDB à partir de Node.JS

```
MongoClient.connect('mongodb://localhost:27017/myNewDB',function (err,db) {
    if(err)
        console.log("Unable to connect DB. Error: " + err)
    else
        console.log('Connected to DB');

    db.close();
});
```

myNewDB est le nom de la base de données, s'il n'existe pas dans la base de données, il sera automatiquement créé avec cet appel.

Un moyen simple de connecter mongoDB avec Node.JS de base

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

    //connection with mongoDB
    MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {
          //check the connection
 if(err){
           console.log("connection failed.");
}else{
                    console.log("successfully connected to mongoDB.");
    });
```

Lire Connectez-vous à Mongodb en ligne: https://riptutorial.com/fr/node-js/topic/6280/connectez-vous-a-mongodb

Chapitre 23: Création d'API avec Node.js

Examples

GET api en utilisant Express

`Node.js` apis peut être facilement construit dans un framework Web `Express` .

L'exemple suivant crée un simple api `GET` pour répertorier tous les utilisateurs.

Exemple

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

var users =[{
        id: 1,
        name: "John Doe",
        age : 23,
        email: "john@doe.com"
    }];

// GET /api/users
app.get('/api/users', function(req, res){
    return res.json(users);    //return response as JSON
});

app.listen('3000', function(){
    console.log('Server listening on port 3000');
});
```

POST api en utilisant Express

L'exemple suivant crée l'API `POST` utilisant `Express` . Cet exemple est similaire à l'exemple `GET` à l'exception de l'utilisation de `body-parser` qui analyse les données de publication et les ajoute à `req.body` .

Exemple

```
var express = require('express');
var app = express();
// for parsing the body in POST request
var bodyParser = require('body-parser');

var users =[{
    id: 1,
    name: "John Doe",
    age : 23,
    email: "john@doe.com"
}];

app.use(bodyParser.urlencoded({ extended: false }));
```

```
app.use(bodyParser.json());

// GET /api/users
app.get('/api/users', function(req, res){
    return res.json(users);
});


/* POST /api/users
    {
        "user": {
           "id": 3,
            "name": "Test User",
            "age" : 20,
            "email": "test@test.com"
        }
    }
*/
app.post('/api/users', function (req, res) {
    var user = req.body.user;
    users.push(user);

    return res.send('User has been added successfully');
});

app.listen('3000', function(){
    console.log('Server listening on port 3000');
});
```

Lire Création d'API avec Node.js en ligne: https://riptutorial.com/fr/node-js/topic/5991/creation-d-api-avec-node-js

Chapitre 24: Création d'une bibliothèque Node.js prenant en charge les promesses et les rappels d'erreur en premier

Introduction

Beaucoup de gens aiment travailler avec des promesses et / ou une syntaxe asynchrone / en attente, mais lors de l'écriture d'un module, il serait utile que certains programmeurs supportent également les méthodes classiques de style de rappel. Plutôt que de créer deux modules, ou deux ensembles de fonctions, ou de demander au programmeur d'annoncer votre module, votre module peut prendre en charge les deux méthodes de programmation en utilisant asCallback () de bluebird ou nodeify () de Q.

Examples

Exemple de module et programme correspondant utilisant Bluebird

math.js

```
'use strict';

const Promise = require('bluebird');

module.exports = {

  // example of a callback-only method
  callbackSum: function(a, b, callback) {
    if (typeof a !== 'number')
      return callback(new Error('"a" must be a number'));
    if (typeof b !== 'number')
      return callback(new Error('"b" must be a number'));

    return callback(null, a + b);
  },

  // example of a promise-only method
  promiseSum: function(a, b) {
    return new Promise(function(resolve, reject) {
      if (typeof a !== 'number')
        return reject(new Error('"a" must be a number'));
      if (typeof b !== 'number')
        return reject(new Error('"b" must be a number'));
      resolve(a + b);
    });
  },

  // a method that can be used as a promise or with callbacks
  sum: function(a, b, callback) {
    return new Promise(function(resolve, reject) {
```

```
      if (typeof a !== 'number')
        return reject(new Error('"a" must be a number'));
      if (typeof b !== 'number')
        return reject(new Error('"b" must be a number'));
      resolve(a + b);
    }).asCallback(callback);
  },

};
```

index.js

```
'use strict';

const math = require('./math');


// classic callbacks

math.callbackSum(1, 3, function(err, result) {
  if (err)
    console.log('Test 1: ' + err);
  else
    console.log('Test 1: the answer is ' + result);
});

math.callbackSum(1, 'd', function(err, result) {
  if (err)
    console.log('Test 2: ' + err);
  else
    console.log('Test 2: the answer is ' + result);
});


// promises

math.promiseSum(2, 5)
.then(function(result) {
  console.log('Test 3: the answer is ' + result);
})
.catch(function(err) {
  console.log('Test 3: ' + err);
});

math.promiseSum(1)
.then(function(result) {
  console.log('Test 4: the answer is ' + result);
})
.catch(function(err) {
  console.log('Test 4: ' + err);
});


// promise/callback method used like a promise

math.sum(8, 2)
.then(function(result) {
  console.log('Test 5: the answer is ' + result);
})
.catch(function(err) {
```

```
  console.log('Test 5: ' + err);
});


// promise/callback method used with callbacks

math.sum(7, 11, function(err, result) {
  if (err)
    console.log('Test 6: ' + err);
  else
    console.log('Test 6: the answer is ' + result);
});


// promise/callback method used like a promise with async/await syntax

(async () => {

  try {
    let x = await math.sum(6, 3);
    console.log('Test 7a: ' + x);

    let y = await math.sum(4, 's');
    console.log('Test 7b: ' + y);

  } catch(err) {
    console.log(err.message);
  }

})();
```

Lire Création d'une bibliothèque Node.js prenant en charge les promesses et les rappels d'erreur en premier en ligne: https://riptutorial.com/fr/node-js/topic/9874/creation-d-une-bibliotheque-node-js-prenant-en-charge-les-promesses-et-les-rappels-d-erreur-en-premier

Chapitre 25: Débogage à distance dans Node.JS

Examples

Configuration de l'exécution de NodeJS

Pour configurer le débogage distant du noeud, exécutez simplement le processus de noeud avec l'indicateur `--debug` . Vous pouvez ajouter un port sur lequel le débogueur doit s'exécuter avec `--debug=<port>` .

Lorsque votre processus nœud démarre, vous devriez voir le message

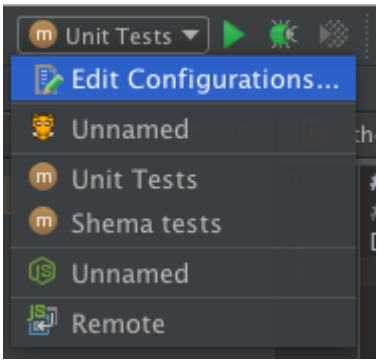
```
Debugger listening on port <port>
```

Ce qui vous dira que tout va bien.

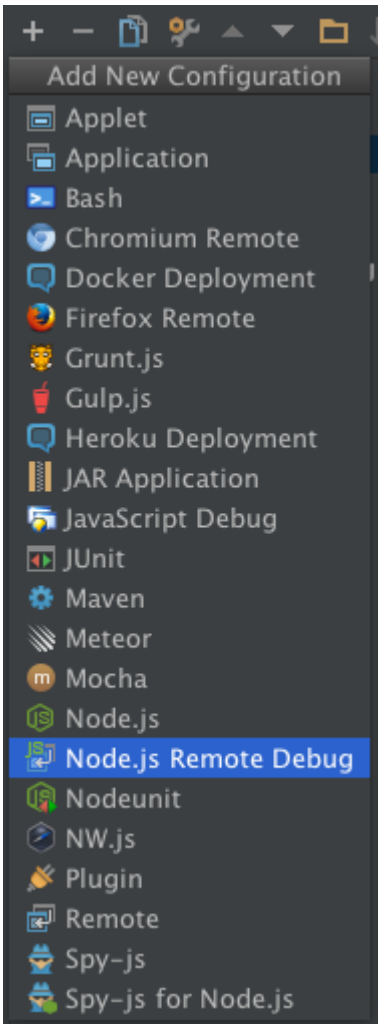
Ensuite, vous configurez la cible de débogage distant dans votre IDE spécifique.

Configuration IntelliJ / Webstorm

1. Assurez-vous que le plug-in NodeJS est activé
2. Sélectionnez vos configurations d'exécution (écran)



3. Sélectionnez **+** > **Débogage distant Node.js**



4. Assurez-vous d'entrer le port sélectionné ci-dessus ainsi que l'hôte correct

Une fois ceux-ci configurés, exécutez simplement la cible de débogage comme vous le feriez normalement et celle-ci s'arrêtera sur vos points d'arrêt.

Utilisez le proxy pour le débogage via le port sous Linux

Si vous démarrez votre application sous Linux, utilisez le proxy pour le débogage via le port, par exemple:

```
socat TCP-LISTEN:9958,fork TCP:127.0.0.1:5858 &
```

Utilisez ensuite le port 9958 pour le débogage à distance.

Lire Débogage à distance dans Node.JS en ligne: https://riptutorial.com/fr/node-js/topic/6335/debogage-a-distance-dans-node-js

Chapitre 26: Débogage de l'application Node.js

Examples

Core node.js debugger et inspecteur de noeud

Utiliser le débogueur de base

Node.js fournit un utilitaire de débogage non graphique. Pour démarrer la génération dans le débogueur, démarrez l'application avec cette commande:

```
node debug filename.js
```

Considérez l'application simple Node.js suivante contenue dans le `debugDemo.js`

```
'use strict';

function addTwoNumber(a, b){
// function returns the sum of the two numbers
debugger
  return a + b;
}

var result = addTwoNumber(5, 9);
console.log(result);
```

Le mot clé `debugger` arrête le débogueur à ce stade dans le code.

Référence de commande

1. Pas à pas

```
cont, c - Continue execution
next, n - Step next
step, s - Step in
out, o - Step out
```

2. Points d'arrêt

```
setBreakpoint(), sb() - Set breakpoint on current line
setBreakpoint(line), sb(line) - Set breakpoint on specific line
```

Pour déboguer le code ci-dessus, exécutez la commande suivante

```
node debug debugDemo.js
```

Une fois que les commandes ci-dessus s'exécutent, vous verrez la sortie suivante. Pour quitter l'interface du débogueur, tapez `process.exit()`

```
ankuranand:~/workspace/nodejs/nodejsDebugging $ node debug debugDemo.js
< Debugger listening on port 5858
debug> . ok
break in debugDemo.js:3
  1 // A Demo Code Showing the basic capabilities of the nodejs  debugging module
  2
> 3 'use strict';
  4
  5 function addTwoNumber(a, b){
debug> n
break in debugDemo.js:11
  9 }
 10
>11 let result = addTwoNumber(5, 9);
 12 console.log(result);
 13
debug> c
break in debugDemo.js:7
  5 function addTwoNumber(a, b){
  6 // function returns the sum of the two numbers
> 7 debugger
  8   return a + b;
  9 }
debug> c
< 14
debug> process.exit()
ankuranand:~/workspace/nodejs/nodejsDebugging $
```

Utilisez la commande `watch(expression)` pour ajouter la variable ou l'expression dont vous souhaitez surveiller la valeur et `restart` pour redémarrer l'application et le déboguer.

Utilisez `repl` pour saisir le code de manière interactive. Le mode repl a le même contexte que la ligne que vous déboguez. Cela vous permet d'examiner le contenu des variables et de tester des lignes de code. Appuyez sur `Ctrl+C` pour laisser le debug repl.

Utilisation de l'inspecteur de noeud intégré

v6.3.0

Vous pouvez exécuter l'inspecteur v8 intégré au nœud! Le plug-in d' inspecteur de nœud n'est plus nécessaire.

Passez simplement le drapeau de l'inspecteur et vous recevrez une URL pour l'inspecteur

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

Utilisation de l'inspecteur de noeud

Installez l'inspecteur de noeud:

```
npm install -g node-inspector
```

Exécutez votre application avec la commande node-debug:

```
node-debug filename.js
```

Après cela, appuyez sur Chrome:

```
http://localhost:8080/debug?port=5858
```

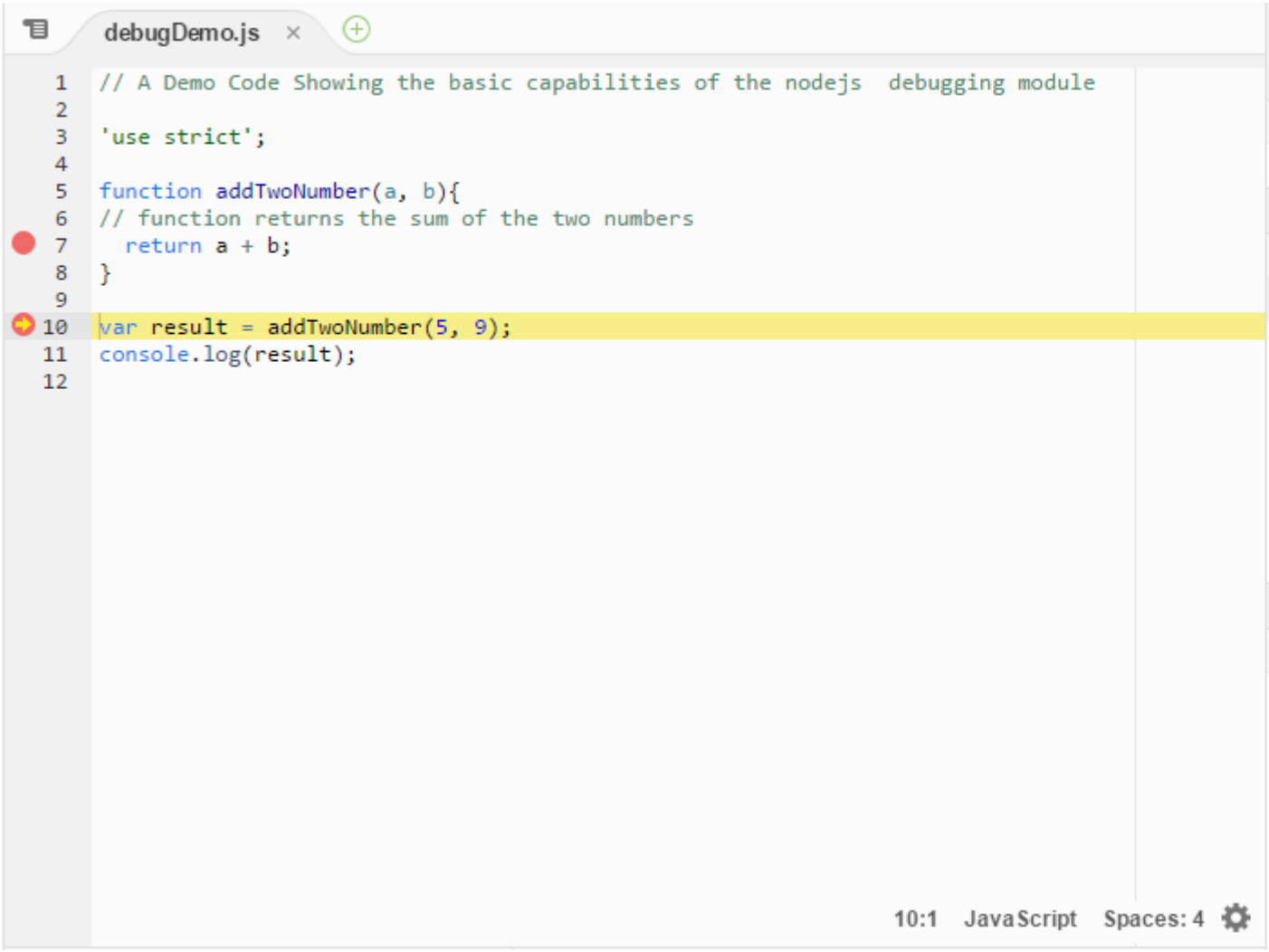
Parfois, le port 8080 peut ne pas être disponible sur votre ordinateur. Vous pouvez obtenir l'erreur suivante:

> Impossible de démarrer le serveur à 0.0.0.0:8080. Erreur: écoutez EACCES.

Dans ce cas, démarrez l'inspecteur de noeud sur un autre port à l'aide de la commande suivante.

```
$node-inspector --web-port=6500
```

Vous verrez quelque chose comme ceci:



Lire Débogage de l'application Node.js en ligne: https://riptutorial.com/fr/node-js/topic/5900/debogage-de-l-application-node-js

Chapitre 27: Défis de performance

Examples

Traitement des requêtes longues avec Node

Dans la mesure où Node est à thread unique, une solution de contournement s'impose s'il s'agit de calculs de longue durée.

Remarque: cet exemple est "prêt à fonctionner". Juste, n'oubliez pas d'obtenir jQuery et installez les modules requis.

Logique principale de cet exemple:

1. Le client envoie une demande au serveur.
2. Le serveur démarre la routine dans une instance de noeud distincte et envoie une réponse immédiate avec l'ID de tâche associé.
3. Le client envoie continuellement des chèques à un serveur pour les mises à jour de statut de l'ID de tâche donné.

Structure du projet:

```
project
|   package.json
|   index.html
|
├──js
|      main.js
|      jquery-1.12.0.min.js
|
└──srv
    |    app.js
    ├── models
    |      task.js
    └── tasks
           data-processor.js
```

app.js:

```
var express     = require('express');
var app         = express();
var http        = require('http').Server(app);
var mongoose    = require('mongoose');
var bodyParser  = require('body-parser');

var childProcess= require('child_process');

var Task        = require('./models/task');

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

```
app.use(express.static(__dirname + '/../'));

app.get('/', function(request, response){
    response.render('index.html');
});

//route for the request itself
app.post('/long-running-request', function(request, response){
    //create new task item for status tracking
    var t = new Task({ status: 'Starting ...' });

    t.save(function(err, task){
        //create new instance of node for running separate task in another thread
        taskProcessor = childProcess.fork('./srv/tasks/data-processor.js');

        //process the messages comming from the task processor
        taskProcessor.on('message', function(msg){
            task.status = msg.status;
            task.save();
        }.bind(this));

        //remove previously openned node instance when we finished
        taskProcessor.on('close', function(msg){
            this.kill();
        });

        //send some params to our separate task
        var params = {
            message: 'Hello from main thread'
        };

        taskProcessor.send(params);
        response.status(200).json(task);
    });
});

//route to check is the request is finished the calculations
app.post('/is-ready', function(request, response){
    Task
        .findById(request.body.id)
        .exec(function(err, task){
            response.status(200).json(task);
        });
});

mongoose.connect('mongodb://localhost/test');
http.listen('1234');
```

task.js:

```
var mongoose    = require('mongoose');

var taskSchema = mongoose.Schema({
    status: {
        type: String
    }
});

mongoose.model('Task', taskSchema);
```

```
module.exports  = mongoose.model('Task');
```

data-processor.js:

```
process.on('message', function(msg){
    init = function(){
        processData(msg.message);
    }.bind(this)();

    function processData(message){
        //send status update to the main app
        process.send({ status: 'We have started processing your data.' });

        //long calculations ..
        setTimeout(function(){
            process.send({ status: 'Done!' });

            //notify node, that we are done with this task
            process.disconnect();
        }, 5000);
    }
});

process.on('uncaughtException',function(err){
    console.log("Error happened: " + err.message + "\n" + err.stack + ".\n");
    console.log("Gracefully finish the routine.");
});
```

index.html:

```
<!DOCTYPE html>
<html>
    <head>
        <script src="./js/jquery-1.12.0.min.js"></script>
        <script src="./js/main.js"></script>
    </head>
    <body>
        <p>Example of processing long-running node requests.</p>
        <button id="go" type="button">Run</button>

        <br />

        <p>Log:</p>
        <textarea id="log" rows="20" cols="50"></textarea>
    </body>
</html>
```

main.js:

```
$(document).on('ready', function(){

    $('#go').on('click', function(e){
        //clear log
        $("#log").val('');

        $.post("/long-running-request", {some_params: 'params' })
            .done(function(task){
```

```
                $("#log").val( $("#log").val() + '\n' + task.status);

                //function for tracking the status of the task
                function updateStatus(){
                    $.post("/is-ready", {id: task._id })
                        .done(function(response){
                            $("#log").val( $("#log").val() + '\n' + response.status);

                            if(response.status != 'Done!'){
                                checkTaskTimeout = setTimeout(updateStatus, 500);
                            }
                        });
                }

                //start checking the task
                var checkTaskTimeout = setTimeout(updateStatus, 100);
            });
    });
});
```

package.json:

```
{
  "name": "nodeProcessor",
  "dependencies": {
    "body-parser": "^1.15.2",
    "express": "^4.14.0",
    "html": "0.0.10",
    "mongoose": "^4.5.5"
  }
}
```

Disclaimer: cet exemple est destiné à vous donner une idée de base. Pour l'utiliser dans un environnement de production, des améliorations sont nécessaires.

Lire Défis de performance en ligne: https://riptutorial.com/fr/node-js/topic/6325/defis-de-performance

Chapitre 28: Déploiement d'applications Node.js en production

Examples

Réglage NODE_ENV = "production"

Les déploiements de production varieront de plusieurs manières, mais une convention standard lors du déploiement en production consiste à définir une variable d'environnement appelée `NODE_ENV` et à définir sa valeur sur *«production»* .

Drapeaux d'exécution

Tout code exécuté dans votre application (y compris les modules externes) peut vérifier la valeur de `NODE_ENV` :

```
if(process.env.NODE_ENV === 'production') {
    // We are running in production mode
} else {
   // We are running in development mode
}
```

Les dépendances

Lorsque la variable d'environnement `NODE_ENV` est définie sur *'production',* toutes les `devDependencies` dans votre fichier *package.json* seront complètement ignorées lors de l'exécution de `npm install` . Vous pouvez également imposer ceci avec un drapeau `--production` :

```
npm install --production
```

Pour définir `NODE_ENV` vous pouvez utiliser l'une de ces méthodes

méthode 1: définissez NODE_ENV pour toutes les applications de noeud

Les fenêtres :

```
set NODE_ENV=production
```

Linux ou autre système basé sur Unix:

```
export NODE_ENV=production
```

Cela définit `NODE_ENV` pour la session bash en cours. Ainsi, toutes les applications démarrées après

cette instruction auront `NODE_ENV` défini sur `production` .

méthode 2: définissez NODE_ENV pour l'application en cours

```
NODE_ENV=production node app.js
```

Cela définira `NODE_ENV` pour l'application en cours. Cela aide lorsque nous voulons tester nos applications sur différents environnements.

méthode 3: créer `.env` fichier `.env` et l'utiliser

Cela utilise l'idée expliquée ici . Référer ce post pour une explication plus détaillée.

Fondamentalement, vous créez `.env` fichier `.env` et exécutez un script bash pour les définir dans l'environnement.

Pour éviter d'écrire un script bash, le package env-cmd peut être utilisé pour charger les variables d'environnement définies dans le fichier `.env` .

```
env-cmd .env node app.js
```

méthode 4: Utiliser `cross-env` package `cross-env`

Ce paquet permet de définir les variables d'environnement dans un sens pour chaque plate-forme.

Après l'avoir installé avec npm, vous pouvez simplement l'ajouter à votre script de déploiement dans `package.json` comme suit:

```
"build:deploy": "cross-env NODE_ENV=production webpack"
```

Gérer l'application avec le gestionnaire de processus

Il est recommandé d'exécuter les applications NodeJS contrôlées par les gestionnaires de processus. Le gestionnaire de processus aide à maintenir l'application active pour toujours, à redémarrer en cas d'échec, à recharger sans interruption et à simplifier l'administration. Les plus puissants d'entre eux (comme PM2) ont un équilibreur de charge intégré. PM2 vous permet également de gérer la journalisation, la surveillance et la mise en cluster des applications.

Gestionnaire de processus PM2

Installation de PM2:

```
npm install pm2 -g
```

Le processus peut être démarré en mode cluster impliquant un équilibreur de charge intégré pour répartir la charge entre les processus:

`pm2 start app.js -i 0 --name "api"` (*-i* doit spécifier le nombre de processus à générer. S'il est à 0, le numéro de processus sera basé sur le nombre de cœurs de processeur)

Tout en ayant plusieurs utilisateurs en production, il doit y avoir un seul point pour PM2. Par conséquent, la commande pm2 doit être préfixée par un emplacement (pour la configuration PM2), sinon elle générera un nouveau processus pm2 pour chaque utilisateur avec config dans son répertoire de base respectif. Et ce sera incohérent.

Utilisation: `PM2_HOME=/etc/.pm2 pm2 start app.js`

Déploiement à l'aide de PM2

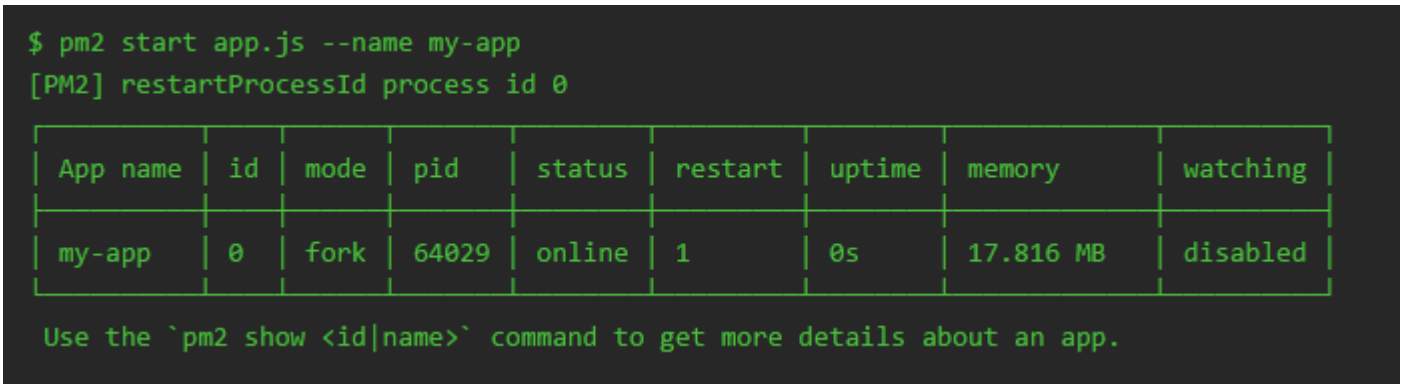
`PM2` est un gestionnaire de processus de production pour les applications `Node.js` , qui vous permet de garder les applications en vie pour toujours et de les recharger sans interruption. PM2 vous permet également de gérer la journalisation, la surveillance et la mise en cluster des applications.

Installez `pm2` globalement.

```
npm install -g pm2
```

Ensuite, exécutez l'application `node.js` utilisant `PM2`.

```
pm2 start server.js --name "my-app"
```



Les commandes suivantes sont utiles lorsque vous travaillez avec `PM2` .

Liste tous les processus en cours d'exécution:

```
pm2 list
```

Arrêtez une application:

```
pm2 stop my-app
```

Redémarrez une application:

```
pm2 restart my-app
```

Pour afficher des informations détaillées sur une application:

```
pm2 show my-app
```

Pour supprimer une application du registre de PM2:

```
pm2 delete my-app
```

Déploiement à l'aide du gestionnaire de processus

Le gestionnaire de processus est généralement utilisé en production pour déployer une application nodejs. Les principales fonctions d'un gestionnaire de processus sont le redémarrage du serveur en cas de panne, la vérification de la consommation des ressources, l'amélioration des performances d'exécution, la surveillance, etc.

Certains des gestionnaires de processus populaires créés par la communauté de nœuds sont pour toujours, pm2, etc.

Forvever

`forever` est un outil d'interface de ligne de commande permettant de garantir l'exécution continue d'un script donné. L'interface simple de `forever` le rend idéal pour exécuter de plus petits déploiements d'applications et de scripts `Node.js`

surveille `forever` votre processus et le redémarre s'il se bloque.

Installer `forever` globalement.

```
$ npm install -g forever
```

Exécuter l'application:

```
$ forever start server.js
```

Cela démarre le serveur et donne un identifiant pour le processus (à partir de 0).

Redémarrer l'application:

```
$ forever restart 0
```

Ici `0` est l'id du serveur.

Arrêter l'application:

```
$ forever stop 0
```

Semblable à redémarrer, `0` correspond à l'identifiant du serveur. Vous pouvez également donner un identifiant de processus ou un nom de script à la place de l'identifiant fourni par l'indispensable.

Pour plus de commandes: https://www.npmjs.com/package/forever

Utiliser différentes propriétés / configurations pour différents environnements tels que dev, qa, staging etc.

Les applications à grande échelle nécessitent souvent des propriétés différentes lorsqu'elles sont exécutées dans des environnements différents. Nous pouvons y parvenir en transmettant des arguments à l'application NodeJs et en utilisant le même argument dans le processus de noeud pour charger un fichier de propriétés d'environnement spécifique.

Supposons que nous ayons deux fichiers de propriétés pour un environnement différent.

- dev.json

```
{
    "PORT": 3000,
    "DB": {
        "host": "localhost",
        "user": "bob",
        "password": "12345"
    }
}
```

- qa.json

```
{
    "PORT": 3001,
    "DB": {
        "host": "where_db_is_hosted",
        "user": "bob",
        "password": "54321"
    }
}
```

Le code suivant dans l'application exportera le fichier de propriétés respectif que nous souhaitons utiliser.

```
process.argv.forEach(function (val) {
    var arg = val.split("=");
    if (arg.length > 0) {
        if (arg[0] === 'env') {
            var env = require('./' + arg[1] + '.json');
            exports.prop = env;
        }
    }
});
```

Nous donnons des arguments à l'application comme suit

```
node app.js env=dev
```

si nous utilisons gestionnaire de processus comme *pour toujours* aussi simple que

```
forever start app.js env=dev
```

Profiter des clusters

Une seule instance de Node.js s'exécute dans un seul thread. Pour tirer parti des systèmes multi-core, l'utilisateur voudra parfois lancer un cluster de processus Node.js pour gérer la charge.

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

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

        if (cluster.isMaster) {
          // In real life, you'd probably use more than just 2 workers,
          // and perhaps not put the master and worker in the same file.
          //
          // You can also of course get a bit fancier about logging, and
          // implement whatever custom logic you need to prevent DoS
          // attacks and other bad behavior.
          //
          // See the options in the cluster documentation.
          //
          // The important thing is that the master does very little,
          // increasing our resilience to unexpected errors.
            console.log('your server is working on ' + numCPUs + ' cores');

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

          cluster.on('disconnect', function(worker) {
            console.error('disconnect!');
            //clearTimeout(timeout);
            cluster.fork();
          });


        } else {
            require('./app.js');

        }
```

Lire Déploiement d'applications Node.js en production en ligne: https://riptutorial.com/fr/node-js/topic/2975/deploiement-d-applications-node-js-en-production

Chapitre 29: Déploiement de l'application Node.js sans temps d'arrêt.

Examples

Déploiement à l'aide de PM2 sans temps d'arrêt.

`ecosystem.json`

```
{
    "name": "app-name",
    "script": "server",
    "exec_mode": "cluster",
    "instances": 0,
    "wait_ready": true
    "listen_timeout": 10000,
    "kill_timeout": 5000,
}
```

```
wait_ready
```

Au lieu de recharger en attente d'un événement d'écoute, attendez le processus.send ('ready');

```
listen_timeout
```

Temps en ms avant de forcer un rechargement si l'application n'écoute pas.

```
kill_timeout
```

Temps en ms avant d'envoyer un SIGKLL final.

`server.js`

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

const app = express();
const server = http.Server(app);
const port = 80;

server.listen(port, function() {
    process.send('ready');
});

process.on('SIGINT', function() {
    server.close(function() {
        process.exit(0);
    });
});
```

Vous devrez peut-être attendre que votre application ait établi des connexions avec vos bases de données / caches / travailleurs / autres. PM2 doit attendre avant de considérer votre application comme étant en ligne. Pour ce faire, vous devez fournir `wait_ready: true` dans un fichier de processus. Cela fera écouter PM2 pour cet événement. Dans votre application, vous devrez ajouter `process.send('ready');` lorsque vous souhaitez que votre demande soit considérée comme prête.

Lorsqu'un processus est arrêté / redémarré par PM2, certains signaux système sont envoyés à votre processus dans un ordre donné.

Tout d'abord, un signal `SIGINT` est envoyé à vos processus, signalant que vous pouvez savoir que votre processus va être arrêté. Si votre application ne sort pas d'elle-même avant 1.6s (personnalisable), elle recevra un signal `SIGKILL` pour forcer la sortie du processus. Donc, si votre application doit nettoyer des états ou des tâches, vous pouvez intercepter le signal `SIGINT` pour préparer votre application à quitter.

Lire Déploiement de l'application Node.js sans temps d'arrêt. en ligne: https://riptutorial.com/fr/node-js/topic/9752/deploiement-de-l-application-node-js-sans-temps-d-arret-

Chapitre 30: Désinstallation de Node.js

Examples

Désinstallez complètement Node.js sur Mac OSX

Dans Terminal sur votre système d'exploitation Mac, entrez les 2 commandes suivantes:

```
lsbom -f -l -s -pf /var/db/receipts/org.nodejs.pkg.bom | while read f; do  sudo rm
/usr/local/${f}; done

sudo rm -rf /usr/local/lib/node /usr/local/lib/node_modules /var/db/receipts/org.nodejs.*
```

Désinstallez Node.js sous Windows

Pour désinstaller Node.js sous Windows, utilisez Ajout / Suppression de programmes comme celui-ci:

1. Ouvrez `Add or Remove Programs` dans le menu Démarrer.
2. Recherchez `Node.js`

Windows 10:

3. Cliquez sur Node.js.
4. Cliquez sur Désinstaller.
5. Cliquez sur le nouveau bouton Désinstaller.

Windows 7-8.1:

3. Cliquez sur le bouton Désinstaller sous Node.js.

Lire Désinstallation de Node.js en ligne: https://riptutorial.com/fr/node-js/topic/2821/desinstallation-de-node-js

Chapitre 31: ECMAScript 2015 (ES6) avec Node.js

Examples

déclarations const / let

Contrairement à `var` , `const` / `let` est lié à la portée lexicale plutôt qu'à la portée de la fonction.

```
{
  var x = 1 // will escape the scope
  let y = 2 // bound to lexical scope
  const z = 3 // bound to lexical scope, constant
}

console.log(x) // 1
console.log(y) // ReferenceError: y is not defined
console.log(z) // ReferenceError: z is not defined
```

Exécuter dans RunKit

Fonctions de flèche

Les fonctions fléchées se lient automatiquement à la portée lexicale «this» du code environnant.

```
performSomething(result => {
  this.someVariable = result
})
```

contre

```
performSomething(function(result) {
  this.someVariable = result
}.bind(this))
```

Exemple de fonction de flèche

Considérons cet exemple, qui produit les carrés des nombres 3, 5 et 7:

```
let nums = [3, 5, 7]
let squares = nums.map(function (n) {
  return n * n
})
console.log(squares)
```

Exécuter dans RunKit

La fonction transmise à `.map` peut également être écrite en tant que fonction flèche en supprimant

le mot-clé `function` et en ajoutant la flèche `=>` :

```
let nums = [3, 5, 7]
let squares = nums.map((n) => {
  return n * n
})
console.log(squares)
```

Exécuter dans RunKit

Cependant, cela peut être écrit encore plus concis. Si le corps de la fonction se compose d'une seule instruction et que cette instruction calcule la valeur de retour, les accolades de l'enveloppe du corps de la fonction peuvent être supprimées, ainsi que le mot clé `return` .

```
let nums = [3, 5, 7]
let squares = nums.map(n => n * n)
console.log(squares)
```

Exécuter dans RunKit

déstructurer

```
    let [x,y, ...nums] = [0, 1, 2, 3, 4, 5, 6];
console.log(x, y, nums);

let {a, b, ...props} = {a:1, b:2, c:3, d:{e:4}}
console.log(a, b, props);

let dog = {name: 'fido', age: 3};
let {name:n, age} = dog;
console.log(n, age);
```

couler

```
/* @flow */

function product(a: number, b: number){
  return a * b;
}

const b = 3;
let c = [1,2,3,,{}];
let d = 3;

import request from 'request';

request('http://dev.markitondemand.com/MODApis/Api/v2/Quote/json?symbol=AAPL', (err, res,
payload)=>{
  payload = JSON.parse(payload);
  let {LastPrice} = payload;
  console.log(LastPrice);
});
```

Classe ES6

```
class Mammel {
  constructor(legs){
    this.legs = legs;
  }
  eat(){
    console.log('eating...');
  }
  static count(){
    console.log('static count...');
  }
}

class Dog extends Mammel{
  constructor(name, legs){
    super(legs);
    this.name = name;
  }
  sleep(){
    super.eat();
    console.log('sleeping');
  }
}

let d = new Dog('fido', 4);
d.sleep();
d.eat();
console.log('d', d);
```

Lire ECMAScript 2015 (ES6) avec Node.js en ligne: https://riptutorial.com/fr/node-js/topic/6732/ecmascript-2015--es6--avec-node-js

Chapitre 32: Émetteurs d'événements

Remarques

Lorsqu'un événement "se déclenche" (ce qui signifie la même chose que "publier un événement" ou "émettre un événement"), chaque écouteur sera appelé de manière synchrone (source), avec toutes les données d'accompagnement transmises à `emit()` , non Peu importe le nombre d'arguments que vous transmettez:

```
myDog.on('bark', (howLoud, howLong, howIntense) => {
  // handle the event
})
myDog.emit('bark', 'loudly', '5 seconds long', 'fiercely')
```

Les auditeurs seront appelés dans l'ordre dans lequel ils ont été enregistrés:

```
myDog.on('urinate', () => console.log('My first thought was "Oh-no"'))
myDog.on('urinate', () => console.log('My second thought was "Not my lawn :)"'))
myDog.emit('urinate')
// The console.logs will happen in the right order because they were registered in that order.
```

Mais si vous avez besoin d'un écouteur pour lancer en premier, avant que tous les autres écouteurs déjà ajoutés, vous pouvez utiliser `prependListener()` comme ceci:

```
myDog.prependListener('urinate', () => console.log('This happens before my first and second
thoughts, even though it was registered after them'))
```

Si vous avez besoin d'écouter un événement, mais vous voulez seulement entendre parler une fois, vous pouvez utiliser `once` au lieu de `on` ou `prependOnceListener` au lieu de `prependListener` . Après le déclenchement de l'événement et l'appel de l'écouteur, le programme d'écoute sera automatiquement supprimé et ne sera plus appelé lors du prochain déclenchement de l'événement.

Enfin, si vous souhaitez supprimer tous les auditeurs et recommencer, n'hésitez pas à le faire:

```
myDog.removeAllListeners()
```

Examples

Analyses HTTP via un émetteur d'événements

Dans le code du serveur HTTP (par exemple `server.js`):

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

```
// Set up an HTTP server
const http = require('http')
const httpServer = http.createServer((request, response) => {
  // Handler the request...
  // Then emit an event about what happened
  serverEvents.emit('request', request.method, request.url)
});

// Expose the event emitter
module.exports = serverEvents
```

En code superviseur (par exemple, `supervisor.js`):

```
const server = require('./server.js')
// Since the server exported an event emitter, we can listen to it for changes:
server.on('request', (method, url) => {
  console.log(`Got a request: ${method} ${url}`)
})
```

Chaque fois que le serveur reçoit une demande, il émet un événement appelé `request` que le superviseur écoute, puis le superviseur peut réagir à l'événement.

Les bases

Les émetteurs d'événements sont intégrés au nœud et sont destinés à un sous-groupe, un modèle dans lequel un *éditeur* émet des événements, auxquels les *abonnés* peuvent écouter et réagir. Dans le jargon Node, les éditeurs sont appelés *émetteurs d'événement* et émettent des événements, tandis que les abonnés sont appelés *auditeurs* et réagissent aux événements.

```
// Require events to start using them
const EventEmitter = require('events').EventEmitter;
// Dogs have events to publish, or emit
class Dog extends EventEmitter {};
class Food {};

let myDog = new Dog();

// When myDog is chewing, run the following function
myDog.on('chew', (item) => {
  if (item instanceof Food) {
    console.log('Good dog');
  } else {
    console.log(`Time to buy another ${item}`);
  }
});

myDog.emit('chew', 'shoe'); // Will result in console.log('Time to buy another shoe')
const bacon = new Food();
myDog.emit('chew', bacon); // Will result in console.log('Good dog')
```

Dans l'exemple ci-dessus, le chien est l'éditeur / EventEmitter, tandis que la fonction qui vérifie l'élément est l'abonné / auditeur. Vous pouvez aussi faire plus d'auditeurs:

```
myDog.on('bark', () => {
```

```
  console.log('WHO\'S AT THE DOOR?');
  // Panic
});
```

Il peut également y avoir plusieurs écouteurs pour un même événement et même supprimer des écouteurs:

```
myDog.on('chew', takeADeepBreathe);
myDog.on('chew', calmDown);
// Undo the previous line with the next one:
myDog.removeListener('chew', calmDown);
```

Si vous souhaitez écouter un événement une seule fois, vous pouvez utiliser:

```
myDog.once('chew', pet);
```

Qui supprimera automatiquement l'auditeur sans conditions de course.

Obtenez les noms des événements auxquels vous êtes abonné

La fonction **EventEmitter.eventNames ()** renverra un tableau contenant les noms des événements auxquels vous **êtes** actuellement abonné.

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}

var emitter = new MyEmitter();

emitter
.on("message", function(){ //listen for message event
    console.log("a message was emitted!");
})
.on("message", function(){ //listen for message event
    console.log("this is not the right message");
})
.on("data", function(){ //listen for data event
    console.log("a data just occured!!");
});

console.log(emitter.eventNames()); //=> ["message","data"]
emitter.removeAllListeners("data");//=> removeAllListeners to data event
console.log(emitter.eventNames()); //=> ["message"]
```

Exécuter dans RunKit

Obtenir le nombre d'auditeurs inscrits pour écouter un événement spécifique

La fonction Emitter.listenerCount (eventName) renverra le nombre d'écouteurs qui écoutent actuellement l'événement fourni en argument.

```
const EventEmitter = require("events");
class MyEmitter extends EventEmitter{}
```

```
var emitter = new MyEmitter();

emitter
.on("data", ()=>{ // add listener for data event
    console.log("data event emitter");
});

console.log(emitter.listenerCount("data"))    // => 1
console.log(emitter.listenerCount("message")) // => 0

emitter.on("message", function mListener(){ //add listener for message event
    console.log("message event emitted");
});
console.log(emitter.listenerCount("data"))    // => 1
console.log(emitter.listenerCount("message")) // => 1

emitter.once("data", (stuff)=>{ //add another listener for data event
    console.log(`Tell me my ${stuff}`);
})

console.log(emitter.listenerCount("data"))   // => 2
console.log(emitter.listenerCount("message"))// => 1
```

Lire Émetteurs d'événements en ligne: https://riptutorial.com/fr/node-js/topic/1623/emetteurs-d-evenements

Chapitre 33: Environnement

Examples

Accès aux variables d'environnement

La propriété `process.env` renvoie un objet contenant l'environnement utilisateur.

Il retourne un objet comme celui-ci:

```
{
  TERM: 'xterm-256color',
  SHELL: '/usr/local/bin/bash',
  USER: 'maciej',
  PATH: '~/.bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  PWD: '/Users/maciej',
  EDITOR: 'vim',
  SHLVL: '1',
  HOME: '/Users/maciej',
  LOGNAME: 'maciej',
  _: '/usr/local/bin/node'
}
```

```
process.env.HOME // '/Users/maciej'
```

Si vous définissez la variable d'environnement `FOO` sur `foobar` , elle sera accessible avec:

```
process.env.FOO // 'foobar'
```

Arguments de ligne de commande process.argv

process.argv est un tableau contenant les arguments de la ligne de commande. Le premier élément sera `node` , le second élément sera le nom du fichier JavaScript. Les éléments suivants seront des arguments de ligne de commande supplémentaires.

Exemple de code:

Somme de tous les arguments de la ligne de commande

`index.js`

```
var sum = 0;
for (i = 2; i < process.argv.length; i++) {
    sum += Number(process.argv[i]);
}

console.log(sum);
```

Utilisation Exaple:

```
node index.js 2 5 6 7
```

La sortie sera `20`

Une brève explication du code:

Ici, pour boucle `for (i = 2; i < process.argv.length; i++)` boucle commence par 2 car les deux premiers éléments du tableau process.argv sont **toujours** `['path/to/node.exe', 'path/to/js/file', ...]`

Conversion en nombre `Number(process.argv[i])` car les éléments du tableau process.argv sont **toujours des** chaînes

Utiliser différentes propriétés / configurations pour différents environnements tels que dev, qa, staging etc.

Les applications à grande échelle nécessitent souvent des propriétés différentes lorsqu'elles sont exécutées dans des environnements différents. Nous pouvons y parvenir en transmettant des arguments à l'application NodeJs et en utilisant le même argument dans le processus de noeud pour charger un fichier de propriétés d'environnement spécifique.

Supposons que nous ayons deux fichiers de propriétés pour un environnement différent.

- dev.json

```
{
    PORT : 3000,
    DB : {
        host : "localhost",
        user : "bob",
        password : "12345"
    }
}
```

- qa.json

```
{
        PORT : 3001,
        DB : {
            host : "where_db_is_hosted",
            user : "bob",
            password : "54321"
        }
}
```

Le code suivant dans l'application exportera le fichier de propriétés respectif que nous souhaitons utiliser.

Supposons que le code soit dans environment.js

```
process.argv.forEach(function (val, index, array) {
    var arg = val.split("=");
    if (arg.length > 0) {
        if (arg[0] === 'env') {
            var env = require('./' + arg[1] + '.json');
            module.exports = env;
        }
    }
});
```

Nous donnons des arguments à l'application comme suit

```
node app.js env=dev
```

si nous utilisons gestionnaire de processus comme *pour toujours* aussi simple que

```
forever start app.js env=dev
```

Comment utiliser le fichier de configuration

```
 var env= require("environment.js");
```

Chargement des propriétés de l'environnement à partir d'un "fichier de propriétés"

- Lecteur de propriétés d'installation:

```
npm install properties-reader --save
```

- Créez un **répertoire env** pour stocker vos fichiers de propriétés:

```
mkdir env
```

- Créer des **environnements.js** :

```
process.argv.forEach(function (val, index, array) {
    var arg = val.split("=");
    if (arg.length > 0) {
        if (arg[0] === 'env') {
            var env = require('./env/' + arg[1] + '.properties');
            module.exports = env;
        }
    }
});
```

- Exemple de fichier de propriétés **development.properties** :

```
# Dev properties
```

```
[main]
# Application port to run the node server
app.port=8080

[database]
# Database connection to mysql
mysql.host=localhost
mysql.port=2500
...
```

- Exemple d'utilisation des propriétés chargées:

```
var enviorment = require('./environments');
var PropertiesReader = require('properties-reader');
var properties = new PropertiesReader(enviorment);

var someVal = properties.get('main.app.port');
```

- Démarrer le serveur express

```
npm start env=development
```

ou

```
npm start env=production
```

Lire Environnement en ligne: https://riptutorial.com/fr/node-js/topic/2340/environnement

Chapitre 34: Envoi d'un flux de fichiers au client

Examples

Utiliser fs et pipe Pour diffuser des fichiers statiques à partir du serveur

Un bon service VOD (Video On Demand) devrait commencer avec les bases. Disons que vous avez un répertoire sur votre serveur qui n'est pas accessible au public, mais par le biais d'une sorte de portail ou de paywall que vous souhaitez autoriser les utilisateurs à accéder à vos médias.

```
var movie = path.resolve('./public/' + req.params.filename);

        fs.stat(movie, function (err, stats) {

            var range = req.headers.range;

            if (!range) {

                return res.sendStatus(416);

            }

            //Chunk logic here
            var positions = range.replace(/bytes=/, "").split("-");
            var start = parseInt(positions[0], 10);
            var total = stats.size;
            var end = positions[1] ? parseInt(positions[1], 10) : total - 1;
            var chunksize = (end - start) + 1;

            res.writeHead(206, {

                'Transfer-Encoding': 'chunked',

                "Content-Range": "bytes " + start + "-" + end + "/" + total,

                "Accept-Ranges": "bytes",

                "Content-Length": chunksize,

                "Content-Type": mime.lookup(req.params.filename)

            });

            var stream = fs.createReadStream(movie, { start: start, end: end, autoClose: true
})

                .on('end', function () {

                    console.log('Stream Done');

                })
```

```
                .on("error", function (err) {

                    res.end(err);

                })

                .pipe(res, { end: true });

        });
```

L'extrait ci-dessus est une description de base de la manière dont vous souhaitez diffuser votre vidéo sur un client. La logique de segment dépend de divers facteurs, notamment du trafic réseau et de la latence. Il est important d'équilibrer la taille du mandrin par la quantité.

Enfin, l'appel .pipe permet à node.js de garder une connexion ouverte avec le serveur et d'envoyer des blocs supplémentaires si nécessaire.

Streaming Utilisation de ffmpeg fluide

Vous pouvez également utiliser flent-ffmpeg pour convertir des fichiers .mp4 en fichiers .flv ou d'autres types:

res.contentType ('flv');

```
    var pathToMovie = './public/' + req.params.filename;

    var proc = ffmpeg(pathToMovie)

        .preset('flashvideo')

        .on('end', function () {

            console.log('Stream Done');

        })

        .on('error', function (err) {

            console.log('an error happened: ' + err.message);

            res.send(err.message);

        })

        .pipe(res, { end: true });
```

Lire Envoi d'un flux de fichiers au client en ligne: https://riptutorial.com/fr/node-js/topic/6994/envoi-d-un-flux-de-fichiers-au-client

Chapitre 35: Envoyer une notification Web

Examples

Envoyer une notification Web à l'aide de GCM (Google Cloud Messaging System)

Un tel exemple consiste à connaître une large diffusion parmi les applications Web progressives (**PWA**) et, dans cet exemple, nous allons envoyer une notification simple du type Backend en utilisant **NodeJS** et **ES6.**

1. Installer le module Node-GCM: `npm install node-gcm`
2. Installez Socket.io: `npm install socket.io`
3. Créez une application GCM activée à l'aide de Google Console.
4. Grabe votre identifiant d'application GCM (nous en aurons besoin plus tard)
5. Grabe votre code secret d'application GCM.
6. Ouvrez votre éditeur de code préféré et ajoutez le code suivant:

```
'use strict';

const express = require('express');
const app = express();
const gcm = require('node-gcm');
app.io = require('socket.io')();

// [*] Configuring our GCM Channel.
const sender = new gcm.Sender('Project Secret');
const regTokens = [];
let message = new gcm.Message({
    data: {
        key1: 'msg1'
    }
});

// [*] Configuring our static files.
app.use(express.static('public/'));

// [*] Configuring Routes.
app.get('/', (req, res) => {
    res.sendFile(__dirname + '/public/index.html');
});

// [*] Configuring our Socket Connection.
app.io.on('connection', socket => {
    console.log('we have a new connection ...');
    socket.on('new_user', (reg_id) => {
        // [*] Adding our user notification registration token to our list typically
hided in a secret place.
```

```
        if (regTokens.indexOf(reg_id) === -1) {
            regTokens.push(reg_id);

            // [*] Sending our push messages
            sender.send(message, {
                registrationTokens: regTokens
            }, (err, response) => {
                if (err) console.error('err', err);
                else console.log(response);
            });
        }
    })
});

module.exports = app
```

PS: J'utilise ici un hack spécial pour que Socket.io fonctionne avec Express car il ne fonctionne tout simplement pas en dehors de la boîte.

Maintenant, créez un fichier **.json** et nommez-le: **Manifest.json** , ouvrez-le et passez les éléments suivants:

```
{
    "name": "Application Name",
    "gcm_sender_id": "GCM Project ID"
}
```

Fermez-le et enregistrez-le dans votre répertoire **racine d'** application.

PS: le fichier Manifest.json doit être dans le répertoire racine ou ne fonctionnera pas.

Dans le code ci-dessus, je fais ce qui suit:

1. J'ai créé et envoyé une page index.html normale qui utilisera également socket.io.
2. J'écoute un événement de **connexion** déclenché depuis le **front-end** aka ma **page index.html** (il sera déclenché une fois qu'un nouveau client sera connecté à notre lien prédéfini)
3. J'envoie un jeton spécial connu comme **jeton d'enregistrement à** partir de mon index.html via l'événement new.user de **socket.io** , ce jeton sera notre mot de passe unique pour chaque utilisateur et chaque code sera généré par un navigateur compatible pour l' **API de notification Web** (en savoir plus ici.
4. J'utilise simplement le module **node-gcm** pour envoyer ma notification qui sera traitée et montrée plus tard en utilisant **Service Workers** `.

Ceci est du point de vue de **NodeJS** . Dans d'autres exemples, je montrerai comment nous pouvons envoyer des données personnalisées, des icônes ..etc dans notre message push.

PS: vous pouvez trouver la démo complète ici.

Lire Envoyer une notification Web en ligne: https://riptutorial.com/fr/node-js/topic/6333/envoyer-une-notification-web

Chapitre 36: Eventloop

Introduction

Dans cet article, nous allons discuter de la manière dont le concept Eventloop a émergé et de la manière dont il peut être utilisé pour les serveurs hautes performances et les applications pilotées par des événements telles que les interfaces graphiques.

Examples

Comment le concept de boucle d'événement a évolué.

Eventloop en pseudo-code

Une boucle d'événement est une boucle qui attend des événements et réagit à ces événements

```
while true:
    wait for something to happen
    react to whatever happened
```

Exemple de serveur HTTP mono-thread sans boucle d'événement

```
    while true:
    socket = wait for the next TCP connection
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
```

Voici une forme simple d'un serveur HTTP qui est un thread unique mais pas de boucle d'événement. Le problème ici est qu'il attend que chaque requête soit terminée avant de commencer à traiter la suivante. S'il faut un certain temps pour lire les en-têtes de requête HTTP ou pour extraire le fichier du disque, nous devrions pouvoir commencer à traiter la requête suivante en attendant que cela se termine.

La solution la plus courante consiste à rendre le programme multithread.

Exemple de serveur HTTP multithread sans

boucle d'événement

```
function handle_connection(socket):
    read the HTTP request headers from (socket)
    file_contents = fetch the requested file from disk
    write the HTTP response headers to (socket)
    write the (file_contents) to (socket)
    close(socket)
while true:
    socket = wait for the next TCP connection
    spawn a new thread doing handle_connection(socket)
```

Maintenant, nous avons rendu notre petit serveur HTTP multi-thread. De cette façon, nous pouvons immédiatement passer à la requête suivante car la requête en cours est exécutée dans un thread d'arrière-plan. De nombreux serveurs, y compris Apache, utilisent cette approche.

Mais ce n'est pas parfait. Une limitation est que vous ne pouvez générer que beaucoup de threads. Pour les charges de travail où vous avez un grand nombre de connexions, mais chaque connexion ne nécessite qu'une attention de temps en temps, le modèle multi-thread ne fonctionnera pas très bien. La solution pour ces cas est d'utiliser une boucle d'événement:

Exemple de serveur HTTP avec boucle d'événements

```
while true:
    event = wait for the next event to happen
    if (event.type == NEW_TCP_CONNECTION):
        conn = new Connection
        conn.socket = event.socket
        start reading HTTP request headers from (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_SOCKET):
        conn = event.userdata
        start fetching the requested file from disk with userdata = (conn)
    else if (event.type == FINISHED_READING_FROM_DISK):
        conn = event.userdata
        conn.file_contents = the data we fetched from disk
        conn.current_state = "writing headers"
        start writing the HTTP response headers to (conn.socket) with userdata = (conn)
    else if (event.type == FINISHED_WRITING_TO_SOCKET):
        conn = event.userdata
        if (conn.current_state == "writing headers"):
            conn.current_state = "writing file contents"
            start writing (conn.file_contents) to (conn.socket) with userdata = (conn)
        else if (conn.current_state == "writing file contents"):
            close(conn.socket)
```

Espérons que ce pseudo-code soit intelligible. Voici ce qui se passe: nous attendons que les choses se passent. Chaque fois qu'une nouvelle connexion est créée ou qu'une connexion existante nécessite notre attention, nous nous en occupons, puis revenons à l'attente. De cette façon, nous fonctionnons bien quand il y a beaucoup de connexions et que chacune d'elles

nécessite rarement une attention.

Dans une application réelle (pas de pseudocode) fonctionnant sous Linux, la partie "Attendre l'événement suivant" serait implémentée en appelant l'appel système poll () ou epoll (). Les parties "commencer à lire / écrire quelque chose dans un socket" seraient implémentées en appelant les appels système recv () ou send () en mode non bloquant.

Référence:

[1]. "Comment fonctionne une boucle d'événement?" [En ligne]. Disponible: https://www.quora.com/How-does-an-event-loop-work

Lire Eventloop en ligne: https://riptutorial.com/fr/node-js/topic/8652/eventloop

Chapitre 37: Évitez le rappel de l'enfer

Examples

Module asynchrone

La source est disponible pour téléchargement à partir de GitHub. Alternativement, vous pouvez installer en utilisant npm:

$ npm install --save async

En plus d'utiliser Bower:

$ bower install async

Exemple:

```
var async = require("async");
async.parallel([
    function(callback) { ... },
    function(callback) { ... }
], function(err, results) {
    // optional callback
});
```

Module asynchrone

Heureusement, des bibliothèques comme Async.js existent pour essayer de limiter le problème. Async ajoute une couche mince de fonctions au-dessus de votre code, mais peut considérablement réduire la complexité en évitant l'imbrication de rappel.

Il existe de nombreuses méthodes d'assistance dans Async qui peuvent être utilisées dans différentes situations, telles que les séries, les parallèles, les cascades, etc. Chaque fonction a un cas d'utilisation spécifique.

Aussi bon que Async est, comme tout, ce n'est pas parfait. Il est très facile de se laisser emporter par la combinaison de séries, de parallèles, pour toujours, etc. Veillez à ne pas optimiser prématurément. Juste parce que quelques tâches asynchrones peuvent être exécutées en parallèle ne signifie pas toujours qu'elles le devraient. En réalité, Node étant un seul thread, l'exécution de tâches en parallèle lors de l'utilisation d'Async ne génère que peu ou pas de performances.

La source est disponible pour téléchargement sur https://github.com/caolan/async . Alternativement, vous pouvez installer en utilisant npm:

$ npm install --save async

En plus d'utiliser Bower:

$ bower install async

Exemple de chute d'eau d'Async:

```
var fs = require('fs');
var async = require('async');

var myFile = '/tmp/test';

async.waterfall([
    function(callback) {
        fs.readFile(myFile, 'utf8', callback);
    },
    function(txt, callback) {
        txt = txt + '\nAppended something!';
        fs.writeFile(myFile, txt, callback);
    }
], function (err, result) {
    if(err) return console.log(err);
    console.log('Appended text!');
});
```

Lire Évitez le rappel de l'enfer en ligne: https://riptutorial.com/fr/node-js/topic/10045/evitez-le-rappel-de-l-enfer

Chapitre 38: Exécuter des fichiers ou des commandes avec des processus enfants

Syntaxe

- child_process.exec (commande [, options] [, callback])
- child_process.execFile (fichier [, args] [, options] [, callback])
- child_process.fork (modulePath [, args] [, options])
- child_process.spawn (commande [, args] [, options])
- child_process.execFileSync (fichier [, args] [, options])
- child_process.execSync (commande [, options])
- child_process.spawnSync (commande [, args] [, options])

Remarques

Lorsque vous traitez des processus enfants, toutes les méthodes asynchrones `ChildProcess` une instance de `ChildProcess` , alors que toutes les versions synchrones `ChildProcess` la sortie de tout ce qui a été exécuté. Comme d'autres opérations synchrones dans Node.js, si une erreur survient, elle *sera* lancée.

Examples

Création d'un nouveau processus pour exécuter une commande

Pour générer un nouveau processus dans lequel vous avez besoin d'une sortie sans *tampon* (par exemple, des processus de longue durée qui peuvent imprimer des résultats sur une `child_process.spawn()` période plutôt que d'imprimer et de quitter immédiatement), utilisez `child_process.spawn()` .

Cette méthode génère un nouveau processus en utilisant une commande donnée et un tableau d'arguments. La valeur de retour est une instance de `ChildProcess` , qui à son tour fournit les propriétés `stdout` et `stderr` . Ces deux flux sont des instances de `stream.Readable` .

Le code suivant est équivalent à l'utilisation de la commande `ls -lh /usr` .

```
const spawn = require('child_process').spawn;
const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.log(`stderr: ${data}`);
});
```

```
ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

Un autre exemple de commande:

```
zip -0vr "archive" ./image.png
```

Peut-être écrit comme:

```
spawn('zip', ['-0vr', '"archive"', './image.png']);
```

Création d'un shell pour exécuter une commande

Pour exécuter une commande dans un shell dans lequel vous avez besoin d'une sortie en mémoire tampon (c'est-à-dire que ce n'est pas un flux), utilisez `child_process.exec` . Par exemple, si vous souhaitez exécuter la commande `cat *.js file | wc -l` , sans options, cela ressemblerait à ceci:

```
const exec = require('child_process').exec;
exec('cat *.js file | wc -l', (err, stdout, stderr) => {
  if (err) {
    console.error(`exec error: ${err}`);
    return;
  }

  console.log(`stdout: ${stdout}`);
  console.log(`stderr: ${stderr}`);
});
```

La fonction accepte jusqu'à trois paramètres:

```
child_process.exec(command[, options][, callback]);
```

Le paramètre de commande est une chaîne et est requis, tandis que l'objet options et le rappel sont tous deux facultatifs. Si aucun objet d'options n'est spécifié, alors `exec` utilise par défaut ce qui suit:

```
{
  encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200*1024,
  killSignal: 'SIGTERM',
  cwd: null,
  env: null
}
```

L'objet options prend également en charge un paramètre `shell` , par défaut `/bin/sh` sous UNIX et `cmd.exe` sous Windows, une option `uid` permettant de définir l'identité utilisateur du processus et une option `gid` pour l'identité du groupe.

Le rappel, appelé lors de l'exécution de la commande, est appelé avec les trois arguments `(err, stdout, stderr)` . Si la commande s'exécute avec succès, `err` sera `null` , sinon ce sera une instance de `Error` , `err.code` étant le code de sortie du processus et `err.signal` étant le signal envoyé pour le terminer.

Les arguments `stdout` et `stderr` sont la sortie de la commande. Il est décodé avec le codage spécifié dans l'objet options (default: `string`), mais peut être renvoyé sous forme d'objet `Buffer` .

Il existe également une version synchrone de `exec` , qui est `execSync` . La version synchrone ne prend pas de rappel et renverra `stdout` au lieu d'une instance de `ChildProcess` . Si la version synchrone rencontre une erreur, elle *lancera* et arrêtera votre programme. Cela ressemble à ceci:

```
const execSync = require('child_process').execSync;
const stdout = execSync('cat *.js file | wc -l');
console.log(`stdout: ${stdout}`);
```

Création d'un processus pour exécuter un exécutable

Si vous souhaitez exécuter un fichier, tel qu'un exécutable, utilisez `child_process.execFile` . Au lieu de créer un shell comme pourrait le faire `child_process.exec` , cela créera directement un nouveau processus, ce qui est légèrement plus efficace que d'exécuter une commande. La fonction peut être utilisée comme ceci:

```
const execFile = require('child_process').execFile;
const child = execFile('node', ['--version'], (err, stdout, stderr) => {
  if (err) {
    throw err;
  }

  console.log(stdout);
});
```

Contrairement à `child_process.exec` , cette fonction accepte jusqu'à quatre paramètres, le second paramètre étant un tableau d'arguments que vous souhaitez fournir à l'exécutable:

```
child_process.execFile(file[, args][, options][, callback]);
```

Sinon, les options et le format de rappel sont identiques à `child_process.exec` . Il en va de même pour la version synchrone de la fonction:

```
const execFileSync = require('child_process').execFileSync;
const stdout = execFileSync('node', ['--version']);
console.log(stdout);
```

Lire Exécuter des fichiers ou des commandes avec des processus enfants en ligne: https://riptutorial.com/fr/node-js/topic/2726/executer-des-fichiers-ou-des-commandes-avec-des-processus-enfants

Chapitre 39: Exécution de node.js en tant que service

Introduction

Contrairement à de nombreux serveurs Web, Node n'est pas installé en tant que service par défaut. Mais en production, il est préférable de le faire fonctionner comme un démon, géré par un système init.

Examples

Node.js en tant que systemd dæmon

systemd est le système d'initialisation *de facto* dans la plupart des distributions Linux. Une fois que Node a été configuré pour s'exécuter avec systemd, il est possible d'utiliser la commande de `service` pour le gérer.

Tout d'abord, il faut un fichier de configuration, créons-le. Pour les distributions basées sur Debian, ce sera dans `/etc/systemd/system/node.service`

```
[Unit]
Description=My super nodejs app

[Service]
# set the working directory to have consistent relative paths
WorkingDirectory=/var/www/app

# start the server file (file is relative to WorkingDirectory here)
ExecStart=/usr/bin/node serverCluster.js

# if process crashes, always try to restart
Restart=always

# let 500ms between the crash and the restart
RestartSec=500ms

# send log tot syslog here (it doesn't compete with other log config in the app itself)
StandardOutput=syslog
StandardError=syslog

# nodejs process name in syslog
SyslogIdentifier=nodejs

# user and group starting the app
User=www-data
Group=www-data

# set the environement (dev, prod…)
Environment=NODE_ENV=production
```

```
[Install]
# start node at multi user system level (= sysVinit runlevel 3)
WantedBy=multi-user.target
```

Il est maintenant possible de démarrer, d'arrêter et de redémarrer respectivement l'application avec:

```
service node start
service node stop
service node restart
```

Pour que systemd démarre automatiquement le noeud au démarrage, tapez simplement: `systemctl enable node`.

C'est tout, le noeud fonctionne désormais comme un dæmon.

Lire Exécution de node.js en tant que service en ligne: https://riptutorial.com/fr/node-js/topic/9258/execution-de-node-js-en-tant-que-service

Chapitre 40: Exiger()

Introduction

Cette documentation se concentre sur l'explication des utilisations et de l'instruction `require()` que NodeJS inclut dans leur langage.

Require est une importation de certains fichiers ou packages utilisés avec les modules de NodeJS. Il est utilisé pour améliorer la structure du code et ses utilisations. `require()` est utilisé sur les fichiers installés localement, avec une route directe à partir du fichier `require` .

Syntaxe

- module.exports = {testFunction: testFunction};
- var test_file = require ('./ testFile.js'); // `testFile` un fichier nommé `testFile`
- test_file.testFunction (our_data); // Laissez `testFile` avoir la fonction `testFunction`

Remarques

L'utilisation de `require()` permet au code d'être structuré de manière similaire à l'utilisation des classes et des méthodes publiques par Java. Si une fonction est `.export` « ed, il peut être `require` » ed dans un autre fichier à utiliser. Si un fichier n'est pas `.export` 'ed, il ne peut pas être utilisé dans un autre fichier.

Examples

Début require () utilisation avec une fonction et un fichier

Require est une instruction interprétée par Node comme une fonction `getter` . Par exemple, disons que vous avez un fichier nommé `analysis.js` et que l'intérieur de votre fichier ressemble à ceci:

```
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}
```

Ce fichier contient uniquement la méthode, `analyzeWeather(weather_data)` . Si nous voulons utiliser cette fonction, elle doit être soit utilisée dans ce fichier, soit copiée dans le fichier par lequel elle veut être utilisée. Cependant, Node a inclus un outil très utile pour aider à l'organisation du code et des fichiers, à savoir les modules .

Pour utiliser notre fonction, nous devons d’abord `export` la fonction via une instruction au début. Notre nouveau fichier ressemble à ceci,

```
module.exports = {
  analyzeWeather: analyzeWeather
}
function analyzeWeather(weather_data) {
  console.log('Weather information for ' + weather_data.time + ': ');
  console.log('Rainfall: ' + weather_data.precip);
  console.log('Temperature: ' + weather_data.temp);
  //More weather_data analysis/printing...
}
```

Avec cette petite instruction `module.exports` , notre fonction est maintenant prête à être utilisée en dehors du fichier. Il ne reste plus qu'à utiliser `require()` .

Lorsque vous `require` d'une fonction ou d'un fichier, la syntaxe est très similaire. Il est généralement effectué au début du fichier et défini sur `var` ou `const` pour être utilisé dans tout le fichier. Par exemple, nous avons un autre fichier (au même niveau que `analyze.js` nommé `handleWeather.js` qui ressemble à ceci,

```
const analysis = require('./analysis.js');

weather_data = {
  time: '01/01/2001',
  precip: 0.75,
  temp: 78,
  //More weather data...
};
analysis.analyzeWeather(weather_data);
```

Dans ce fichier, nous utilisons `require()` pour récupérer notre fichier `analysis.js` . Lorsqu'elle est utilisée, nous appelons simplement la variable ou la constante affectée à cette `require` et utilisons la fonction qui est exportée.

Début de require () utilisation avec un paquet NPM

Le `require` de noeud est également très utile lorsqu'il est utilisé en association avec un package NPM . Supposons, par exemple, que vous `getWeather.js` utiliser le paquetage NPM `require` dans un fichier nommé `getWeather.js` . Après l' installation de votre package par NPM via votre ligne de commande (`git install request`), vous êtes prêt à l'utiliser. Votre fichier `getWeather.js` pourrait ressembler à ceci,

```
var https = require('request');

//Construct your url variable...
https.get(url, function(error, response, body) {
  if (error) {
    console.log(error);
  } else {
    console.log('Response => ' + response);
    console.log('Body => ' + body);
  }
});
```

Lorsque ce fichier est exécuté, il d' abord `require` de « s (importations) le paquet que vous venez

d' installer appelé `request` . À l'intérieur du fichier de `request` , il y a beaucoup de fonctions auxquelles vous avez maintenant accès, dont l'une s'appelle `get` . Dans les deux lignes suivantes, la fonction est utilisée pour effectuer une requête HTTP GET .

Lire Exiger() en ligne: https://riptutorial.com/fr/node-js/topic/10742/exiger--

Chapitre 41: Exportation et consommation de modules

Remarques

Alors que tout dans Node.js est généralement effectué de manière asynchrone, `require()` ne fait pas partie de ces choses. Dans la pratique, les modules ne doivent être chargés qu'une seule fois, il s'agit d'une opération de blocage qui doit être utilisée correctement.

Les modules sont mis en cache après la première fois qu'ils sont chargés. Si vous modifiez un module en développement, vous devrez supprimer son entrée dans le cache du module pour pouvoir utiliser les nouvelles modifications. Cela étant dit, même si un module est effacé du cache du module, le module lui-même n'est pas récupéré, il convient donc de veiller à son utilisation dans les environnements de production.

Examples

Chargement et utilisation d'un module

Un module peut être "importé" ou "requis" par la fonction `require()` . Par exemple, pour charger le module `http` fourni avec Node.js, vous pouvez utiliser les éléments suivants:

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

Outre les modules fournis avec le runtime, vous pouvez également exiger des modules installés à partir de npm, tels qu'express. Si vous avez déjà installé express sur votre système via `npm install express` , vous pouvez simplement écrire:

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

Vous pouvez également inclure des modules que vous avez écrits vous-même dans le cadre de votre application. Dans ce cas, pour inclure un fichier nommé `lib.js` dans le même répertoire que le fichier actuel:

```
const mylib = require('./lib');
```

Notez que vous pouvez omettre l'extension et que `.js` sera `.js` . Une fois que vous avez chargé un module, la variable est remplie avec un objet qui contient les méthodes et les propriétés publiées à partir du fichier requis. Un exemple complet:

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

// The `http` module has the property `STATUS_CODES`
console.log(http.STATUS_CODES[404]); // outputs 'Not Found'
```

```
// Also contains `createServer()`
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<html><body>Module Test</body></html>');
  res.end();
}).listen(80);
```

Créer un module hello-world.js

Node fournit l'interface `module.exports` pour exposer les fonctions et les variables à d'autres fichiers. La manière la plus simple de le faire est d'exporter un seul objet (fonction ou variable), comme indiqué dans le premier exemple.

bonjour-world.js

```
module.exports = function(subject) {
    console.log('Hello ' + subject);
};
```

Si nous ne voulons pas que l'exportation entière soit un objet unique, nous pouvons exporter des fonctions et des variables en tant que propriétés de l'objet d' `exports` . Les trois exemples suivants le démontrent tous de manière légèrement différente:

- hello-venus.js: la définition de la fonction est faite séparément puis ajoutée en tant que propriété de `module.exports`
- hello-jupiter.js: les définitions de fonctions sont directement définies comme la valeur des propriétés de `module.exports`
- hello-mars.js: la définition de la fonction est directement déclarée comme une propriété des `exports` qui est une version courte de `module.exports`

hello-venus.js

```
function hello(subject) {
    console.log('Venus says Hello ' + subject);
}

module.exports = {
    hello: hello
};
```

bonjour-jupiter.js

```
module.exports = {
    hello: function(subject) {
      console.log('Jupiter says hello ' + subject);
    },

    bye: function(subject) {
      console.log('Jupiter says goodbye ' + subject);
    }
};
```

bonjour-mars.js

```
exports.hello = function(subject) {
    console.log('Mars says Hello ' + subject);
};
```

Module de chargement avec nom de répertoire

Nous avons un répertoire nommé `hello` qui inclut les fichiers suivants:

index.js

```
// hello/index.js
module.exports = function(){
    console.log('Hej');
};
```

main.js

```
// hello/main.js
// We can include the other files we've defined by using the `require()` method
var hw = require('./hello-world.js'),
    hm = require('./hello-mars.js'),
    hv = require('./hello-venus.js'),
    hj = require('./hello-jupiter.js'),
    hu = require('./index.js');

// Because we assigned our function to the entire `module.exports` object, we
// can use it directly
hw('World!'); // outputs "Hello World!"

// In this case, we assigned our function to the `hello` property of exports, so we must
// use that here too
hm.hello('Solar System!'); // outputs "Mars says Hello Solar System!"

// The result of assigning module.exports at once is the same as in hello-world.js
hv.hello('Milky Way!'); // outputs "Venus says Hello Milky Way!"

hj.hello('Universe!'); //  outputs "Jupiter says hello Universe!"
hj.bye('Universe!'); // outputs "Jupiter says goodbye Universe!"

hu(); //output 'hej'
```

Invalider le cache du module

En développement, vous pouvez constater que l'utilisation de `require()` sur le même module plusieurs fois renvoie toujours le même module, même si vous avez apporté des modifications à ce fichier. Cela est dû au fait que les modules sont mis en cache la première fois qu'ils sont chargés et que tous les chargements ultérieurs de module sont chargés à partir du cache.

Pour contourner ce problème, vous devrez `delete` l'entrée dans le cache. Par exemple, si vous avez chargé un module:

```
var a = require('./a');
```

Vous pouvez ensuite supprimer l'entrée de cache:

```
var rpath = require.resolve('./a.js');
delete require.cache[rpath];
```

Et puis demandez à nouveau le module:

```
var a = require('./a');
```

Notez que cela n'est pas recommandé en production car la `delete` uniquement la référence au module chargé, pas les données chargées elles-mêmes. Le module n'étant pas récupéré, une mauvaise utilisation de cette fonctionnalité peut entraîner une fuite de mémoire.

Construire vos propres modules

Vous pouvez également référencer un objet pour l'exporter publiquement et ajouter en continu des méthodes à cet objet:

```
const auth = module.exports = {}
const config = require('../config')
const request = require('request')

auth.email = function (data, callback) {
  // Authenticate with an email address
}

auth.facebook = function (data, callback) {
  // Authenticate with a Facebook account
}

auth.twitter = function (data, callback) {
  // Authenticate with a Twitter account
}

auth.slack = function (data, callback) {
  // Authenticate with a Slack account
}

auth.stack_overflow = function (data, callback) {
  // Authenticate with a Stack Overflow account
}
```

Pour utiliser l'un de ces éléments, il vous suffit de demander le module comme vous le feriez normalement:

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

module.exports = function (req, res, next) {
  auth.facebook(req.body, function (err, user) {
    if (err) return next(err)
```

```
    req.user = user
    next()
  })
}
```

Chaque module injecté une seule fois

NodeJS exécute le module uniquement la première fois que vous en avez besoin. Toute autre fonction requise réutilisera le même objet, donc n'exécutera pas le code dans le module une autre fois. Aussi Node met en cache les modules la première fois qu'ils sont chargés en utilisant require. Cela réduit le nombre de lectures de fichiers et accélère l'application.

`myModule.js`

```
console.log(123) ;
exports.var1 = 4 ;
```

index.js

```
var a=require('./myModule') ; // Output 123
var b=require('./myModule') ; // No output
console.log(a.var1) ; // Output 4
console.log(b.var1) ; // Output 4
a.var2 = 5 ;
console.log(b.var2) ; // Output 5
```

Module de chargement depuis node_modules

Les modules peuvent être `require` sans utiliser de chemins relatifs en les plaçant dans un répertoire spécial appelé `node_modules` .

Par exemple, pour `require` un module appelé `foo` depuis un fichier `index.js` , vous pouvez utiliser la structure de répertoires suivante:

```
index.js
 \- node_modules
  \- foo
   |- foo.js
   \- package.json
```

Les modules doivent être placés dans un répertoire, avec un fichier `package.json` . Le champ `main` du fichier `package.json` doit pointer vers le point d'entrée de votre module - c'est le fichier qui est importé lorsque les utilisateurs ont `require('your-module')` . `main` défaut à `index.js` si non fourni. Sinon, vous pouvez vous référer à des fichiers par rapport à votre module simplement en ajoutant le chemin par rapport au `require` appel: `require('your-module/path/to/file')` .

Les modules peuvent également être `require` depuis les répertoires `node_modules` dans la hiérarchie du système de fichiers. Si nous avons la structure de répertoires suivante:

```
my-project
```

```
\- node_modules
 |- foo   // the foo module
  \- ...
 \- baz   // the baz module
  \- node_modules
   \- bar   // the bar module
```

Nous pourrons `require` le module `foo` de tout fichier dans la `bar` utilisant `require('foo')` .

Notez que ce nœud ne correspondra qu'au module le plus proche du fichier dans la hiérarchie du système de fichiers, à partir de (répertoire courant du fichier / modules_noeud). Le noeud correspond aux répertoires de la racine du système de fichiers.

Vous pouvez installer de nouveaux modules à partir du registre npm ou d'autres registres npm, ou créer le vôtre.

Dossier en tant que module

Les modules peuvent être divisés en plusieurs fichiers .js dans le même dossier. Un exemple dans un dossier *my_module* :

function_one.js

```
module.exports = function() {
  return 1;
}
```

function_two.js

```
module.exports = function() {
  return 2;
}
```

index.js

```
exports.f_one = require('./function_one.js');
exports.f_two = require('./function_two.js');
```

Un module comme celui-ci est utilisé en s'y référant par le nom du dossier:

```
var split_module = require('./my_module');
```

Notez que si vous en aviez besoin en omettant `./` ou toute indication d'un chemin vers un dossier depuis l'argument de la fonction require, Node essaiera de charger un module à partir du dossier *node_modules* .

Sinon, vous pouvez créer dans le même dossier un fichier `package.json` avec ces contenus:

```
{
    "name": "my_module",
```

```
    "main": "./your_main_entry_point.js"
}
```

De cette façon, vous n'êtes pas obligé de nommer le fichier principal du module "index".

Lire Exportation et consommation de modules en ligne: https://riptutorial.com/fr/node-js/topic/547/exportation-et-consommation-de-modules

Chapitre 42: Exportation et importation d'un module dans node.js

Examples

Utiliser un module simple dans node.js

Qu'est-ce qu'un module node.js (lien vers l'article):

> Un module encapsule le code associé dans une seule unité de code. Lors de la création d'un module, cela peut être interprété comme déplaçant toutes les fonctions associées dans un fichier.

Voyons maintenant un exemple. Imaginez que tous les fichiers se trouvent dans le même répertoire:

Fichier: `printer.js`

```
"use strict";

exports.printHelloWorld = function (){
    console.log("Hello World!!!");
}
```

Une autre façon d'utiliser les modules:

Fichier `animals.js`

```
"use strict";

module.exports = {
    lion: function() {
        console.log("ROAARR!!!");
    }

};
```

Fichier: `app.js`

Exécutez ce fichier en accédant à votre répertoire et en tapant: `node app.js`

```
"use strict";

//require('./path/to/module.js') node which module to load
var printer = require('./printer');
var animals = require('./animals');

printer.printHelloWorld(); //prints "Hello World!!!"
animals.lion(); //prints "ROAARR!!!"
```

Utilisation des importations dans ES6

Node.js est construit contre les versions modernes de V8. En se mettant à jour avec les dernières versions de ce moteur, nous nous assurons que les nouvelles fonctionnalités de la spécification JavaScript ECMA-262 sont apportées aux développeurs Node.js en temps opportun, ainsi que des améliorations continues des performances et de la stabilité.

Toutes les fonctionnalités ECMAScript 2015 (ES6) sont divisées en trois groupes pour les fonctions d'expédition, de transfert et de progression:

Toutes les fonctionnalités d'expédition, que V8 considère comme stables, sont activées par défaut sur Node.js et ne nécessitent AUCUN type d'indicateur d'exécution. Les fonctions par étapes, qui sont des fonctions presque terminées qui ne sont pas considérées comme stables par l'équipe V8, nécessitent un indicateur d'exécution: --harmony. Les fonctions en cours peuvent être activées individuellement par leur drapeau d'harmonie respectif, bien que cela soit fortement déconseillé, sauf à des fins de test. Remarque: ces indicateurs sont exposés par V8 et risquent de changer sans préavis.

Actuellement, ES6 prend en charge les instructions d'importation en mode natif .

Donc, si nous avons un fichier appelé `fun.js` …

```
export default function say(what){
  console.log(what);
}

export function sayLoud(whoot) {
  say(whoot.toUpperCase());
}
```

… Et s'il y avait un autre fichier nommé `app.js` lequel nous souhaitons utiliser nos fonctions précédemment définies, il existe trois manières de les importer.

Importer par défaut

```
import say from './fun';
say('Hello Stack Overflow!!');  // Output: Hello Stack Overflow!!
```

Importe la fonction `say()` car elle est marquée comme export par défaut dans le fichier source (`export default …`)

Importations nommées

```
import { sayLoud } from './fun';
sayLoud('JS modules are awesome.'); // Output: JS MODULES ARE AWESOME.
```

Les importations nommées nous permettent d'importer exactement les parties d'un module dont nous avons réellement besoin. Nous faisons cela en les nommant explicitement. Dans notre cas, en nommant `sayLoud` entre accolades dans l'instruction import.

Importation groupée

```
import * as i from './fun';
i.say('What?'); // Output: What?
i.sayLoud('Whoot!'); // Output: WHOOT!
```

Si nous voulons tout avoir, c'est la voie à suivre. En utilisant la syntaxe `* as i` nous avons un objet `import` contenant un objet `i` contenant toutes les exportations de notre module `fun` comme propriétés correspondantes.

Les chemins

N'oubliez pas que vous devez explicitement marquer vos chemins d'importation en tant que chemins *relatifs,* même si le fichier à importer réside dans le même répertoire que le fichier dans lequel vous importez en utilisant `./` . Importations à partir de chemins non préfixés comme

```
import express from 'express';
```

sera recherché dans les dossiers `node_modules` locaux et globaux et `node_modules` une erreur si aucun module correspondant n'est trouvé.

Exportation avec la syntaxe ES6

Ceci est l'équivalent de l'autre exemple, mais en utilisant plutôt ES6.

```
export function printHelloWorld() {
  console.log("Hello World!!!");
}
```

Lire Exportation et importation d'un module dans node.js en ligne: https://riptutorial.com/fr/node-js/topic/1173/exportation-et-importation-d-un-module-dans-node-js

Chapitre 43: forgeron

Examples

Construire un blog simple

En supposant que noeud et npm soient installés et disponibles, créez un dossier de projet avec un `package.json` valide. Installez les dépendances nécessaires:

```
npm install --save-dev metalsmith metalsmith-in-place handlebars
```

Créez un fichier appelé `build.js` à la racine de votre dossier de projet, contenant les éléments suivants:

```
var metalsmith = require('metalsmith');
var handlebars = require('handlebars');
var inPlace = require('metalsmith-in-place');

Metalsmith(__dirname)
  .use(inPlace('handlebars'))
  .build(function(err) {
    if (err) throw err;
    console.log('Build finished!');
  });
```

Créez un dossier appelé `src` à la racine de votre dossier de projet. Créez `index.html` dans `src` , contenant les éléments suivants:

```
---
title: My awesome blog
---
<h1>{{ title }}</h1>
```

L'exécution du `node build.js` va maintenant générer tous les fichiers dans `src` . Après avoir exécuté cette commande, vous aurez `index.html` dans votre dossier de génération, avec le contenu suivant:

```
<h1>My awesome blog</h1>
```

Lire forgeron en ligne: https://riptutorial.com/fr/node-js/topic/6111/forgeron

Chapitre 44: Frameworks de tests unitaires

Examples

Mocha synchrone

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      expect([ 1, 2, 3 ].length).to.be.equal(3)
    })
  })
})
```

Mocha asynchrone (rappel)

```
var expect = require("chai").expect;
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function(done) {
      testSomething(err => {
        expect(err).to.not.be.equal(null)
        done()
      })
    })
  })
})
```

Mocha asynchrone (Promise)

```
describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', function() {
      return doSomething().then(result => {
         expect(result).to.be.equal('hello world')
      })
    })
  })
})
```

Mocha Asynchrone (asynchrone / wait)

```
const { expect } = require('chai')

describe('Suite Name', function() {
  describe('#method()', function() {
    it('should run without an error', async function() {
      const result = await answerToTheUltimateQuestion()
      expect(result).to.be.equal(42)
    })
  })
```

```
})
```

Lire Frameworks de tests unitaires en ligne: https://riptutorial.com/fr/node-js/topic/6731/frameworks-de-tests-unitaires

Chapitre 45: Frameworks NodeJS

Examples

Web Server Frameworks

Express

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

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

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Koa

```
var koa = require('koa');
var app = koa();

app.use(function *(next){
  var start = new Date;
  yield next;
  var ms = new Date - start;
  console.log('%s %s - %s', this.method, this.url, ms);
});

app.use(function *(){
  this.body = 'Hello World';
});

app.listen(3000);
```

Cadres d'interface de ligne de commande

Commander.js

```
var program = require('commander');

program
  .version('0.0.1')

program
  .command('hi')
  .description('initialize project configuration')
  .action(function(){
```

```
        console.log('Hi my Friend!!!');
});

program
  .command('bye [name]')
  .description('initialize project configuration')
  .action(function(name){
        console.log('Bye ' + name + '. It was good to see you!');
});

program
  .command('*')
  .action(function(env){
    console.log('Enter a Valid command');
    terminate(true);
});

program.parse(process.argv);
```

Vorpal.js

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

vorpal
  .command('foo', 'Outputs "bar".')
  .action(function(args, callback) {
    this.log('bar');
    callback();
  });

vorpal
  .delimiter('myapp$')
  .show();
```

Lire Frameworks NodeJS en ligne: https://riptutorial.com/fr/node-js/topic/6042/frameworks-nodejs

Chapitre 46: Garder une application de noeud en cours d'exécution

Examples

Utiliser PM2 comme gestionnaire de processus

PM2 vous permet d'exécuter vos scripts nodejs pour toujours. Dans le cas où votre application se bloque, PM2 le redémarrera également pour vous.

Installez globalement PM2 pour gérer vos instances de nodejs

```
npm install pm2 -g
```

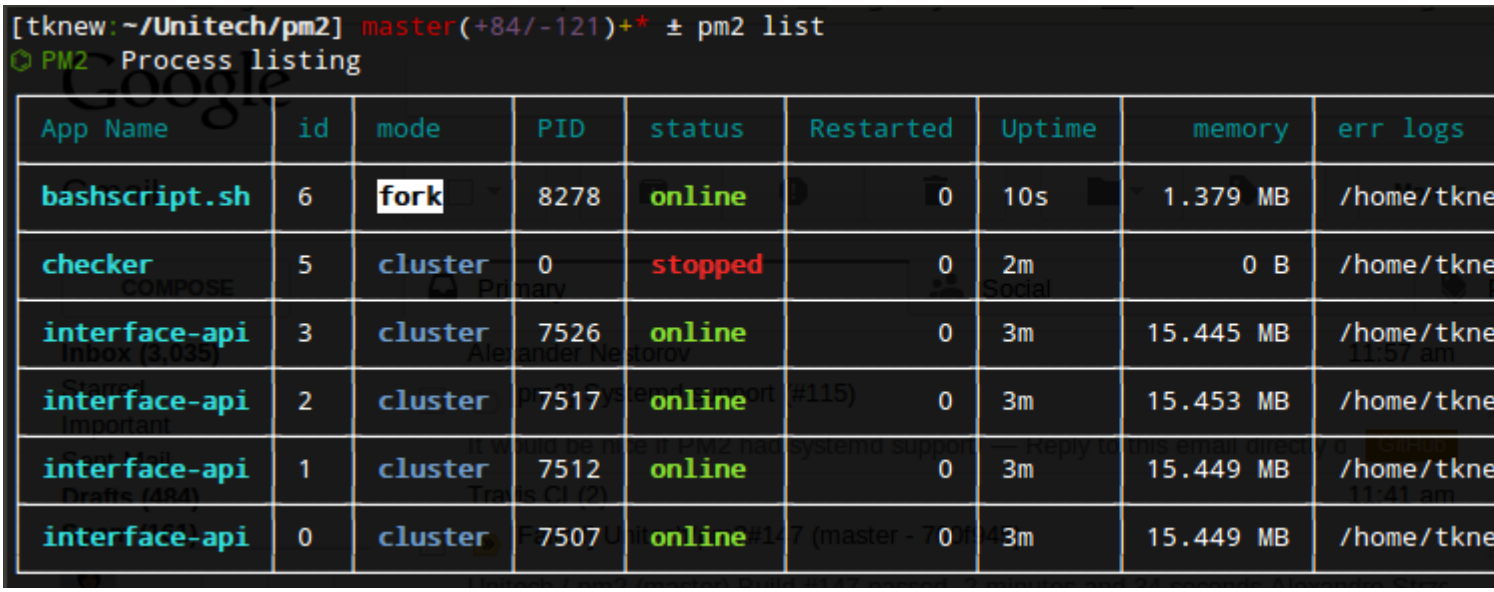
Accédez au répertoire dans lequel réside votre script nodejs et exécutez la commande suivante à chaque fois que vous souhaitez démarrer une instance nodejs à surveiller par pm2:

```
pm2 start server.js --name "app1"
```

Commandes utiles pour surveiller le processus

1. Liste toutes les instances de nodejs gérées par pm2

   ```
   pm2 list
   ```



2. Arrêtez une instance nodejs particulière

   ```
   pm2 stop <instance named>
   ```

3. Supprimer une instance nodejs particulière

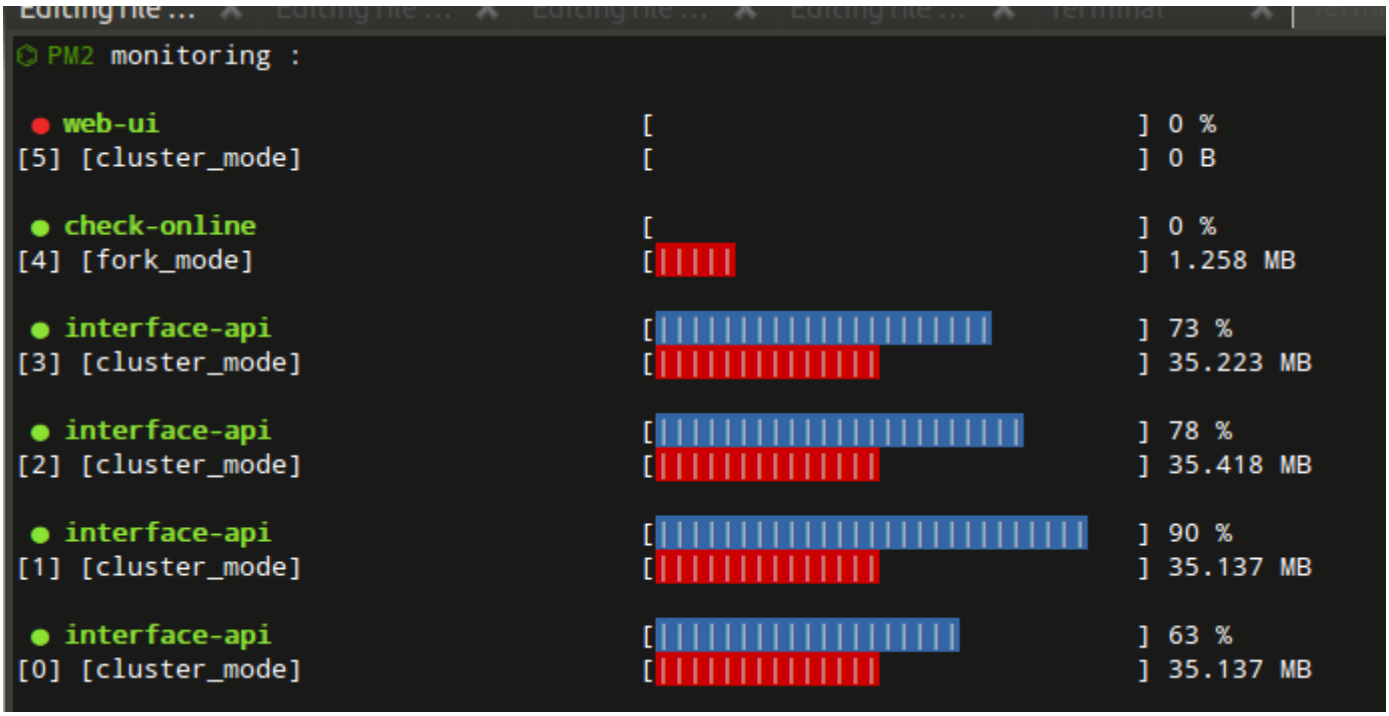
```
pm2 delete <instance name>
```

4. Redémarrer une instance nodejs particulière

```
pm2 restart <instance name>
```

5. Surveillance de toutes les instances de nodejs

```
pm2 monit
```



6. Arrêter pm2

```
pm2 kill
```

7. Au lieu de redémarrer, ce qui tue et redémarre le processus, reload réalise un rechargement de 0 seconde

```
pm2 reload <instance name>
```

8. Regardes les connexions

```
pm2 logs <instance_name>
```

Lancer et arrêter un démon Forever

Pour démarrer le processus:

```
$ forever start index.js
warn:    --minUptime not set. Defaulting to: 1000ms
warn:    --spinSleepTime not set. Your script will exit if it does not stay up for at least
1000ms
info:    Forever processing file: index.js
```

Liste exécutant des instances Forever:

```
$ forever list
info:    Forever processes running

|data: | index | uid | command          | script      |forever pid|id   | logfile
|uptime        |
|------|-------|-----|------------------|-------------|-----------|-----|---------------------
---|--------------|
|data: | [0]   |f4Kt |/usr/bin/nodejs   | src/index.js|2131       |
2146|/root/.forever/f4Kt.log | 0:0:0:11.485 |
```

Arrêtez le premier processus:

```
$ forever stop 0

$ forever stop 2146

$ forever stop --uid f4Kt

$ forever stop --pidFile 2131
```

Fonctionnement continu avec nohup

Une alternative à toujours sur Linux est nohup.

Pour démarrer une instance Nohup

1. cd à l'emplacement du dossier `app.js` ou `www`
2. lance `nohup nodejs app.js &`

Pour tuer le processus

1. lancez `ps -ef|grep nodejs`
2. `kill -9 <the process number>`

Processus de gestion avec pour toujours

Installation

```
npm install forever -g
cd /node/project/directory
```

Coutumes

```
forever start app.js
```

Lire Garder une application de noeud en cours d'exécution en ligne: https://riptutorial.com/fr/node-js/topic/2820/garder-une-application-de-noeud-en-cours-d-execution

Chapitre 47: Gestion de la requête POST dans Node.js

Remarques

Node.js utilise des flux pour gérer les données entrantes.

Citant les docs,

> Un flux est une interface abstraite pour travailler avec des données en continu dans Node.js. Le module de flux fournit une API de base qui facilite la création d'objets qui implémentent l'interface de flux.

Pour gérer le corps de requête d'une requête POST, utilisez l'objet `request` , qui est un flux lisible. Les flux de données sont émis en tant qu'événements de `data` sur l'objet de `request` .

```
request.on('data', chunk => {
  buffer += chunk;
});
request.on('end', () => {
  // POST request body is now available as `buffer`
});
```

Créez simplement une chaîne de mémoire tampon vide et ajoutez les données de la mémoire tampon lors de la réception via `data` événements de `data` .

REMARQUE

1. Les données de tampon reçues sur `data` événements de `data` sont de type Buffer
2. Créez une nouvelle chaîne de tampon pour collecter les données mises en mémoire tampon à partir des événements de données **pour chaque requête,** c.-à-d. Créer `buffer` chaîne de `buffer` dans le gestionnaire de requêtes.

Examples

Exemple de serveur node.js qui gère uniquement les requêtes POST

```
'use strict';

const http = require('http');

const PORT = 8080;
const server = http.createServer((request, response) => {
  let buffer = '';
  request.on('data', chunk => {
    buffer += chunk;
  });
  request.on('end', () => {
```

```
    const responseString = `Received string ${buffer}`;
    console.log(`Responding with: ${responseString}`);
    response.writeHead(200, "Content-Type: text/plain");
    response.end(responseString);
  });
}).listen(PORT, () => {
  console.log(`Listening on ${PORT}`);
});
```

Lire Gestion de la requête POST dans Node.js en ligne: https://riptutorial.com/fr/node-js/topic/5676/gestion-de-la-requete-post-dans-node-js

Chapitre 48: Gestion des exceptions

Examples

Gestion des exceptions dans Node.Js

Node.js dispose de 3 méthodes de base pour gérer les exceptions / erreurs:

1. **essayer** - **attraper le** bloc
2. **erreur** comme premier argument d'un `callback`
3. `emit` un événement d' **erreur** en utilisant eventEmitter

try-catch est utilisé pour intercepter les exceptions lancées à partir de l'exécution du code synchrone. Si l'appelant (ou l'appelant de l'appelant, ...) a essayé / rattrapé, il peut alors détecter l'erreur. Si aucun des appelants n'a essayé, le programme se bloque.
Si vous utilisez try-catch sur une opération asynchrone et que l'exception est renvoyée par le rappel de la méthode asynchrone, elle ne sera pas interceptée par try-catch. Pour intercepter une exception de rappel d'opération asynchrone, il est préférable d'utiliser des *promesses* .
Exemple pour mieux comprendre

```
// ** Example - 1  **
function doSomeSynchronousOperation(req, res) {
    if(req.body.username === ''){
        throw new Error('User Name cannot be empty');
    }
    return true;
}

// calling the method above
try {
    // synchronous code
    doSomeSynchronousOperation(req, res)
catch(e) {
    //exception handled here
    console.log(e.message);
}

// ** Example - 2 **
function doSomeAsynchronousOperation(req, res, cb) {
    // imitating async operation
    return setTimeout(function(){
        cb(null, []);
    },1000);
}

try {
    // asynchronous code
    doSomeAsynchronousOperation(req, res, function(err, rs){
        throw new Error("async operation exception");
    })
} catch(e) {
     // Exception will not get handled here
     console.log(e.message);
```

```
}
// The exception is unhandled and hence will cause application to break
```

Les callbacks sont principalement utilisés dans Node.js car le callback délivre un événement de manière asynchrone. L'utilisateur vous transmet une fonction (le rappel) et vous l'appelez plus tard lorsque l'opération asynchrone est terminée.
Le schéma habituel est que le rappel est appelé comme un *rappel (err, result)* , où un seul des erreurs et des résultats est non nul, selon que l'opération a réussi ou échoué.

```
function doSomeAsynchronousOperation(req, res, callback) {
   setTimeout(function(){
        return callback(new Error('User Name cannot be empty'));
   }, 1000);
   return true;
}

doSomeAsynchronousOperation(req, res, function(err, result) {
   if (err) {
       //exception handled here
       console.log(err.message);
   }

   //do some stuff with valid data
});
```

emit Pour des cas plus compliqués, au lieu d'utiliser un rappel, la fonction elle-même peut renvoyer un objet EventEmitter, et l'appelant est censé écouter les événements d'erreur sur l'émetteur.

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

function doSomeAsynchronousOperation(req, res) {
    let myEvent = new EventEmitter();

    // runs asynchronously
    setTimeout(function(){
        myEvent.emit('error', new Error('User Name cannot be empty'));
    }, 1000);

    return myEvent;
}

// Invoke the function
let event = doSomeAsynchronousOperation(req, res);

event.on('error', function(err) {
    console.log(err);
});

event.on('done', function(result) {
    console.log(result); // true
});
```

Gestion des exceptions non maîtrisées

Étant donné que Node.js s'exécute sur un seul processus, les exceptions non interceptées constituent un problème à prendre en compte lors du développement d'applications.

Gestion silencieuse des exceptions

La plupart des personnes ont laissé le ou les serveurs node.js avaler les erreurs en silence.

- Manipulation silencieuse de l'exception

```
process.on('uncaughtException', function (err) {
  console.log(err);
});
```

C'est mauvais , ça marchera mais:

- La cause première restera inconnue, car elle ne contribuera pas à la résolution de ce qui a causé l’exception (erreur).
- En cas de connexion à une base de données (pool) fermée pour une raison quelconque, cela entraînera une propagation constante des erreurs, ce qui signifie que le serveur fonctionnera mais ne se reconnectera pas à la base de données.

Retourner à l'état initial

En cas d’exception "uncaughtException", il est bon de redémarrer le serveur et de le ramener à son **état initial** , où nous savons qu’il fonctionnera. L'exception est consignée, l'application est terminée, mais étant donné qu'elle s'exécutera dans un conteneur qui s'assurera que le serveur fonctionne, nous allons redémarrer le serveur (en revenant à son état de fonctionnement initial).

- Installer le pour toujours (ou un autre outil CLI pour s'assurer que le serveur de noeuds fonctionne en continu)

```
npm install forever -g
```

- Démarrer le serveur pour toujours

```
forever start app.js
```

  Raison pour laquelle est-il commencé et pourquoi nous utilisons pour toujours est après que le serveur est **arrêté** pour toujours, le processus va redémarrer le serveur.

- Redémarrer le serveur

```
process.on('uncaughtException', function (err) {
    console.log(err);

    // some logging mechanisam
```

```
    // ....

    process.exit(1); // terminates process
});
```

Sur un côté, il y avait un moyen de gérer les exceptions avec les **clusters et les domaines** .

Les domaines sont obsolètes plus d'informations ici .

Erreurs et promesses

Les promesses traitent les erreurs différemment du code synchrone ou du code de rappel.

```
const p = new Promise(function (resolve, reject) {
    reject(new Error('Oops'));
});

// anything that is `reject`ed inside a promise will be available through catch
// while a promise is rejected, `.then` will not be called
p
    .then(() => {
        console.log("won't be called");
    })
    .catch(e => {
        console.log(e.message); // output: Oops
    })
    // once the error is caught, execution flow resumes
    .then(() => {
        console.log('hello!'); // output: hello!
    });
```

Actuellement, les erreurs dans une promesse qui ne sont pas détectées entraînent une erreur, ce qui peut rendre difficile le suivi de l'erreur. Cela peut être résolu en utilisant des outils de peluches comme eslint ou en vous assurant de toujours avoir une clause de `catch` .

Ce comportement est obsolète dans le noeud 8 en faveur de la fin du processus de noeud.

Lire Gestion des exceptions en ligne: https://riptutorial.com/fr/node-js/topic/2819/gestion-des-exceptions

Chapitre 49: Gestionnaire de paquets de fils

Introduction

Yarn est un gestionnaire de paquets pour Node.js, similaire à npm. Tout en partageant beaucoup de points communs, il existe des différences essentielles entre Yarn et npm.

Examples

Installation de fil

Cet exemple explique les différentes méthodes d'installation de Yarn pour votre système d'exploitation.

macOS

Homebrew

```
brew update
brew install yarn
```

MacPorts

```
sudo port install yarn
```

Ajouter du fil à votre PATH

Ajoutez ce qui suit à votre profil de shell préféré (`.profile` , `.bashrc` , `.zshrc` etc.)

```
export PATH="$PATH:`yarn global bin`"
```

les fenêtres

Installateur

D'abord, installez Node.js s'il n'est pas déjà installé.

Téléchargez le programme d'installation de Yarn en tant que `.msi` sur le site Web de Yarn .

Chocolaté

```
choco install yarn
```

Linux

Debian / Ubuntu

Assurez-vous que Node.js est installé pour votre distribution ou exécutez la commande suivante

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Configurez le référentiel YarnPkg

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee
/etc/apt/sources.list.d/yarn.list
```

Installer le fil

```
sudo apt-get update && sudo apt-get install yarn
```

CentOS / Fedora / RHEL

Installez Node.js s'il n'est pas déjà installé

```
curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
```

Installer le fil

```
sudo wget https://dl.yarnpkg.com/rpm/yarn.repo -O /etc/yum.repos.d/yarn.repo
sudo yum install yarn
```

Cambre

Installez le fil via AUR.

Exemple utilisant yaourt:

```
yaourt -S yarn
```

Solus

```
sudo eopkg install yarn
```

Toutes les distributions

Ajoutez ce qui suit à votre profil de shell préféré (`.profile` , `.bashrc` , `.zshrc` etc.)

```
export PATH="$PATH:`yarn global bin`"
```

Méthode d'installation alternative

Script shell

```
curl -o- -L https://yarnpkg.com/install.sh | bash
```

ou spécifiez une version à installer

```
curl -o- -L https://yarnpkg.com/install.sh | bash -s -- --version [version]
```

Tarball

```
cd /opt
wget https://yarnpkg.com/latest.tar.gz
tar zvxf latest.tar.gz
```

Npm

Si vous avez déjà installé npm, lancez simplement

```
npm install -g yarn
```

Post Install

Vérifiez la version installée de Yarn en cours d'exécution

```
yarn --version
```

Créer un package de base

La commande `yarn init` vous guidera à travers la création d'un fichier `package.json` pour configurer certaines informations sur votre paquet. Ceci est similaire à la commande `npm init` dans npm.

Créez et accédez à un nouveau répertoire pour contenir votre package, puis exécutez `yarn init`

```
mkdir my-package && cd my-package
yarn init
```

Répondez aux questions qui suivent dans la CLI

```
question name (my-package): my-package
question version (1.0.0):
question description: A test package
question entry point (index.js):
question repository url:
question author: StackOverflow Documentation
question license (MIT):
success Saved package.json
  Done in 27.31s.
```

Cela générera un fichier `package.json` similaire au suivant

```
{
  "name": "my-package",
  "version": "1.0.0",
  "description": "A test package",
  "main": "index.js",
  "author": "StackOverflow Documentation",
  "license": "MIT"
}
```

Maintenant, essayons d'ajouter une dépendance. La syntaxe de base pour ceci est `yarn add [package-name]`

Exécutez les opérations suivantes pour installer ExpressJS

```
yarn add express
```

Cela ajoutera une section de `dependencies` à votre `package.json` et ajoutera ExpressJS

```
"dependencies": {
    "express": "^4.15.2"
}
```

Installer le paquet avec du fil

Yarn utilise le même registre que npm. Cela signifie que chaque paquet disponible sur npm est le même sur Yarn.

Pour installer un paquet, lancez le `yarn add package` .

Si vous avez besoin d'une version spécifique du package, vous pouvez utiliser `yarn add package@version` .

Si la version à installer a été marquée, vous pouvez utiliser `yarn add package@tag` .

Lire Gestionnaire de paquets de fils en ligne: https://riptutorial.com/fr/node-js/topic/9441/gestionnaire-de-paquets-de-fils

Chapitre 50: grognement

Remarques

Lectures complémentaires:

Le Guide d'installation de Grunt contient des informations détaillées sur l'installation de versions spécifiques, en production ou en développement, de Grunt et de grunt-cli.

Le guide de configuration des tâches contient des explications détaillées sur la configuration des tâches, des cibles, des options et des fichiers dans le Gruntfile, ainsi qu'une explication des modèles, des modèles de globalisation et de l'importation de données externes.

Le guide Création de tâches répertorie les différences entre les types de tâches Grunt et présente plusieurs exemples de tâches et de configurations.

Examples

Introduction aux GruntJs

Grunt est un gestionnaire de tâches JavaScript, utilisé pour automatiser des tâches répétitives telles que la minification, la compilation, les tests unitaires, le linting, etc.

Pour commencer, vous devez installer l'interface de ligne de commande (CLI) de Grunt globalement.

```
npm install -g grunt-cli
```

Préparation d'un nouveau projet Grunt: Une configuration typique implique l'ajout de deux fichiers à votre projet: package.json et le fichier Gruntfile.

package.json: ce fichier est utilisé par npm pour stocker les métadonnées des projets publiés en tant que modules npm. Vous allez lister les plugins Grunt et Grunt dont votre projet a besoin en tant que devDependencies dans ce fichier.

Gruntfile: Ce fichier s'appelle Gruntfile.js et permet de configurer ou de définir des tâches et de charger des plugins Grunt.

```
Example package.json:

{
  "name": "my-project-name",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "~0.4.5",
    "grunt-contrib-jshint": "~0.10.0",
    "grunt-contrib-nodeunit": "~0.4.1",
    "grunt-contrib-uglify": "~0.5.0"
```

```
  }
}
```

Exemple: gruntfile:

```
module.exports = function(grunt) {

  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    uglify: {
      options: {
        banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
      },
      build: {
        src: 'src/<%= pkg.name %>.js',
        dest: 'build/<%= pkg.name %>.min.js'
      }
    }
  });

  // Load the plugin that provides the "uglify" task.
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Default task(s).
  grunt.registerTask('default', ['uglify']);

};
```

Installation de gruntplugins

Ajout de la dépendance

Pour utiliser un gruntplugin, vous devez d'abord l'ajouter en tant que dépendance à votre projet. Utilisons le plugin jshint comme exemple.

```
npm install grunt-contrib-jshint --save-dev
```

L'option `--save-dev` est utilisée pour ajouter le plugin dans le `package.json` , ainsi le plugin est toujours installé après une `npm install` .

Charger le plugin

Vous pouvez charger votre plug-in dans le fichier gruntfile en utilisant `loadNpmTasks` .

```
grunt.loadNpmTasks('grunt-contrib-jshint');
```

Configuration de la tâche

Vous configurez la tâche dans le gruntfile en ajoutant une propriété appelée `jshint` à l'objet passé à `grunt.initConfig` .

```
grunt.initConfig({
```

```
  jshint: {
    all: ['Gruntfile.js', 'lib/**/*.js', 'test/**/*.js']
  }
});
```

N'oubliez pas que vous pouvez avoir d'autres propriétés pour d'autres plugins que vous utilisez.

Exécuter la tâche

Pour exécuter la tâche avec le plug-in, vous pouvez utiliser la ligne de commande.

```
grunt jshint
```

Ou vous pouvez ajouter `jshint` à une autre tâche.

```
grunt.registerTask('default', ['jshint']);
```

La tâche par défaut s'exécute avec la commande grunt dans le terminal sans aucune option.

Lire grognement en ligne: https://riptutorial.com/fr/node-js/topic/6059/grognement

Chapitre 51: Guide du débutant NodeJS

Examples

Bonjour le monde !

Placez le code suivant dans un nom de fichier `helloworld.js`

```
console.log("Hello World");
```

Enregistrez le fichier et exécutez-le via Node.js:

```
node helloworld.js
```

Lire Guide du débutant NodeJS en ligne: https://riptutorial.com/fr/node-js/topic/7693/guide-du-debutant-nodejs

Chapitre 52: Histoire de Nodejs

Introduction

Ici, nous allons discuter de l'historique de Node.js, des informations de version et de son état actuel.

Examples

Événements clés de chaque année

2009

- 3 mars: Le projet a été nommé "noeud"
- 1er octobre: Premier aperçu de npm, le paquet Node directeur
- 8 novembre: Ryan Dahl (Créateur de Node.js) Talk Node.js original à JSConf 2009

2010

- Express: un framework de développement web Node.js
- Socket.io version initiale
- 28 avril: support expérimental de Node.js sur Heroku
- 28 juillet: Google Tech Talk de Ryan Dahl sur Node.js
- 20 août: sortie de Node.js 0.2.0

2011

- 31 mars: Guide Node.js
- 1er mai: npm 1.0: sorti
- 1er mai: AMA de Ryan Dahl sur Reddit
- 10 juillet: Le Node Beginner Book, une introduction à Node.js, est terminé .
    - Un tutoriel complet Node.js pour les débutants.
- 16 août: LinkedIn utilise Node.js
    - LinkedIn a lancé son application mobile entièrement révisée avec de nouvelles fonctionnalités et de nouvelles pièces sous le capot.
- 5 octobre: Ryan Dahl parle de l'histoire de Node.js et pourquoi il l'a créée
- 5 décembre: Node.js en production chez Uber
    - Le directeur de Uber Engineering, Curtis Chambers, explique pourquoi son entreprise a complètement repensé son application à l’aide de Node.js pour accroître son efficacité et améliorer l’expérience des partenaires et des clients.

2012

- 30 janvier: Ryan Dahl, créateur de Node.js, s'éloigne du quotidien de Node
- 25 juin: Node.js v0.8.0 [stable] est sorti
- 20 décembre: Hapi, un framework Node.js est sorti

2013

- 30 avril: La pile MEAN: MongoDB, ExpressJS, AngularJS et Node.js
- 17 mai: Comment nous avons créé la première application Node.js d'eBay
- 15 novembre: PayPal libère Kraken, un framework Node.js
- 22 novembre: fuite de mémoire de Node.js chez Walmart
    - Eran Hammer des laboratoires Wal-Mart est venu à l'équipe principale de Node.js se plaignant d'une fuite de mémoire qu'il traquait depuis des mois.
- 19 décembre: Koa - Framework Web pour Node.js

2014

- 15 janvier: TJ Fontaine prend en charge le projet Node
- 23 octobre: Node.js Advisory Board
    - Joyent et plusieurs membres de la communauté Node.js ont annoncé une proposition pour un conseil consultatif Node.js comme prochaine étape vers un modèle de gouvernance entièrement ouvert pour le projet open source Node.js.
- 19 novembre: Node.js dans les graphiques de flammes - Netflix
- 28 novembre: IO.js - E / S événementielle pour Javascript Javascript

2015

Q1

- 14 janvier: IO.js 1.0.0
- 10 février: Joyent s'installe pour créer la fondation Node.js
    - Joyent, IBM, Microsoft, PayPal, Fidelity, SAP et la Linux Foundation s'unissent pour soutenir la communauté Node.js avec une gouvernance neutre et ouverte
- 27 février : proposition de rapprochement IO.js et Node.js

Q2

- 14 avril: npm Modules privés
- 28 mai: TJ Fontaine, chef de groupe, quitte son poste et quitte Joyent
- 13 mai: Node.js et io.js fusionnent sous la fondation Node

Q3

- 2 août: Trace - Surveillance des performances et débogage de Node.js
    - Trace est un outil de surveillance de microservices visualisé qui vous fournit toutes les métriques dont vous avez besoin lors de l'utilisation de microservices.
- 13 août: 4.0 est le nouveau 1.0

Q4

- 12 octobre: Node v4.2.0, première version du support à long terme
- 8 décembre: Apigee, RisingStack et Yahoo rejoignent la fondation Node.js
- 8 et 9 décembre: Node Interactive
    - La première conférence annuelle Node.js de la Fondation Node.js

2016

Q1

- 10 février: Express devient un projet incubé
- 23 mars: l' incident de gauche
- 29 mars: Google Cloud Platform rejoint la fondation Node.js

Q2

- 26 avril: npm a 210.000 utilisateurs

Q3

- 18 juillet: CJ Silverio devient CTO de npm
- 1er août: Trace, la solution de débogage Node.js est généralement disponible
- 15 septembre: Le premier nœud interactif en Europe

Q4

- 11 octobre: le gestionnaire de paquets de fils est sorti
- 18 octobre: Node.js 6 devient la version LTS

Référence

1. "Historique de Node.js sur une timeline" [en ligne]. Disponible: [https://blog.risingstack.com/history-of-node-js]

Lire Histoire de Nodejs en ligne: https://riptutorial.com/fr/node-js/topic/8653/histoire-de-nodejs

Chapitre 53: http

Examples

serveur http

Un exemple basique de serveur HTTP.

écrire le code suivant dans le fichier http_server.js:

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

var httpPort = 80;

http.createServer(handler).listen(httpPort, start_callback);

function handler(req, res) {

    var clientIP = req.connection.remoteAddress;
    var connectUsing = req.connection.encrypted ? 'SSL' : 'HTTP';
    console.log('Request received: '+ connectUsing + ' ' + req.method + ' ' + req.url);
    console.log('Client IP: ' + clientIP);

    res.writeHead(200, "OK", {'Content-Type': 'text/plain'});
    res.write("OK");
    res.end();
    return;
}

function start_callback(){
    console.log('Start HTTP on port ' + httpPort)
}
```

puis à partir de votre emplacement http_server.js, exécutez cette commande:

```
node http_server.js
```

vous devriez voir ce résultat:

```
> Start HTTP on port 80
```

Maintenant, vous devez tester votre serveur, vous devez ouvrir votre navigateur Internet et accéder à cette URL:

```
http://127.0.0.1:80
```

Si votre machine utilise un serveur Linux, vous pouvez le tester comme ceci:

```
curl 127.0.0.1:80
```

vous devriez voir le résultat suivant:

```
ok
```

dans votre console, celle qui exécute l'application, vous verrez ce résultat:

```
> Request received: HTTP GET /
> Client IP: ::ffff:127.0.0.1
```

client http

un exemple de base pour le client http:

écrivez le code suivant dans le fichier http_client.js:

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

var options = {
  hostname: '127.0.0.1',
  port: 80,
  path: '/',
  method: 'GET'
};

var req = http.request(options, function(res) {
    console.log('STATUS: ' + res.statusCode);
    console.log('HEADERS: ' + JSON.stringify(res.headers));
    res.setEncoding('utf8');
    res.on('data', function (chunk) {
        console.log('Response: ' + chunk);
    });
    res.on('end', function (chunk) {
        console.log('Response ENDED');
    });
});

req.on('error', function(e) {
    console.log('problem with request: ' + e.message);
});


req.end();
```

puis à partir de votre emplacement http_client.js, exécutez cette commande:

```
node http_client.js
```

vous devriez voir ce résultat:

```
> STATUS: 200
> HEADERS: {"content-type":"text/plain","date":"Thu, 21 Jul 2016 11:27:17
GMT","connection":"close","transfer-encoding":"chunked"}
> Response: OK
> Response ENDED
```

note: cet exemple dépend de l'exemple du serveur http.

Lire http en ligne: https://riptutorial.com/fr/node-js/topic/2973/http

Chapitre 54: Injection de dépendance

Examples

Pourquoi utiliser l'injection de dépendance

1. **Processus de développement rapide**
2. **Découplage**
3. **Écriture de test unitaire**

Processus de développement rapide
Lorsque vous utilisez un nœud d'injection de dépendance, le développeur peut accélérer son processus de développement car, après l'exécution des DI, il y a moins de conflits de code et il est facile de gérer tous les modules.

Découplage
Les modules deviennent moins couplés que faciles à entretenir.

Écriture de test unitaire
Les dépendances codées en dur peuvent être transférées dans le module puis faciles à écrire pour chaque module.

Lire Injection de dépendance en ligne: https://riptutorial.com/fr/node-js/topic/7681/injection-de-dependance

Chapitre 55: Installer Node.js

Examples

Installer Node.js sur Ubuntu

Utiliser le gestionnaire de paquets apt

```
sudo apt-get update
sudo apt-get install nodejs
sudo apt-get install npm
sudo ln -s /usr/bin/nodejs /usr/bin/node

# the node & npm versions in apt are outdated. This is how you can update them:
sudo npm install -g npm
sudo npm install -g n
sudo n stable # (or lts, or a specific version)
```

Utiliser la dernière version spécifique (par exemple, LTS 6.x) directement à partir de nodesource

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
apt-get install -y nodejs
```

En outre, pour installer correctement les modules npm globaux, définissez le répertoire personnel (élimine le besoin de sudo et évite les erreurs EACCES):

```
mkdir ~/.npm-global
echo "export PATH=~/.npm-global/bin:$PATH" >> ~/.profile
source ~/.profile
npm config set prefix '~/.npm-global'
```

Installer Node.js sur Windows

Installation standard

Tous les fichiers binaires, programmes d'installation et fichiers source Node.js peuvent être téléchargés ici .

Vous pouvez uniquement télécharger le runtime `node.exe` ou utiliser le programme d'installation de Windows (`.msi`), qui installera également `npm` , le gestionnaire de packages recommandé pour Node.js, et configurera les chemins.

Installation par gestionnaire de paquets

Vous pouvez également installer par le gestionnaire de paquets *Chocolatey* (Software Management Automation).

```
# choco install nodejs.install
```

Plus d'informations sur la version actuelle, vous pouvez trouver dans le dépôt de choco ici .

Utiliser Node Version Manager (nvm)

Node Version Manager , également appelé nvm, est un script bash qui simplifie la gestion de plusieurs versions de Node.js.

Pour installer NVM, utilisez le script d'installation fourni:

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Pour Windows, il existe un package nvm-windows avec un installateur. Cette page GithHub contient les détails pour installer et utiliser le package nvm-windows.

Après avoir installé nvm, exécutez "nvm on" à partir de la ligne de commande. Cela permet à NVM de contrôler les versions de nœud.

Remarque: Vous devrez peut-être redémarrer votre terminal pour qu'il reconnaisse la commande `nvm` nouvellement installée.

Ensuite, installez la dernière version du noeud:

```
$ nvm install node
```

Vous pouvez également installer une version de noeud spécifique en transmettant les versions majeures, mineures et / ou correctives:

```
$ nvm install 6
$ nvm install 4.2
```

Pour répertorier les versions disponibles pour l'installation:

```
$ nvm ls-remote
```

Vous pouvez ensuite changer de version en passant la version de la même manière que lors de l'installation:

```
$ nvm use 5
```

Vous pouvez définir une version spécifique de Node que vous avez installée comme étant la **version** par **défaut** en entrant:

```
$ nvm alias default 4.2
```

Pour afficher une liste des versions de nœud installées sur votre ordinateur, entrez:

```
$ nvm ls
```

Pour utiliser des versions de noeud spécifiques au projet, vous pouvez enregistrer la version dans le fichier .nvmrc. De cette façon, commencer à travailler avec un autre projet sera moins sujet aux erreurs après l'avoir extrait de son référentiel.

```
$ echo "4.2" > .nvmrc
$ nvm use
Found '/path/to/project/.nvmrc' with version <4.2>
Now using node v4.2 (npm v3.7.3)
```

Lorsque Node est installé via nvm, il n'est pas nécessaire d'utiliser `sudo` pour installer les packages globaux car ils sont installés dans le dossier de base. Ainsi, `npm i -g http-server` fonctionne sans erreur de permission.

Installer Node.js From Source avec le gestionnaire de paquets APT

Conditions préalables

```
sudo apt-get install build-essential
sudo apt-get install python

[optional]
sudo apt-get install git
```

Obtenir la source et construire

```
cd ~
git clone https://github.com/nodejs/node.git
```

OU Pour la dernière version de LTS Node.js 6.10.2

```
cd ~
wget https://nodejs.org/dist/v6.3.0/node-v6.10.2.tar.gz
tar -xzvf node-v6.10.2.tar.gz
```

Accédez au répertoire source tel que `cd ~/node-v6.10.2`

```
./configure
make
sudo make install
```

Installer Node.js sur Mac en utilisant le gestionnaire de paquets

Homebrew

Vous pouvez installer Node.js en utilisant le gestionnaire de paquets Homebrew .

Commencez par mettre à jour l'infusion:

```
brew update
```

Vous devrez peut-être modifier les autorisations ou les chemins. Il est préférable de l'exécuter avant de continuer:

```
brew doctor
```

Ensuite, vous pouvez installer Node.js en exécutant:

```
brew install node
```

Une fois Node.js installé, vous pouvez valider la version installée en exécutant:

```
node -v
```

Macports

Vous pouvez également installer node.js via Macports .

Commencez par le mettre à jour pour vous assurer que les derniers packages sont référencés:

```
sudo port selfupdate
```

Ensuite, installez nodejs et npm

```
sudo port install nodejs npm
```

Vous pouvez maintenant exécuter le noeud directement via l'interface de ligne de commande en appelant le `node` . En outre, vous pouvez vérifier votre version de noeud actuelle avec

```
node -v
```

Installation à l'aide de MacOS X Installer

Vous pouvez trouver les installateurs sur la page de téléchargement de Node.js. Node.js recommande normalement deux versions de Node, la version LTS (support à long terme) et la version actuelle (dernière version). Si vous êtes nouveau sur le nœud, allez simplement sur le LTS, puis cliquez sur le bouton `Macintosh Installer` pour télécharger le package.

Si vous souhaitez trouver d'autres versions de NodeJS, cliquez ici , choisissez votre version, puis cliquez sur Télécharger. À partir de la page de téléchargement, recherchez un fichier avec l'extension `.pkg` .

Une fois que vous avez téléchargé le fichier (avec l'extension `.pkg` ofcourse), double-cliquez dessus pour l'installer. Le programme d'installation contenant `Node.js` et `npm` , par défaut, le package sera installé les deux, mais vous pouvez personnaliser celui à installer en cliquant sur le bouton `customize` dans l'étape `Installation Type` . Sinon, suivez simplement les instructions d'installation, c'est assez simple.

Vérifiez si le nœud est installé

`terminal` ouvert (si vous ne savez pas comment ouvrir votre terminal, regardez ce wikihow). Dans le terminal, tapez `node --version` puis entrez. Votre terminal ressemblera à ceci si Node est installé:

```
$ node --version
v7.2.1
```

La `v7.2.1` est votre version de Node.js, si vous recevez la `command not found: node` message `command not found: node` au lieu de cela, alors cela signifie qu'il y a un problème avec votre installation.

Installation de Node.js sur Raspberry PI

Pour installer v6.x, mettez à jour les packages

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
```

Utiliser le gestionnaire de paquets apt

```
sudo apt-get install -y nodejs
```

Installation avec Node Version Manager sous Fish Shell avec Oh My Fish!

Node Version Manager (`nvm`) simplifie grandement la gestion des versions de Node.js, leur installation et supprime la nécessité de `sudo` pour gérer les packages (par exemple, `npm install ...`). Fish Shell (`fish`) " *est un shell de ligne de commande intelligent et convivial pour OS X, Linux et le reste de la famille* ", une alternative populaire parmi les programmeurs pour les shells courants tels que `bash` . Enfin, Oh My Fish (`omf`) permet de personnaliser et d'installer des paquets dans Fish Shell.

Ce guide suppose que vous utilisez déjà Fish comme shell .

Installez nvm

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.4/install.sh | bash
```

Installer Oh mon poisson

```
curl -L https://github.com/oh-my-fish/oh-my-fish/raw/master/bin/install | fish
```

(Remarque: vous serez invité à redémarrer votre terminal à ce stade. Allez-y et faites-le maintenant.)

Installer le plugin-nvm pour Oh My Fish

Nous allons installer plugin-nvm via Oh My Fish pour exposer les fonctionnalités de `nvm` dans le shell Fish:

```
omf install nvm
```

Installer Node.js avec Node Version Manager

Vous êtes maintenant prêt à utiliser `nvm` . Vous pouvez installer et utiliser la version de Node.js à votre goût. Quelques exemples:

- Installez la version de nœud la plus récente: `nvm install node`
- Installez spécifiquement 6.3.1: `nvm install 6.3.1`
- Liste des vérifications installées: `nvm ls`
- Passer à un 4.3.1 précédemment installé: `nvm use 4.3.1`

Notes finales

Rappelez-vous encore une fois que nous n'avons plus besoin de `sudo` avec Node.js en utilisant cette méthode! Les versions de noeud, les packages, etc. sont installés dans votre répertoire personnel.

Installez Node.js depuis la source sur Centos, RHEL et Fedora

Conditions préalables

- git
- `clang` et `clang++` 3.4 ^ ou `gcc` et `g++` 4.8 ^
- Python 2.6 ou 2.7
- GNU Make 3.81 ^

Obtenir la source

Node.js v6.x LTS

```
git clone -b v6.x https://github.com/nodejs/node.git
```

Node.js v7.x

```
git clone -b v7.x https://github.com/nodejs/node.git
```

Construire

```
cd node
./configure
make -jX
su -c make install
```

X - le nombre de cœurs de processeur, accélère considérablement la construction

Nettoyage [facultatif]

```
cd
rm -rf node
```

Installer Node.js avec n

Tout d'abord, il existe un très bon wrapper pour configurer `n` sur votre système. Il suffit de courir:

```
curl -L https://git.io/n-install | bash
```

installer `n` . Installez ensuite les fichiers binaires de différentes manières:

dernier

```
n latest
```

stable

```
n stable
```

c'est

```
n lts
```

Toute autre version

```
n <version>
```

par exemple `n 4.4.7`

Si cette version est déjà installée, cette commande activera cette version.

Changer de version

`n` soi, cela produira une liste de sélection des binaires installés. Utilisez le haut et le bas pour trouver celui que vous voulez et appuyez sur Entrée pour l'activer.

Lire Installer Node.js en ligne: https://riptutorial.com/fr/node-js/topic/1294/installer-node-js

Chapitre 56: Intégration de Cassandra

Examples

Bonjour le monde

Pour accéder à Cassandra, vous pouvez utiliser le module Cassandra `cassandra-driver` de DataStax. Il prend en charge toutes les fonctionnalités et peut être facilement configuré.

```
const cassandra = require("cassandra-driver");
const clientOptions = {
    contactPoints: ["host1", "host2"],
    keyspace: "test"
};

const client = new cassandra.Client(clientOptions);

const query = "SELECT hello FROM world WHERE name = ?";
client.execute(query, ["John"], (err, results) => {
    if (err) {
      return console.error(err);
    }

    console.log(results.rows);
});
```

Lire Intégration de Cassandra en ligne: https://riptutorial.com/fr/node-js/topic/5949/integration-de-cassandra

Chapitre 57: Intégration des passeports

Remarques

Le mot de passe doit **toujours** être haché. Un moyen simple de sécuriser les mots de passe avec **NodeJS** serait d'utiliser le module **bcrypt-nodejs** .

Examples

Commencer

Le passeport doit être initialisé à l'aide du middleware `passport.initialize()` . Pour utiliser les sessions de connexion, le middleware `passport.session()` est requis.

Notez que `passport.serialize()` méthodes `passport.serialize()` et `passport.deserializeUser()` doivent être définies. **Passport** va sérialiser et désérialiser les instances d'utilisateurs à partir de la session

```
const express = require('express');
const session = require('express-session');
const passport = require('passport');
const cookieParser = require('cookie-parser');
const app = express();

// Required to read cookies
app.use(cookieParser());

passport.serializeUser(function(user, next) {
    // Serialize the user in the session
    next(null, user);
});

passport.deserializeUser(function(user, next) {
    // Use the previously serialized user
        next(null, user);
});

// Configuring express-session middleware
app.use(session({
    secret: 'The cake is a lie',
    resave: true,
    saveUninitialized: true
}));

// Initializing passport
app.use(passport.initialize());
app.use(passport.session());

// Starting express server on port 3000
app.listen(3000);
```

Authentification locale

Le module **passport-local** est utilisé pour implémenter une authentification locale.

> Ce module vous permet de vous authentifier à l'aide d'un nom d'utilisateur et d'un mot de passe dans vos applications Node.js.

Enregistrer l'utilisateur:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

// A named strategy is used since two local strategy are used :
// one for the registration and the other to sign-in
passport.use('localSignup', new LocalStrategy({
    // Overriding defaults expected parameters,
    // which are 'username' and 'password'
    usernameField: 'email',
    passwordField: 'password',
    passReqToCallback: true // allows us to pass back the entire request to the callback    .
},
function(req, email, password, next) {
    // Check in database if user is already registered
    findUserByEmail(email, function(user) {
        // If email already exists, abort registration process and
        // pass 'false' to the callback
        if (user) return next(null, false);
        // Else, we create the user
        else {
            // Password must be hashed !
            let newUser = createUser(email, password);

            newUser.save(function() {
                // Pass the user to the callback
                return next(null, newUser);
            });
        }
    });
});
```

Connexion à l'utilisateur:

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;

passport.use('localSignin', new LocalStrategy({
    usernameField : 'email',
    passwordField : 'password',
},
function(email, password, next) {
    // Find the user
    findUserByEmail(email, function(user) {
        // If user is not found, abort signing in process
        // Custom messages can be provided in the verify callback
        // to give the user more details concerning the failed authentication
        if (!user)
            return next(null, false, {message: 'This e-mail address is not associated with any
account.'});
        // Else, we check if password is valid
        else {
```

```
            // If password is not correct, abort signing in process
            if (!isPasswordValid(password)) return next(null, false);
            // Else, pass the user to callback
            else return next(null, user);
        }
    });
});
```

Création d'itinéraires:

```
// ...
app.use(passport.initialize());
app.use(passport.session());

// Sign-in route
// Passport strategies are middlewares
app.post('/login', passport.authenticate('localSignin', {
    successRedirect: '/me',
    failureRedirect: '/login'
});

// Sign-up route
app.post('/register', passport.authenticate('localSignup', {
    successRedirect: '/',
    failureRedirect: '/signup'
});

// Call req.logout() to log out
app.get('/logout', function(req, res) {
    req.logout();
    res.redirect('/');
});

app.listen(3000);
```

Authentification Facebook

Le module **passeport-facebook** est utilisé pour mettre en œuvre une authentification **Facebook** . Dans cet exemple, si l'utilisateur n'existe pas lors de la connexion, il est créé.

Stratégie de mise en œuvre:

```
const passport = require('passport');
const FacebookStrategy = require('passport-facebook').Strategy;

// Strategy is named 'facebook' by default
passport.use({
    clientID: 'yourclientid',
    clientSecret: 'yourclientsecret',
    callbackURL: '/auth/facebook/callback'
},
// Facebook will send a token and user's profile
function(token, refreshToken, profile, next) {
    // Check in database if user is already registered
    findUserByFacebookId(profile.id, function(user) {
        // If user exists, returns his data to callback
        if (user) return next(null, user);
```

```
        // Else, we create the user
        else {
            let newUser = createUserFromFacebook(profile, token);

            newUser.save(function() {
                // Pass the user to the callback
                return next(null, newUser);
            });
        }
    });
});
```

Création d'itinéraires:

```
// ...
app.use(passport.initialize());
app.use(passport.session());

// Authentication route
app.get('/auth/facebook', passport.authenticate('facebook', {
    // Ask Facebook for more permissions
    scope : 'email'
}));

// Called after Facebook has authenticated the user
app.get('/auth/facebook/callback',
    passport.authenticate('facebook', {
        successRedirect : '/me',
        failureRedirect : '/'
}));


//...

app.listen(3000);
```

Authentification par nom d'utilisateur-mot de passe simple

Dans vos itinéraires / index.js

Ici, l' `user` est le modèle de userSchema

```
router.post('/login', function(req, res, next) {
    if (!req.body.username || !req.body.password) {
        return res.status(400).json({
            message: 'Please fill out all fields'
        });
    }

    passport.authenticate('local', function(err, user, info) {
        if (err) {
            console.log("ERROR : " + err);
            return next(err);
        }

        if(user) (
```

```
            console.log("User Exists!")
            //All the data of the user can be accessed by user.x
            res.json({"success" : true});
            return;
        } else {
            res.json({"success" : false});
            console.log("Error" + errorResponse());
            return;
        }
    })(req, res, next);
});
```

Authentification Google Passport

Nous avons un module simple disponible dans npm pour le nom d'authentification de masque. Passport **-google-oauth20**

Considérez l'exemple suivant Dans cet exemple, créez un dossier à savoir config ayant le fichier passport.js et google.js dans le répertoire racine. Dans votre app.js inclure les éléments suivants

```
var express = require('express');
var session = require('express-session');
var passport = require('./config/passport'); // path where the passport file placed
var app = express();
passport(app);
```

// autre code pour initialiser le serveur, handle d'erreur

Dans le fichier passport.js du dossier config, incluez le code suivant

```
var passport = require  ('passport'),
google = require('./google'),
User = require('./../model/user'); // User is the  mongoose model

module.exports = function(app){
  app.use(passport.initialize());
  app.use(passport.session());
  passport.serializeUser(function(user, done){
      done(null, user);
  });
  passport.deserializeUser(function (user, done) {
     done(null, user);
  });
  google();
};
```

Dans le fichier google.js dans le même dossier de configuration, incluez

```
var passport = require('passport'),
GoogleStrategy = require('passport-google-oauth20').Strategy,
User = require('./../model/user');
module.exports = function () {
passport.use(new GoogleStrategy({
        clientID: 'CLIENT ID',
        clientSecret: 'CLIENT SECRET',
```

```
        callbackURL: "http://localhost:3000/auth/google/callback"
    },
    function(accessToken, refreshToken, profile, cb) {
        User.findOne({ googleId : profile.id }, function (err, user) {
            if(err){
                return cb(err, false, {message : err});
            }else {
                if (user != '' && user != null) {
                    return cb(null, user, {message : "User "});
                } else {
                    var username  = profile.displayName.split(' ');
                    var userData = new User({
                        name : profile.displayName,
                        username : username[0],
                        password : username[0],
                        facebookId : '',
                        googleId : profile.id,
                    });
                    // send email to user just in case required to send the newly created
                    // credentails to user for future login without using google login
                    userData.save(function (err, newuser) {
                        if (err) {
                            return cb(null, false, {message : err + " !!! Please try again"});
                        }else{
                            return cb(null, newuser);
                        }
                    });
                }
            }
        });
      }
  ));
};
```

Dans cet exemple, si l'utilisateur n'est pas dans DB, créez un nouvel utilisateur dans la base de données pour référence locale à l'aide du nom de champ googleId dans le modèle utilisateur.

Lire Intégration des passeports en ligne: https://riptutorial.com/fr/node-js/topic/7666/integration-des-passeports

Chapitre 58: Intégration Mongodb

Syntaxe

- db. *collection* .insertOne (*document* , *options (w, wtimeout, j, serializeFuntions, forceServerObjectId, bypassDocumentValidation)* , *rappel*)
- db. *collection* .insertMany (*[documents]* , *options (w, wtimeout, j, serializeFuntions, forceServerObjectId, bypassDocumentValidation)* , *rappel*)
- db. *collection* .find (*query*)
- db. *collection* .updateOne (*filtre* , *mise à jour* , *options (upsert, w, wtimeout, j, bypassDocumentValidation)* , *callback*)
- db. *collection* .updateMany (*filtre* , *mise à jour* , *options (upsert, w, wtimeout, j)* , *callback*)
- db. *collection* .deleteOne (*filtre* , *options (upsert, w, wtimeout, j)* , *callback*)
- db. *collection* .deleteMany (*filtre* , *options (upsert, w, wtimeout, j)* , *callback*)

Paramètres

Paramètre	Détails
document	Un objet javascript représentant un document
documents	Un tableau de documents
question	Un objet définissant une requête de recherche
filtre	Un objet définissant une requête de recherche
rappeler	Fonction à appeler lorsque l'opération est terminée
options	*(facultatif)* Paramètres facultatifs *(par défaut: null)*
w	*(facultatif)* Le souci d'écriture
wtimeout	*(facultatif)* Le délai d'expiration de l'écriture. *(par défaut: null)*
j	*(facultatif)* Spécifiez un problème d'écriture de journal *(par défaut: false)*
renversé	*(facultatif)* Opération de mise à jour *(par défaut: false)*
multi	*(facultatif)* Mettre à jour un / tous les documents *(par défaut: false)*
serializeFunctions	*(facultatif)* Sérialiser les fonctions sur n'importe quel objet *(par défaut: false)*
forceServerObjectId	*(facultatif)* Force le serveur à affecter des valeurs _id au lieu du

Paramètre	Détails
	pilote *(par défaut: false)*
bypassDocumentValidation	*(facultatif)* Autoriser le pilote à contourner la validation du schéma dans MongoDB 3.2 ou version ultérieure *(par défaut: false)*

Examples

Connectez-vous à MongoDB

Connectez-vous à MongoDB, imprimez "Connecté!" et fermez la connexion.

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

var url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function(err, db) { // MongoClient method 'connect'
    if (err) throw new Error(err);
    console.log("Connected!");
    db.close(); // Don't forget to close the connection when you are done
});
```

Méthode MongoClient `Connect()`

MongoClient.connect (*url* , *options* , *callback*)

Argument	Type	La description
`url`	chaîne	Une chaîne spécifiant l'adresse IP / nom d'hôte du serveur, le port et la base de données
`options`	objet	*(facultatif)* Paramètres facultatifs *(par défaut: null)*
`callback`	Fonction	Fonction à appeler lorsque la tentative de connexion est terminée

La fonction de `callback` prend deux arguments

- `err` : Erreur - Si une erreur survient, l'argument `err` sera défini
- `db` : object - L'instance MongoDB

Insérer un document

Insérer un document appelé «myFirstDocument» et définir **2** propriétés, `greetings` et `farewell`

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

```
const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  db.collection('myCollection').insertOne({ // Insert method 'insertOne'
    "myFirstDocument": {
      "greetings": "Hellu",
      "farewell": "Bye"
    }
  }, function (err, result) {
    if (err) throw new Error(err);
    console.log("Inserted a document into the myCollection collection!");
    db.close(); // Don't forget to close the connection when you are done
  });
});
```

Méthode de collecte `insertOne()`

db.collection (*collection*) .insertOne (*document* , *options* , *callback*)

Argument	Type	La description
`collection`	chaîne	Une chaîne spécifiant la collection
`document`	objet	Le document à insérer dans la collection
`options`	objet	*(facultatif)* Paramètres facultatifs *(par défaut: null)*
`callback`	Fonction	Fonction à appeler lorsque l'opération d'insertion est terminée

La fonction de `callback` prend deux arguments

- `err` : Erreur - Si une erreur survient, l'argument `err` sera défini
- `result` : object - Objet contenant des détails sur l'opération d'insertion

Lire une collection

Obtenez tous les documents de la collection 'myCollection' et imprimez-les sur la console.

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

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
  if (err) throw new Error(err);
  var cursor = db.collection('myCollection').find(); // Read method 'find'
  cursor.each(function (err, doc) {
    if (err) throw new Error(err);
    if (doc != null) {
      console.log(doc); // Print all documents
    } else {
      db.close(); // Don't forget to close the connection when you are done
    }
```

```
  });
});
```

Méthode de collecte `find()`

db.collection (*collection*) .find ()

Argument	Type	La description
`collection`	chaîne	Une chaîne spécifiant la collection

Mettre à jour un document

Trouvez un document avec la propriété `{ greetings: 'Hellu' }` et changez-le en `{ greetings: 'Whut?' }`

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

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
    db.collection('myCollection').updateOne({ // Update method 'updateOne'
        greetings: "Hellu" },
        { $set: { greetings: "Whut?" }},
        function (err, result) {
            if (err) throw new Error(err);
            db.close(); // Don't forget to close the connection when you are done
        });
});
```

Méthode de collecte `updateOne()`

db.collection (*collection*) .updateOne (*filtre* , *mise à jour* , *options* . *callback*)

Paramètre	Type	La description
`filter`	objet	Spécifie les critères de sélection
`update`	objet	Spécifie les modifications à appliquer
`options`	objet	*(facultatif)* Paramètres facultatifs *(par défaut: null)*
`callback`	Fonction	Fonction à appeler lorsque l'opération est terminée

La fonction de `callback` prend deux arguments

- `err` : Erreur - Si une erreur survient, l'argument err sera défini
- `db` : object - L'instance MongoDB

Supprimer un document

Supprimer un document avec la propriété `{ greetings: 'Whut?' }`

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

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
    db.collection('myCollection').deleteOne(// Delete method 'deleteOne'
        { greetings: "Whut?" },
        function (err, result) {
            if (err) throw new Error(err);
            db.close(); // Don't forget to close the connection when you are done
    });
});
```

Méthode de collecte `deleteOne()`

db.collection (*collection*) .deleteOne (*filtre* , *options* , *rappel*)

Paramètre	Type	La description
`filter`	objet	Un document précisant les critères de sélection
`options`	objet	*(facultatif)* Paramètres facultatifs *(par défaut: null)*
`callback`	Fonction	Fonction à appeler lorsque l'opération est terminée

La fonction de `callback` prend deux arguments

- `err` : Erreur - Si une erreur survient, l'argument err sera défini
- `db` : object - L'instance MongoDB

Supprimer plusieurs documents

Supprimez TOUS les documents dont la propriété «adieu» est définie sur «OK».

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

const url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function (err, db) {
    if (err) throw new Error(err);
    db.collection('myCollection').deleteMany(// MongoDB delete method 'deleteMany'
        { farewell: "okay" }, // Delete ALL documents with the property 'farewell: okay'
        function (err, result) {
            if (err) throw new Error(err);
            db.close(); // Don't forget to close the connection when you are done
    });
});
```

Méthode de collecte `deleteMany()`

db.collection (*collection*) .deleteMany (*filtre* , *options* , *rappel*)

Paramètre	Type	La description
`filter`	document	Un document précisant les critères de sélection
`options`	objet	*(facultatif)* Paramètres facultatifs *(par défaut: null)*
`callback`	fonction	Fonction à appeler lorsque l'opération est terminée

La fonction de `callback` prend deux arguments

- `err` : Erreur - Si une erreur survient, l'argument err sera défini
- `db` : object - L'instance MongoDB

Connexion simple

```
MongoDB.connect('mongodb://localhost:27017/databaseName', function(error, database) {
   if(error) return console.log(error);
   const collection = database.collection('collectionName');
   collection.insert({key: 'value'}, function(error, result) {
      console.log(error, result);
   });
});
```

Connexion simple, en utilisant des promesses

```
const MongoDB = require('mongodb');

MongoDB.connect('mongodb://localhost:27017/databaseName')
    .then(function(database) {
        const collection = database.collection('collectionName');
        return collection.insert({key: 'value'});
    })
    .then(function(result) {
        console.log(result);
    });
    ```
```

Lire Intégration Mongodb en ligne: https://riptutorial.com/fr/node-js/topic/5002/integration-mongodb
```

Chapitre 59: Intégration MongoDB pour Node.js / Express.js

Introduction

MongoDB est l'une des bases de données NoSQL les plus populaires, grâce à la pile MEAN. L'interfaçage avec une base de données Mongo à partir d'une application Express est rapide et facile, une fois que vous avez compris la syntaxe de requête kinda-wonky. Nous allons utiliser Mongoose pour nous aider.

Remarques

Plus d'informations peuvent être trouvées ici: http://mongoosejs.com/docs/guide.html

Examples

Installation de MongoDB

```
npm install --save mongodb
npm install --save mongoose //A simple wrapper for ease of development
```

Dans votre fichier serveur (normalement nommé index.js ou server.js)

```
const express = require('express');
const mongodb = require('mongodb');
const mongoose = require('mongoose');
const mongoConnectString = 'http://localhost/database name';

mongoose.connect(mongoConnectString, (err) => {
  if (err) {
    console.log('Could not connect to the database');
  }
});
```

Créer un modèle Mongoose

```
const Schema = mongoose.Schema;
const ObjectId = Schema.Types.ObjectId;

const Article = new Schema({
  title: {
    type: String,
    unique: true,
    required: [true, 'Article must have title']
  },
  author: {
    type: ObjectId,
```

```
    ref: 'User'
  }
});

module.exports = mongoose.model('Article, Article);
```

Disséquons ceci. MongoDB et Mongoose utilisent JSON (en fait BSON, mais ce n'est pas pertinent ici) comme format de données. En haut, j'ai défini quelques variables pour réduire la saisie.

Je crée un `new Schema` et l'assigne à une constante. C'est simple JSON, et chaque attribut est un autre objet avec des propriétés qui aident à appliquer un schéma plus cohérent. Unique oblige les nouvelles instances insérées dans la base de données à être évidemment uniques. Cela est idéal pour empêcher un utilisateur de créer plusieurs comptes sur un service.

Required est un autre, déclaré comme un tableau. Le premier élément est la valeur booléenne et le second le message d'erreur si la valeur insérée ou mise à jour n'existe pas.

Les ObjectIds sont utilisés pour les relations entre les modèles. Des exemples pourraient être «Les utilisateurs ont beaucoup de commentaires». D'autres attributs peuvent être utilisés à la place d'ObjectId. Les chaînes comme un nom d'utilisateur sont un exemple.

Enfin, l'exportation du modèle pour l'utiliser avec vos routes API vous permet d'accéder à votre schéma.

Interroger votre base de données Mongo

Une simple requête GET. Supposons que le modèle de l'exemple ci-dessus se trouve dans le fichier `./db/models/Article.js` .

```
const express = require('express');
const Articles = require('./db/models/Article');

module.exports = function (app) {
  const routes = express.Router();

  routes.get('/articles', (req, res) => {
    Articles.find().limit(5).lean().exec((err, doc) => {
      if (doc.length > 0) {
        res.send({ data: doc });
      } else {
        res.send({ success: false, message: 'No documents retrieved' });
      }
    });
  });

app.use('/api', routes);
};
```

Nous pouvons maintenant obtenir les données de notre base de données en envoyant une requête HTTP à ce noeud final. Quelques points clés cependant:

1. La limite fait exactement ce à quoi elle ressemble. Je ne récupère que 5 documents.

2. Lean retire certains éléments du BSON brut, réduisant ainsi la complexité et les frais généraux. Non requis. Mais utile
3. Lorsque vous utilisez `find` au lieu de `findOne` , confirmez que `doc.length` est supérieur à 0. En effet, `find` retourne toujours un tableau, donc un tableau vide ne gérera pas votre erreur, sauf si sa longueur est vérifiée.
4. Personnellement, j'aime envoyer le message d'erreur dans ce format. Changez-le selon vos besoins. Même chose pour le document retourné.
5. Le code dans cet exemple est écrit sous l'hypothèse que vous l'avez placé dans un autre fichier et non directement sur le serveur express. Pour appeler cela sur le serveur, incluez ces lignes dans le code de votre serveur:

```
const app = express();
require('./path/to/this/file')(app) //
```

Lire Intégration MongoDB pour Node.js / Express.js en ligne: https://riptutorial.com/fr/node-js/topic/9020/integration-mongodb-pour-node-js---express-js

Chapitre 60: Intégration MSSQL

Introduction

Pour intégrer une base de données avec nodejs, vous avez besoin d'un package de pilotes ou vous pouvez l'appeler un module npm qui vous fournira une API de base pour vous connecter à la base de données et effectuer des interactions. La même chose est vraie avec la base de données mssql, ici nous allons intégrer mssql avec nodejs et effectuer quelques requêtes de base sur les tabels SQL.

Remarques

Nous avons supposé que nous aurions une instance locale de serveur de base de données mssql exécutée sur une machine locale. Vous pouvez vous référer à ce document pour faire la même chose.

Assurez-vous également que l'utilisateur approprié créé avec des privilèges a également ajouté.

Examples

Connexion avec SQL via. module mssql npm

Nous commencerons par créer une application de noeud simple avec une structure de base, puis nous connecterons à la base de données du serveur sql local et effectuerons des requêtes sur cette base de données.

Etape 1: Créez un répertoire / dossier par le nom du projet que vous souhaitez créer. Initialisez une application de noeud en utilisant la commande *npm init* qui créera un package.json dans le répertoire en cours.

```
mkdir mySqlApp
//folder created
cd mwSqlApp
//change to newly created directory
npm init
//answer all the question ..
npm install
//This will complete quickly since we have not added any packages to our app.
```

Etape 2: Nous allons maintenant créer un fichier App.js dans ce répertoire et installer des paquets dont nous aurons besoin pour nous connecter à sql db.

```
sudo gedit App.js
//This will create App.js file , you can use your fav. text editor :)
npm install --save mssql
//This will install the mssql package to you app
```

Étape 3: Nous allons maintenant ajouter une variable de configuration de base à notre application qui sera utilisée par le module mssql pour établir une connexion.

```
console.log("Hello world, This is an app to connect to sql server.");
var config = {
        "user": "myusername", //default is sa
        "password": "yourStrong(!)Password",
        "server": "localhost", // for local machine
        "database": "staging", // name of database
        "options": {
            "encrypt": true
        }
      }

sql.connect(config, err => {
    if(err){
        throw err ;
    }
    console.log("Connection Successful !");

    new sql.Request().query('select 1 as number', (err, result) => {
        //handle err
        console.dir(result)
        // This example uses callbacks strategy for getting results.
    })

});

sql.on('error', err => {
    // ... error handler
    console.log("Sql database connection error " ,err);
})
```

Etape 4: C'est l'étape la plus simple, où nous démarrons l'application et où l'application se connecte au serveur sql et imprime des résultats simples.

```
node App.js
// Output :
// Hello world, This is an app to connect to sql server.
// Connection Successful !
// 1
```

> Pour utiliser les promesses ou async pour l'exécution des requêtes, reportez-vous aux documents officiels du package mssql:
>
> - Promesses
> - Async / Attente

Lire Intégration MSSQL en ligne: https://riptutorial.com/fr/node-js/topic/9884/integration-mssql

Chapitre 61: Intégration MySQL

Introduction

Dans cette rubrique, vous apprendrez comment intégrer Node.js à l'aide de l'outil de gestion de base de données MYSQL. Vous apprendrez différentes manières de vous connecter et d'interagir avec des données résidant dans mysql en utilisant un script et un programme nodejs.

Examples

Interroger un objet de connexion avec des paramètres

Lorsque vous souhaitez utiliser le contenu généré par l'utilisateur dans le SQL, cela se fait avec des paramètres. Par exemple, pour rechercher un utilisateur portant le nom `aminadav` vous devez faire:

```
var username = 'aminadav';
var querystring = 'SELECT name, email from users where name = ?';
connection.query(querystring, [username], function(err, rows, fields) {
  if (err) throw err;
  if (rows.length) {
    rows.forEach(function(row) {
      console.log(row.name, 'email address is', row.email);
    });
  } else {
    console.log('There were no results.');
  }
});
```

Utiliser un pool de connexions

une. Exécuter plusieurs requêtes en même temps

Toutes les requêtes en connexion MySQL se font l'une après l'autre. Cela signifie que si vous voulez faire 10 requêtes et que chaque requête prend 2 secondes, cela prendra 20 secondes pour terminer l'exécution. La solution consiste à créer une connexion 10 et à exécuter chaque requête dans une connexion différente. Cela peut être fait automatiquement en utilisant le pool de connexion

```
var pool  = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass',
  database        : 'schema'
});

for(var i=0;i<10;i++){
```

```
  pool.query('SELECT ` as example', function(err, rows, fields) {
    if (err) throw err;
    console.log(rows[0].example); //Show 1
  });
 }
```

Il exécutera les 10 requêtes en parallèle.

Lorsque vous utilisez un `pool` vous n'avez plus besoin de la connexion. Vous pouvez interroger directement la piscine. Le module MySQL recherchera la prochaine connexion gratuite pour exécuter votre requête.

b. Atteindre la multi-location sur un serveur de base de données avec différentes bases de données hébergées sur celui-ci.

La multitenancy est une exigence courante des applications d'entreprise de nos jours et la création d'un pool de connexions pour chaque base de données sur un serveur de base de données n'est pas recommandée. Ainsi, nous pouvons au lieu de cela créer un pool de connexions avec le serveur de base de données, puis les basculer entre les bases de données hébergées sur le serveur de base de données à la demande.

Supposons que notre application dispose de différentes bases de données pour chaque entreprise hébergée sur un serveur de base de données. Nous allons nous connecter à la base de données de l'entreprise respective lorsque l'utilisateur accède à l'application. Voici l'exemple sur la façon de faire cela: -

```
var pool  = mysql.createPool({
      connectionLimit : 10,
      host            : 'example.org',
      user            : 'bobby',
      password        : 'pass'
    });

pool.getConnection(function(err, connection){
    if(err){
        return cb(err);
    }
    connection.changeUser({database : "firm1"});
    connection.query("SELECT * from history", function(err, data){
        connection.release();
        cb(err, data);
    });
});
```

Laissez-moi décomposer l'exemple: -

Lors de la définition de la configuration du pool, je n'ai pas donné le nom de la base de données, mais uniquement le serveur de base de données, c.-à-d.

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

Ainsi, lorsque nous voulons utiliser la base de données spécifique sur le serveur de base de données, nous demandons à la connexion d'accéder à la base de données en utilisant: -

```
    connection.changeUser({database : "firm1"});
```

vous pouvez consulter la documentation officielle ici

Se connecter à MySQL

L'un des moyens les plus simples de se connecter à MySQL est d'utiliser le module `mysql` . Ce module gère la connexion entre l'application Node.js et le serveur MySQL. Vous pouvez l'installer comme n'importe quel autre module:

```
npm install --save mysql
```

Maintenant, vous devez créer une connexion mysql, que vous pourrez ensuite interroger.

```
const mysql      = require('mysql');
const connection = mysql.createConnection({
  host     : 'localhost',
  user     : 'me',
  password : 'secret',
  database : 'database_schema'
});

connection.connect();

// Execute some query statements
// I.e. SELECT * FROM FOO

connection.end();
```

Dans l'exemple suivant, vous apprendrez à interroger l'objet de `connection` .

Interroger un objet de connexion sans paramètres

Vous envoyez la requête en tant que chaîne et en réponse, le rappel avec la réponse est reçu. Le rappel vous donne une `error` , un tableau de `rows` et de champs. Chaque ligne contient toute la colonne de la table renvoyée. Voici un extrait pour l'explication suivante.

```
connection.query('SELECT name,email from users', function(err, rows, fields) {
  if (err) throw err;

  console.log('There are:', rows.length,' users');
  console.log('First user name is:',rows[0].name)
```

```
});
```

Exécuter un certain nombre de requêtes avec une seule connexion à partir d'un pool

Il peut y avoir des situations où vous avez configuré un pool de connexions MySQL, mais vous avez plusieurs requêtes à exécuter en séquence:

```
SELECT 1;
SELECT 2;
```

Vous *pourriez* simplement exécuter en utilisant `pool.query` comme vu ailleurs , cependant si vous n'avez qu'une seule connexion libre dans le pool, vous devez attendre qu'une connexion devienne disponible avant de pouvoir exécuter la deuxième requête.

Vous pouvez toutefois conserver une connexion active depuis le pool et exécuter autant de requêtes que vous le souhaitez en utilisant une seule connexion à l'aide de `pool.getConnection` :

```
pool.getConnection(function (err, conn) {
  if (err) return callback(err);

  conn.query('SELECT 1 AS seq', function (err, rows) {
    if (err) throw err;

    conn.query('SELECT 2 AS seq', function (err, rows) {
      if (err) throw err;

      conn.release();
      callback();
    });
  });
});
```

Note: Vous devez vous rappeler de `release` la connexion, sinon il y a une connexion MySQL de moins disponible pour le reste du pool!

Pour plus d'informations sur la mise en commun des connexions MySQL, consultez les documents MySQL .

Renvoyer la requête en cas d'erreur

Vous pouvez attacher la requête exécutée à votre objet `err` lorsqu'une erreur se produit:

```
var q = mysql.query('SELECT `name` FROM `pokedex` WHERE `id` = ?', [ 25 ], function (err,
result) {
  if (err) {
    // Table 'test.pokedex' doesn't exist
    err.query = q.sql; // SELECT `name` FROM `pokedex` WHERE `id` = 25
    callback(err);
  }
  else {
    callback(null, result);
```

```
  }
});
```

Pool de connexions d'exportation

```
// db.js

const mysql = require('mysql');

const pool = mysql.createPool({
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bob',
  password        : 'secret',
  database        : 'my_db'
});

module.export = {
  getConnection: (callback) => {
    return pool.getConnection(callback);
  }
}
```

```
// app.js

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

db.getConnection((err, conn) => {
  conn.query('SELECT something from sometable', (error, results, fields) => {
    // get the results
    conn.release();
  });
});
```

Lire Intégration MySQL en ligne: https://riptutorial.com/fr/node-js/topic/1406/integration-mysql

Chapitre 62: Intégration PostgreSQL

Examples

Se connecter à PostgreSQL

Utilisation du module npm `PostgreSQL` .

installer la dépendance à partir de npm

```
npm  install pg --save
```

Maintenant, vous devez créer une connexion PostgreSQL, que vous pouvez interroger ultérieurement.

Supposons que Database_Name = students, Host = localhost et DB_User = postgres

```
var pg = require("pg")
var connectionString = "pg://postgres:postgres@localhost:5432/students";
var client = new pg.Client(connectionString);
client.connect();
```

Requête avec objet de connexion

Si vous souhaitez utiliser un objet de connexion pour la base de données de requêtes, vous pouvez utiliser cet exemple de code.

```
var queryString = "SELECT name, age FROM students " ;
var query = client.query(queryString);

query.on("row", (row, result)=> {
result.addRow(row);
});

query.on("end", function (result) {
//LOGIC
});
```

Lire Intégration PostgreSQL en ligne: https://riptutorial.com/fr/node-js/topic/7706/integration-postgresql

Chapitre 63: Interagir avec la console

Syntaxe

- console.log ([data] [, ...])
- console.error ([data] [, ...])
- console.time (label)
- console.timeEnd (label)

Examples

Enregistrement

Module de console

Similaire à l'environnement de navigateur de JavaScript, node.js fournit un module de **console** qui offre des possibilités simples de journalisation et de débogage.

Les méthodes les plus importantes fournies par le module console sont `console.log` , `console.error` et `console.time` . Mais il y a plusieurs autres comme `console.info` .

console.log

Les paramètres seront imprimés sur la sortie standard (`stdout`) avec une nouvelle ligne.

```
console.log('Hello World');
```

```
> console.log('Hello World')
Hello World
```

console.error

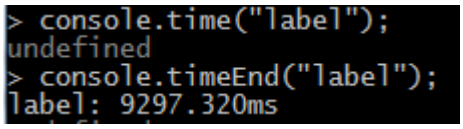
Les paramètres seront imprimés avec l'erreur standard (`stderr`) avec une nouvelle ligne.

```
console.error('Oh, sorry, there is an error.');
```

```
> console.error("Oh, sorry, error");
Oh, sorry, error
```

console.time, console.timeEnd

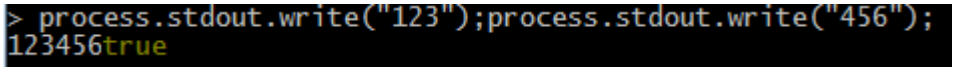
`console.time` démarre une minuterie avec un repère unique pouvant être utilisé pour calculer la durée d'une opération. Lorsque vous appelez `console.timeEnd` avec le même libellé, le minuteur s'arrête et imprime le temps écoulé en millisecondes à la `stdout` .



Module de processus

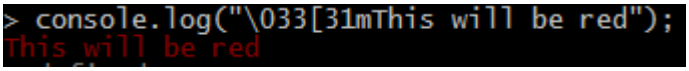
Il est possible d'utiliser le module de **processus** pour écrire **directement** dans la sortie standard de la console. Par conséquent, il existe la méthode `process.stdout.write` . Contrairement à `console.log` cette méthode n'ajoute pas de nouvelle ligne avant votre sortie.

Ainsi, dans l'exemple suivant, la méthode est appelée deux fois, mais aucune nouvelle ligne n'est ajoutée entre leurs sorties.



Mise en forme

On peut utiliser **des codes** de **terminal (de contrôle)** pour émettre des commandes spécifiques comme le changement de couleurs ou le positionnement du curseur.



Général

Effet	Code
Réinitialiser	`\033[0m`
Hicolore	`\033[1m`
Souligner	`\033[4m`
Inverse	`\033[7m`

Couleurs de police

Effet	Code
Noir	`\033[30m`
rouge	`\033[31m`
vert	`\033[32m`

Effet	Code
Jaune	`\033[33m`
Bleu	`\033[34m`
Magenta	`\033[35m`
Cyan	`\033[36m`
blanc	`\033[37m`

Couleurs de fond

Effet	Code
Noir	`\033[40m`
rouge	`\033[41m`
vert	`\033[42m`
Jaune	`\033[43m`
Bleu	`\033[44m`
Magenta	`\033[45m`
Cyan	`\033[46m`
blanc	`\033[47m`

Lire Interagir avec la console en ligne: https://riptutorial.com/fr/node-js/topic/5935/interagir-avec-la-console

Chapitre 64: Koa Framework v2

Examples

Bonjour exemple mondial

```
const Koa = require('koa')

const app = new Koa()

app.use(async ctx => {
   ctx.body = 'Hello World'
})

app.listen(8080)
```

Gestion des erreurs à l'aide du middleware

```
app.use(async (ctx, next) => {
  try {
    await next() // attempt to invoke the next middleware downstream
  } catch (err) {
    handleError(err, ctx) // define your own error handling function
  }
})
```

Lire Koa Framework v2 en ligne: https://riptutorial.com/fr/node-js/topic/6730/koa-framework-v2

Chapitre 65: Livrer du code HTML ou tout autre type de fichier

Syntaxe

- response.sendFile (nom_fichier, options, fonction (err) {});

Examples

Livrer le HTML au chemin spécifié

Voici comment créer un serveur Express et servir `index.html` par défaut (chemin vide `/`), et `page1.html` pour `/page1` chemin.

Structure de dossier

```
project root
|    server.js
|____views
     |    index.html
     |    page1.html
```

server.js

```
var express = require('express');
var path = require('path');
var app = express();

// deliver index.html if no file is requested
app.get("/", function (request, response) {
  response.sendFile(path.join(__dirname, 'views/index.html'));
});

// deliver page1.html if page1 is requested
app.get('/page1', function(request, response) {
    response.sendFile(path.join(__dirname, 'views', 'page1.html', function(error) {
        if (error) {
            // do something in case of error
            console.log(err);
            response.end(JSON.stringify({error:"page not found"}));
        }
    });
});

app.listen(8080);
```

Notez que `sendFile()` ne fait que `sendFile()` un fichier statique en réponse, sans possibilité de le

modifier. Si vous diffusez un fichier HTML et souhaitez y inclure des données dynamiques, vous devez utiliser un *moteur de modèle* tel que Pug, Moustache ou EJS.

Lire Livrer du code HTML ou tout autre type de fichier en ligne: https://riptutorial.com/fr/node-js/topic/6538/livrer-du-code-html-ou-tout-autre-type-de-fichier

Chapitre 66: Localisation du noeud JS

Introduction

Il est très facile de maintenir la localisation nodejs express

Examples

utiliser le module i18n pour maintenir la localisation dans le noeud js app

Module de traduction simple et léger avec stockage json dynamique. Prend en charge les applications plain vanilla node.js et devrait fonctionner avec tout framework (comme express, restify et probablement plus) qui expose une méthode app.use () transmettant des objets res et req. Utilise la syntaxe commune __ ('...') dans les applications et les modèles. Stocke les fichiers de langue dans des fichiers json compatibles avec le format webtranslateit json. Ajoute de nouvelles chaînes à la volée lors de leur première utilisation dans votre application. Aucune analyse supplémentaire nécessaire.

express + i18n-node + cookieParser et éviter les problèmes de concurrence

```
// usual requirements
var express = require('express'),
    i18n = require('i18n'),
    app = module.exports = express();

i18n.configure({
  // setup some locales - other locales default to en silently
  locales: ['en', 'ru', 'de'],

  // sets a custom cookie name to parse locale settings from
  cookie: 'yourcookiename',

  // where to store json files - defaults to './locales'
  directory: __dirname + '/locales'
});

app.configure(function () {
  // you will need to use cookieParser to expose cookies to req.cookies
  app.use(express.cookieParser());

  // i18n init parses req for language headers, cookies, etc.
  app.use(i18n.init);

});

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

// starting server
if (!module.parent) {
```

```
  app.listen(3000);
}
```

Lire Localisation du noeud JS en ligne: https://riptutorial.com/fr/node-js/topic/9594/localisation-du-noeud-js

Chapitre 67: Lodash

Introduction

Lodash est une bibliothèque utilitaire JavaScript pratique.

Examples

Filtrer une collection

L'extrait de code ci-dessous montre les différentes manières de filtrer un tableau d'objets à l'aide de lodash.

```
let lodash = require('lodash');

var countries = [
    {"key": "DE", "name": "Deutschland", "active": false},
    {"key": "ZA", "name": "South Africa", "active": true}
];

var filteredByFunction = lodash.filter(countries, function (country) {
    return country.key === "DE";
});
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByObjectProperties = lodash.filter(countries, { "key": "DE" });
// => [{"key": "DE", "name": "Deutschland"}];

var filteredByProperties = lodash.filter(countries, ["key", "ZA"]);
// => [{"key": "ZA", "name": "South Africa"}];

var filteredByProperty = lodash.filter(countries, "active");
// => [{"key": "ZA", "name": "South Africa"}];
```

Lire Lodash en ligne: https://riptutorial.com/fr/node-js/topic/9161/lodash

Chapitre 68: Loopback - Connecteur basé sur REST

Introduction

Connecteurs basés sur le repos et comment les gérer. Nous savons tous que Loopback n'offre pas d'élégance aux connexions basées sur REST

Examples

Ajout d'un connecteur Web

```
// Cet exemple obtient la réponse d'iTunes
{
  "du repos": {
    "nom": "reste",
    "connecteur": "reste",
    "debug": true,
    "options": {
      "useQuerystring": true,
      "timeout": 10000,
      "en-têtes": {
        "accepte": "application / json",
        "content-type": "application / json"
      }
    },
    "opérations": [
      {
        "modèle": {
          "méthode": "GET",
          "url": "https://itunes.apple.com/search",
          "requête": {
            "terme": "{mot clé}",
            "pays": "{pays = IN}",
            "media": "{itemType = music}",
            "limite": "{limite = 10}",
            "explicite": "faux"
          }
        },
        "les fonctions": {
          "chercher": [
            "mot-clé",
            "pays",
            "type d'élément",
            "limite"
          ]
        }
      },
      {
        "modèle": {
          "méthode": "GET",
          "url": "https://itunes.apple.com/lookup",
          "requête": {
```

```
            "J'ai fait}"
          }
        },
        "les fonctions": {
          "findById": [
            "id"
          ]
        }
      }
    ]
  }
}
```

Lire Loopback - Connecteur basé sur REST en ligne: https://riptutorial.com/fr/node-js/topic/9234/loopback---connecteur-base-sur-rest

Chapitre 69: Module de cluster

Syntaxe

- const cluster = require ("cluster")
- cluster.fork ()
- cluster.isMaster
- cluster.isWorker
- cluster.schedulingPolicy
- cluster.setupMaster (paramètres)
- cluster.settings
- cluster.worker // in worker
- cluster.workers // dans master

Remarques

Notez que `cluster.fork()` génère un processus enfant qui commence à exécuter le script depuis le début, contrairement à l'appel système `fork()` dans *C* qui clone le processus en cours et continue à partir de l'instruction après l'appel système dans parent et processus enfant.

La documentation de Node.js contient un guide plus complet sur les clusters ici

Examples

Bonjour le monde

Ceci est votre `cluster.js` :

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

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

  cluster.on('exit', (worker, code, signal) => {
    console.log(`worker ${worker.process.pid} died`);
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  require('./server.js')();
}
```

Ceci est votre `server.js` principal:

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

function startServer() {
    const server = http.createServer((req, res) => {
      res.writeHead(200);
      res.end('Hello Http');
    });

    server.listen(3000);
}

if(!module.parent) {
    // Start server if file is run directly
    startServer();
} else {
    // Export server, if file is referenced via cluster
    module.exports = startServer;
}
```

Dans cet exemple, nous hébergeons un serveur Web de base, mais nous intégrons les employés (processus enfants) à l'aide du module de **cluster** intégré. Le nombre de processus forker dépend du nombre de cœurs de processeur disponibles. Cela permet à une application Node.js de tirer parti des processeurs multicœurs, car une seule instance de Node.js s'exécute dans un seul thread. L'application va maintenant partager le port 8000 sur tous les processus. Les charges seront automatiquement réparties entre les travailleurs utilisant la méthode Round-Robin par défaut.

Exemple de cluster

Une seule instance de `Node.js` s'exécute dans un seul thread. Pour tirer parti des systèmes multi-core, l'application peut être lancée dans un cluster de processus Node.js pour gérer la charge.

Le module de `cluster` vous permet de créer facilement des processus enfants partageant tous des ports de serveur.

L'exemple suivant crée le processus enfant travailleur dans le processus principal qui gère la charge sur plusieurs cœurs.

Exemple

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length; //number of CPUS

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

  //on exit of cluster
  cluster.on('exit', (worker, code, signal) => {
      if (signal) {
        console.log(`worker was killed by signal: ${signal}`);
      } else if (code !== 0) {
```

```
        console.log(`worker exited with error code: ${code}`);
      } else {
        console.log('worker success!');
      }
  });
} else {
  // Workers can share any TCP connection
  // In this case it is an HTTP server
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(3000);
}
```

Lire Module de cluster en ligne: https://riptutorial.com/fr/node-js/topic/2817/module-de-cluster

Chapitre 70: Multithreading

Introduction

Node.js a été conçu pour être mono-threadé. Donc, à toutes fins utiles, les applications qui démarrent avec Node s'exécuteront sur un seul thread.

Cependant, Node.js lui-même s'exécute en multi-thread. Les opérations d'E / S et autres s'exécuteront à partir d'un pool de threads. En outre, toute instance d'une application de noeud s'exécutera sur un thread différent. Par conséquent, pour exécuter des applications multithread, vous devez lancer plusieurs instances.

Remarques

Comprendre la boucle d'événement est important pour comprendre comment et pourquoi utiliser plusieurs threads.

Examples

Grappe

Le module `cluster` permet de démarrer la même application plusieurs fois.

Le regroupement est souhaitable lorsque les différentes instances ont le même flux d'exécution et ne dépendent pas les unes des autres. Dans ce scénario, vous avez un maître qui peut démarrer les fourches et les fourches (ou les enfants). Les enfants travaillent de manière autonome et disposent d'un espace unique entre Ram et Event Loop.

La configuration de clusters peut être bénéfique pour les sites Web / API. Tout thread peut servir n'importe quel client, car il ne dépend pas d'autres threads. Une base de données (comme Redis) serait utilisée pour partager les cookies, car les **variables ne peuvent pas être partagées!** entre les fils.

```
// runs in each instance
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;

console.log('I am always called');

if (cluster.isMaster) {
    // runs only once (within the master);
    console.log('I am the master, launching workers!');
    for(var i = 0; i < numCPUs; i++) cluster.fork();

} else {
    // runs in each fork
    console.log('I am a fork!');
```

```
    // here one could start, as an example, a web server

}

console.log('I am always called as well');
```

Processus de l'enfant

Les processus enfants sont la voie à suivre pour exécuter des processus de manière indépendante avec des initialisations et des préoccupations différentes. Comme les fourches dans les clusters, un `child_process` exécuté dans son thread, mais contrairement aux forks, il a un moyen de communiquer avec ses parents.

La communication se fait dans les deux sens, afin que les parents et les enfants puissent écouter les messages et envoyer des messages.

Parent (../parent.js)

```
var child_process = require('child_process');
console.log('[Parent]', 'initalize');

var child1 = child_process.fork(__dirname + '/child');
child1.on('message', function(msg) {
    console.log('[Parent]', 'Answer from child: ', msg);
});

// one can send as many messages as one want
child1.send('Hello'); // Hello to you too :)
child1.send('Hello'); // Hello to you too :)

// one can also have multiple children
var child2 = child_process.fork(__dirname + '/child');
```

Enfant (../child.js)

```
// here would one initialize this child
// this will be executed only once
console.log('[Child]', 'initalize');

// here one listens for new tasks from the parent
process.on('message', function(messageFromParent) {

    //do some intense work here
    console.log('[Child]', 'Child doing some intense work');

    if(messageFromParent == 'Hello') process.send('Hello to you too :)');
    else process.send('what?');

})
```

À côté du message, vous pouvez écouter de nombreux événements tels que «erreur», «connecté» ou «déconnecter».

Le démarrage d'un processus enfant est associé à un certain coût. On voudrait en engendrer le

moins possible.

Lire Multithreading en ligne: https://riptutorial.com/fr/node-js/topic/10592/multithreading

Chapitre 71: N-API

Introduction

La N-API est un nouveau et meilleur moyen de créer un module natif pour NodeJS. N-API est au début, il peut donc y avoir une documentation incohérente.

Examples

Bonjour à N-API

Ce module enregistre la fonction hello sur le module hello. bonjour la fonction imprime Bonjour tout le monde sur la console avec `printf` et retourne `1373` de la fonction native dans l'appelant javascript.

```
#include <node_api.h>
#include <stdio.h>


napi_value say_hello(napi_env env, napi_callback_info info)
{
    napi_value retval;

    printf("Hello world\n");

    napi_create_number(env, 1373, &retval);

    return retval;
}

void init(napi_env env, napi_value exports, napi_value module, void* priv)
{
    napi_status status;
    napi_property_descriptor desc = {
        /*
         * String describing the key for the property, encoded as UTF8.
         */
        .utf8name = "hello",
        /*
         * Set this to make the property descriptor object's value property
         * to be a JavaScript function represented by method.
         * If this is passed in, set value, getter and setter to NULL (since these members
won't be used).
         */
        .method = say_hello,
        /*
         * A function to call when a get access of the property is performed.
         * If this is passed in, set value and method to NULL (since these members won't be
used).
         * The given function is called implicitly by the runtime when the property is
accessed
         * from JavaScript code (or if a get on the property is performed using a N-API call).
         */
```

```
        .getter = NULL,
        /*
         * A function to call when a set access of the property is performed.
         * If this is passed in, set value and method to NULL (since these members won't be
used).
         * The given function is called implicitly by the runtime when the property is set
         * from JavaScript code (or if a set on the property is performed using a N-API call).
         */
        .setter = NULL,
        /*
         * The value that's retrieved by a get access of the property if the property is a
data property.
         * If this is passed in, set getter, setter, method and data to NULL (since these
members won't be used).
         */
        .value = NULL,
        /*
         * The attributes associated with the particular property. See
napi_property_attributes.
         */
        .attributes = napi_default,
        /*
         * The callback data passed into method, getter and setter if this function is
invoked.
         */
        .data = NULL
    };
    /*
     * This method allows the efficient definition of multiple properties on a given object.
     */
    status = napi_define_properties(env, exports, 1, &desc);

    if (status != napi_ok)
        return;
}


NAPI_MODULE(hello, init)
```

Lire N-API en ligne: https://riptutorial.com/fr/node-js/topic/10539/n-api

Chapitre 72: Node.js (express.js) avec angular.js Exemple de code

Introduction

Cet exemple montre comment créer une application express de base, puis diffuser AngularJS.

Examples

Créer notre projet

Nous sommes prêts à y aller, nous courons à nouveau depuis la console:

```
mkdir our_project
cd our_project
```

Maintenant, nous sommes à l'endroit où notre code va vivre. Pour créer l'archive principale de notre projet, vous pouvez exécuter

Ok, mais comment créer le projet squelette express?

C'est simple:

```
npm install -g express express-generator
```

Les distributions Linux et Mac doivent utiliser **sudo** pour l'installer car elles sont installées dans le répertoire nodejs accessible uniquement par l'utilisateur **root** . Si tout se passe bien, nous pouvons enfin créer le squelette de l’application express.

```
express
```

Cette commande créera dans notre dossier une application d’exemple express. La structure est la suivante:

```
bin/
public/
routes/
views/
app.js
package.json
```

Maintenant, si vous lancez **npm start** an go sur http: // localhost: 3000, nous verrons l’application express opérationnelle, nous avons généré une application express sans trop de problèmes, mais comment peut-on mélanger avec AngularJS? .

Comment s'exprimer fonctionne, brièvement?

Express est un framework construit sur **Nodejs** , vous pouvez voir la documentation officielle sur le site Express . Mais pour notre propos, nous devons savoir **qu'Express** est le responsable lorsque nous tapons, par exemple, http: // localhost: 3000 / home pour rendre la page d'accueil de notre application. Depuis l'application créée récemment, nous pouvons vérifier:

```
FILE: routes/index.js
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

Ce que ce code nous dit, c'est que lorsque l'utilisateur accède à http: // localhost: 3000, il doit afficher la vue d' **index** et transmettre un **JSON** avec une propriété title et une valeur Express. Mais lorsque nous vérifions le répertoire des vues et ouvrons index.jade, nous pouvons voir ceci:

```
extends layout
block content
    h1= title
    p Welcome to #{title}
```

Ceci est une autre fonctionnalité Express puissante, **les moteurs de template** , ils vous permettent de rendre du contenu dans la page en lui passant des variables ou d'hériter d'un autre modèle pour que vos pages soient plus compactes et plus compréhensibles par les autres. L'extension du fichier est **.jade** pour autant que je sache que **Jade a** changé le nom de **Pug** , essentiellement le même moteur de template mais avec quelques mises à jour et modifications de base.

Installation de Pug et mise à jour du moteur de modèle Express.

Ok, pour commencer à utiliser Pug comme moteur de modèle de notre projet, nous devons exécuter:

```
npm install --save pug
```

Cela va installer Pug comme une dépendance de notre projet et l'enregistrer dans **package.json** . Pour l'utiliser, il faut modifier le fichier **app.js** :

```
var app = express();
// view engine setup
app.set('views', path.join(__dirname, 'views'));
```

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

Et remplacez le moteur de ligne de visée par pug et c'est tout. Nous pouvons relancer notre projet avec **npm start** et nous verrons que tout fonctionne bien.

Comment AngularJS s'intègre-t-il dans tout cela?

AngularJS est un **framework** Javascript **MVW** (Model-View-Whatever) principalement utilisé pour créer une installation **SPA** (Simple Page Application) assez simple, vous pouvez aller sur le site AngularJS et télécharger la dernière version qui est la **v1.6.4** .

Après avoir téléchargé AngularJS quand devrait copier le fichier dans notre dossier **public / javascripts à l'** intérieur de notre projet, une petite explication, c'est le dossier qui sert les ressources statiques de notre site, images, css, fichiers javacript, etc. Bien sûr, cela est configurable via le fichier **app.js** , mais nous allons rester simple. Maintenant, nous créons un fichier nommé **ng-app.js** , le fichier dans lequel notre application vivra, dans notre dossier public javascripts, là où vit AngularJS. Pour que AngularJS soit opérationnel, nous devons modifier le contenu de **views / layout.pug** comme suit:

```
doctype html
html(ng-app='first-app')
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body(ng-controller='indexController')
    block content

  script(type='text-javascript', src='javascripts/angular.min.js')
  script(type='text-javascript', src='javascripts/ng-app.js')
```

Que faisons-nous ici? Eh bien, nous incluons le noyau AngularJS et notre fichier récemment créé **ng-app.js** . Lorsque le modèle sera rendu, il **ajoutera** AngularJS, notera l'utilisation de la directive **ng-app** , cela dit AngularJS que c'est le nom de notre application et qu'elle devrait s'y tenir. Ainsi, le contenu de notre **ng-app.js** sera:

```
angular.module('first-app', [])
       .controller('indexController', ['$scope', indexController]);

function indexController($scope) {
    $scope.name = 'sigfried';
}
```

Nous utilisons la fonctionnalité AngularJS la plus simple ici, **la liaison de données bidirectionnelle** , ce qui nous permet de rafraîchir instantanément le contenu de notre vue et de notre contrôleur. C'est une explication très simple, mais vous pouvez faire une recherche dans Google ou StackOverflow pour voir comment ça marche vraiment

Donc, nous avons les blocs de base de notre application AngularJS, mais il y a quelque chose que nous devons faire, nous devons mettre à jour notre page index.pug pour voir les modifications de notre application angulaire, faisons-le:

```
extends layout
block content
  div(ng-controller='indexController')
    h1= title
    p Welcome {{name}}
    input(type='text' ng-model='name')
```

Ici, nous ne faisons que lier l'entrée à notre nom de propriété défini dans la portée AngularJS de notre contrôleur:

```
$scope.name = 'sigfried';
```

Le but de ceci est que chaque fois que nous changeons le texte dans l'entrée le paragraphe ci-dessus le mettra à jour dans le {{name}}, cela s'appelle l' **interpolation** , encore une autre fonction AngularJS pour rendre notre contenu dans le template.

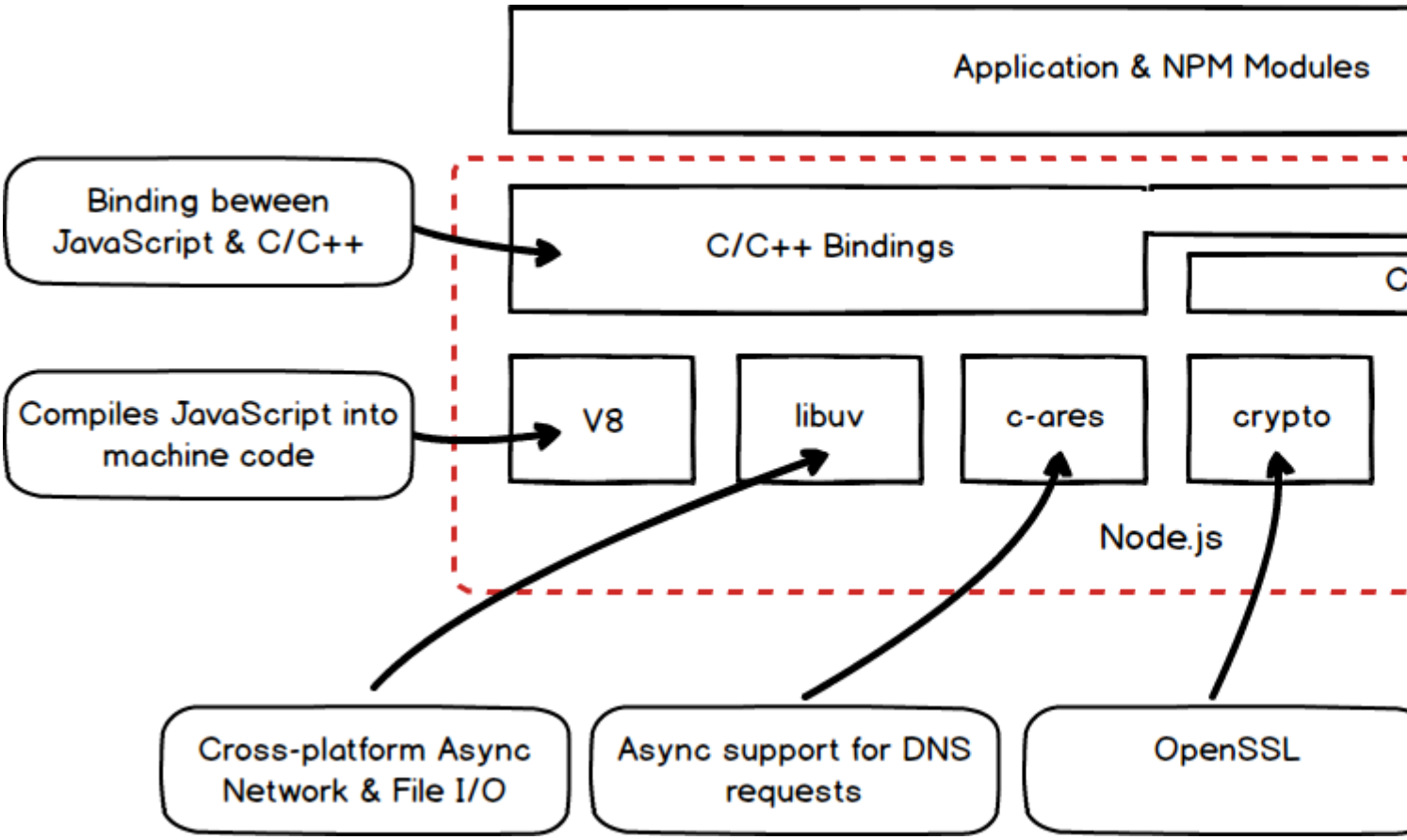
Donc, tout est paramétré, nous pouvons maintenant lancer **npm start** sur http: // localhost: 3000 et voir notre application express au service de la page et AngularJS gérant le front-end de l'application.

Lire Node.js (express.js) avec angular.js Exemple de code en ligne: https://riptutorial.com/fr/node-js/topic/9757/node-js--express-js--avec-angular-js-exemple-de-code

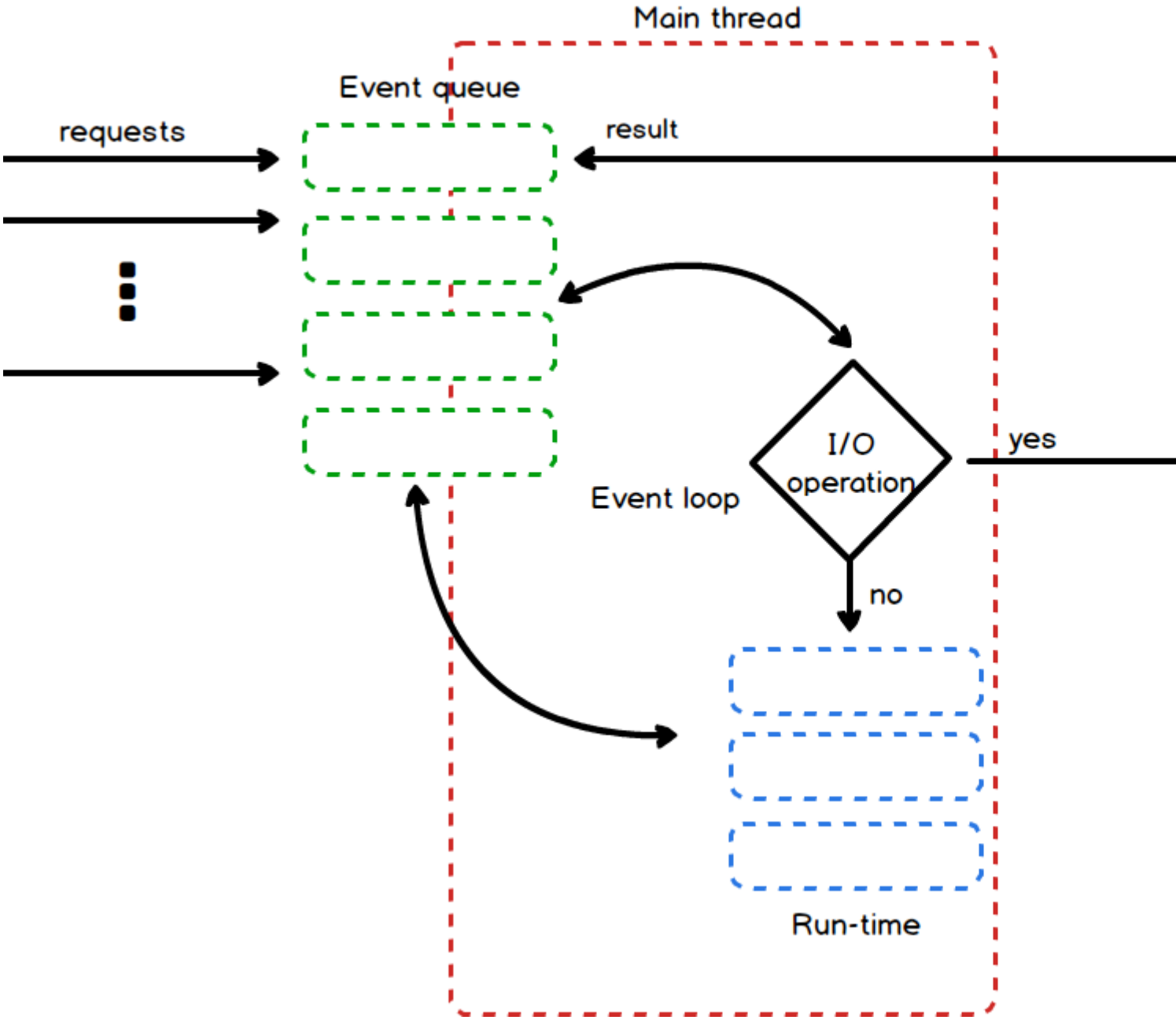
Chapitre 73: Node.js Architecture & Inner Workings

Examples

Node.js - sous le capot



Node.js - en mouvement



Lire Node.js Architecture & Inner Workings en ligne: https://riptutorial.com/fr/node-js/topic/5892/node-js-architecture--amp--inner-workings

Chapitre 74: Node.js avec CORS

Examples

Activer CORS dans express.js

Comme node.js est souvent utilisé pour créer une API, un paramètre CORS correct peut vous sauver la vie si vous souhaitez pouvoir demander l'API à différents domaines.

Dans l'exemple, nous allons le configurer pour la configuration plus large (autoriser tous les types de requêtes à partir de n'importe quel domaine).

Dans votre server.js après initialisation express:

```
// Create express server
const app = express();

app.use((req, res, next) => {
    res.header('Access-Control-Allow-Origin', '*');

    // authorized headers for preflight requests
    // https://developer.mozilla.org/en-US/docs/Glossary/preflight_request
    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type,
Accept');
    next();

    app.options('*', (req, res) => {
        // allowed XHR methods
        res.header('Access-Control-Allow-Methods', 'GET, PATCH, PUT, POST, DELETE, OPTIONS');
        res.send();
    });
});
```

Généralement, le noeud est exécuté derrière un proxy sur les serveurs de production. Par conséquent, le serveur proxy inverse (tel qu'Apache ou Nginx) sera responsable de la configuration CORS.

Pour adapter facilement ce scénario, il est possible d'activer uniquement le fichier node.js CORS lorsqu'il est en développement.

Cela se fait facilement en cochant `NODE_ENV` :

```
const app = express();

if (process.env.NODE_ENV === 'development') {
    // CORS settings
}
```

Lire Node.js avec CORS en ligne: https://riptutorial.com/fr/node-js/topic/9272/node-js-avec-cors

Chapitre 75: Node.JS avec ES6

Introduction

ES6, ECMAScript 6 ou ES2015 est la dernière spécification de JavaScript qui introduit du sucre syntaxique dans le langage. C'est une grande mise à jour du langage et introduit de nombreuses nouvelles fonctionnalités

Plus de détails sur Node et ES6 peuvent être trouvés sur leur site https://nodejs.org/en/docs/es6/

Examples

Node ES6 Support et création d'un projet avec Babel

L'ensemble de la spécification ES6 n'est pas encore implémenté dans son intégralité, vous ne pourrez donc utiliser que certaines des nouvelles fonctionnalités. Vous pouvez voir une liste des fonctionnalités ES6 actuellement prises en charge sur http://node.green/

Depuis NodeJS v6, il y a eu un bon support. Donc, si vous utilisez NodeJS v6 ou supérieur, vous pouvez utiliser ES6. Cependant, vous souhaiterez peut-être également utiliser certaines fonctionnalités inédites et d'autres encore. Pour cela, vous devrez utiliser un transpiler

Il est possible d'exécuter un transpiler à l'exécution et de générer, pour utiliser toutes les fonctionnalités d'ES6 et plus encore. Le transpiler le plus populaire pour JavaScript s'appelle Babel

Babel vous permet d'utiliser toutes les fonctionnalités de la spécification ES6 et quelques fonctionnalités supplémentaires non spécifiées avec 'stage-0' comme `import thing from 'thing` quelque `import thing from 'thing` place de la `var thing = require('thing')`

Si nous voulions créer un projet où nous utiliserions des fonctionnalités telles que l'importation, nous devrons ajouter Babel en tant que transpiler. Vous verrez des projets utilisant react et Vue et d'autres patterns basés sur common JS implémentent assez souvent stage-0.

créer un nouveau projet de noeud

```
mkdir my-es6-app
cd my-es6-app
npm init
```

Installer babel le preset ES6 et stage-0

```
npm install --save-dev babel-preset-es2015 babel-preset-stage-2 babel-cli babel-register
```

Créez un nouveau fichier appelé `server.js` et ajoutez un serveur HTTP de base.

```
import http from 'http'
```

```
http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'})
  res.end('Hello World\n')
}).listen(3000, '127.0.0.1')

console.log('Server running at http://127.0.0.1:3000/')
```

Notez que nous utilisons un `import http from 'http'` il s'agit d'une fonctionnalité de niveau 0 et si cela fonctionne, cela signifie que le transpiler fonctionne correctement.

Si vous exécutez `node server.js` ne saurez pas comment gérer l'importation.

Créer un fichier .babelrc à la racine de votre répertoire et ajouter les paramètres suivants

```
{
  "presets": ["es2015", "stage-2"],
  "plugins": []
}
```

vous pouvez maintenant exécuter le serveur avec le `node src/index.js --exec babel-node`

Terminer ce processus n'est pas une bonne idée d'exécuter un transpiler à l'exécution sur une application de production. Nous pouvons cependant implémenter des scripts dans notre package.json pour le rendre plus facile à utiliser.

```
"scripts": {
    "start": "node dist/index.js",
    "dev": "babel-node src/index.js",
    "build": "babel src -d dist",
    "postinstall": "npm run build"
  },
```

L' `npm install` ci-dessus à `npm install` le code transpiled dans le répertoire dist, autorisez `npm start` à utiliser le code transpiled pour notre application de production.

`npm run dev` démarrera le serveur et le runel babel, ce qui est bien et préférable lorsque vous travaillez sur un projet en local.

En allant plus loin, vous pouvez installer nodemon `npm install nodemon --save-dev` pour surveiller les modifications, puis redémarrer l'application.

Cela accélère le travail avec babel et NodeJS. Dans vous package.json il suffit de mettre à jour le script "dev" pour utiliser nodemon

```
"dev": "nodemon src/index.js --exec babel-node",
```

Utilisez JS es6 sur votre application NodeJS

JS es6 (également connu sous le nom es2015) est un ensemble de nouvelles fonctionnalités pour le langage JS visant à le rendre plus intuitif lors de l'utilisation de la POO ou lors de tâches de développement modernes.

Conditions préalables:

1. Consultez les nouvelles fonctionnalités d'es6 sur http://es6-features.org - cela pourrait vous aider si vous avez vraiment l'intention de l'utiliser sur votre prochaine application NodeJS
2. Vérifiez le niveau de compatibilité de votre version de noeud à l' adresse http://node.green
3. Si tout va bien, codons!

Voici un exemple très court d'une application simple `hello world` avec JS es6

```
'use strict'

class Program
{
    constructor()
    {
        this.message = 'hello es6 :)';
    }

    print()
    {
        setTimeout(() =>
        {
            console.log(this.message);

            this.print();

        }, Math.random() * 1000);
    }
}

new Program().print();
```

Vous pouvez exécuter ce programme et observer comment il imprime le même message encore et encore.

Maintenant, laissez le décomposer ligne par ligne:

```
'use strict'
```

Cette ligne est en fait requise si vous avez l'intention d'utiliser js es6. `strict` mode `strict` , intentionnellement, a une sémantique différente du code normal (veuillez en lire plus sur MDN - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)

```
class Program
```

Incroyable - un mot `class` clé de `class` ! Juste pour une référence rapide - avant es6, la seule façon de définir une classe en js était d'utiliser le mot-clé ... `function` !

```
function MyClass() // class definition
```

```
{

}

var myClassObject = new MyClass(); // generating a new object with a type of MyClass
```

Lors de l'utilisation de la POO, une classe est une capacité très fondamentale qui aide le développeur à représenter une partie spécifique d'un système (la décomposition du code est cruciale lorsque le code grossit .. par exemple: lors de l'écriture du code côté serveur)

```
constructor()
{
    this.message = 'hello es6 :)';
}
```

Vous devez l'admettre - c'est assez intuitif! C'est le c'tor de ma classe - cette "fonction" unique se produira chaque fois qu'un objet est créé à partir de cette classe particulière (dans notre programme - une seule fois)

```
print()
{
    setTimeout(() => // this is an 'arrow' function
    {
        console.log(this.message);

        this.print(); // here we call the 'print' method from the class template itself (a
recursion in this particular case)

    }, Math.random() * 1000);
}
```

Parce que print est défini dans la portée de la classe - c'est en fait une méthode - qui peut être appelée à partir de l'objet de la classe ou de la classe elle-même!

Donc .. jusqu'à maintenant nous avons défini notre classe .. le temps de l'utiliser:

```
new Program().print();
```

Ce qui est vraiment égal à:

```
var prog = new Program(); // define a new object of type 'Program'

prog.print(); // use the program to print itself
```

En conclusion: JS es6 peut simplifier votre code - le rendre plus intuitif et facile à comprendre (en comparaison avec la version précédente de JS). Vous pouvez essayer de réécrire votre code existant et voir la différence par vous-même.

PRENDRE PLAISIR :)

Lire Node.JS avec ES6 en ligne: https://riptutorial.com/fr/node-js/topic/5934/node-js-avec-es6

Chapitre 76: Node.js avec Oracle

Examples

Se connecter à Oracle DB

Un moyen très simple de se connecter à une base de données ORACLE consiste à utiliser le module `oracledb` . Ce module gère la connexion entre votre application Node.js et le serveur Oracle. Vous pouvez l'installer comme n'importe quel autre module:

```
npm install oracledb
```

Vous devez maintenant créer une connexion ORACLE, que vous pouvez interroger ultérieurement.

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

oracledb.getConnection(
  {
    user          : "oli",
    password      : "password",
    connectString : "ORACLE_DEV_DB_TNS_NAME"
  },
  connExecute
);
```

ConnectString "ORACLE_DEV_DB_TNA_NAME" peut vivre dans un fichier tnsnames.org situé dans le même répertoire ou sur lequel votre client instantané Oracle est installé.

Si vous ne disposez d'aucun client instantané Oracle sur votre ordinateur de développement, vous pouvez suivre le `instant client installation guide` à votre système d'exploitation.

Interroger un objet de connexion sans paramètres

Utiliser peut maintenant utiliser la fonction connExecute pour exécuter une requête. Vous avez la possibilité d'obtenir le résultat de la requête en tant qu'objet ou tableau. Le résultat est imprimé sur console.log.

```
function connExecute(err, connection)
{
    if (err) {
        console.error(err.message);
        return;
    }
    sql = "select 'test' as c1, 'oracle' as c2 from dual";
    connection.execute(sql, {}, { outFormat: oracledb.OBJECT }, // or oracledb.ARRAY
        function(err, result)
        {
            if (err) {
                console.error(err.message);
```

```
                connRelease(connection);
                return;
            }
            console.log(result.metaData);
            console.log(result.rows);
            connRelease(connection);
        });
}
```

Comme nous avons utilisé une connexion sans pool, nous devons relancer notre connexion.

```
function connRelease(connection)
{
  connection.close(
    function(err) {
      if (err) {
        console.error(err.message);
      }
    });
}
```

La sortie pour un objet sera

```
[ { name: 'C1' }, { name: 'C2' } ]
[ { C1: 'test', C2: 'oracle' } ]
```

et la sortie pour un tableau sera

```
[ { name: 'C1' }, { name: 'C2' } ]
[ [ 'test', 'oracle' ] ]
```

Utiliser un module local pour faciliter les requêtes

Pour simplifier votre requête à partir d'ORACLE-DB, vous pouvez appeler votre requête comme ceci:

```
const oracle = require('./oracle.js');

const sql = "select 'test' as c1, 'oracle' as c2 from dual";
oracle.queryObject(sql, {}, {})
    .then(function(result) {
        console.log(result.rows[0]['C2']);
    })
    .catch(function(err) {
        next(err);
    });
```

La construction de la connexion et son exécution sont incluses dans ce fichier oracle.js avec le contenu suivant:

```
'use strict';
const oracledb = require('oracledb');
```

```
const oracleDbRelease = function(conn) {
  conn.release(function (err) {
    if (err)
      console.log(err.message);
  });
};

function queryArray(sql, bindParams, options) {
    options.isAutoCommit = false; // we only do SELECTs

    return new Promise(function(resolve, reject) {
        oracledb.getConnection(
                  {
                    user          : "oli",
                    password      : "password",
                    connectString : "ORACLE_DEV_DB_TNA_NAME"
                })
        .then(function(connection){
            //console.log("sql log: " + sql + " params " + bindParams);
            connection.execute(sql, bindParams, options)
            .then(function(results) {
                resolve(results);
                process.nextTick(function() {
                    oracleDbRelease(connection);
                });
            })
            .catch(function(err) {
                reject(err);

                process.nextTick(function() {
                    oracleDbRelease(connection);
                        });
                    });
            })
            .catch(function(err) {
                reject(err);
            });
    });
}

function queryObject(sql, bindParams, options) {
    options['outFormat'] = oracledb.OBJECT; // default is oracledb.ARRAY
    return queryArray(sql, bindParams, options);
}

module.exports = queryArray;
module.exports.queryArray = queryArray;
module.exports.queryObject = queryObject;
```

Notez que vous disposez des deux méthodes queryArray et queryObject pour appeler votre objet oracle.

Lire Node.js avec Oracle en ligne: https://riptutorial.com/fr/node-js/topic/8248/node-js-avec-oracle

Chapitre 77: Node.js code pour STDIN et STDOUT sans utiliser de bibliothèque

Introduction

Ceci est un programme simple dans node.js à qui prend l'entrée de l'utilisateur et l'imprime sur la console.

L'objet de **processus** est un objet global qui fournit des informations et contrôle le processus Node.js en cours. En tant que global, il est toujours disponible pour les applications Node.js sans utiliser require ().

Examples

Programme

La propriété **process.stdin** renvoie un flux lisible équivalent ou associé à stdin.

La propriété **process.stdout** renvoie un flux inscriptible équivalent ou associé à la sortie standard.

```
process.stdin.resume()
console.log('Enter the data to be displayed ');
process.stdin.on('data', function(data) { process.stdout.write(data) })
```

Lire Node.js code pour STDIN et STDOUT sans utiliser de bibliothèque en ligne: https://riptutorial.com/fr/node-js/topic/8961/node-js-code-pour-stdin-et-stdout-sans-utiliser-de-bibliotheque

Chapitre 78: Node.js Conception fondamentale

Examples

La philosophie de Node.js

Petit noyau , **petit module** : -

Construire des modules à objectif simple et petit, non seulement en termes de taille de code, mais également en termes de portée et de finalité

```
    a - "Small is beautiful"
    b - "Make each program do one thing well."
```

Le motif du réacteur

Le modèle de réacteur est le cœur de la nature asynchrone de `node.js` A permis au système d'être implémenté en tant que processus à thread unique avec une série de générateurs d'événements et de gestionnaires d'événements, à l'aide d'une boucle d'événements exécutée en continu.

Le moteur d'E / S non bloquant de Node.js - libuv -

Le modèle d'observateur (EventEmitter) maintient une liste des personnes à charge / observateurs et les notifie

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

var ringBell = function ringBell()
{
  console.log('tring tring tring');
}
eventEmitter.on('doorOpen', ringBell);

eventEmitter.emit('doorOpen');
```

Lire Node.js Conception fondamentale en ligne: https://riptutorial.com/fr/node-js/topic/6274/node-js-conception-fondamentale

Chapitre 79: Node.JS et MongoDB.

Remarques

Ce sont les opérations CRUD de base pour utiliser mongo db avec nodejs.

Question: Y a-t-il d'autres façons de faire ce qui est fait ici?

Réponse: Oui, il existe de nombreuses façons de procéder.

Question: Est-ce que l'utilisation de mangouste est nécessaire?

Réponse: Non. Il y a d'autres forfaits disponibles qui peuvent vous aider.

Question: Où puis-je obtenir une documentation complète de la mangouste?

Réponse: cliquez ici

Examples

Connexion à une base de données

Pour se connecter à une base de données mongo depuis une application de nœud, nous avons besoin de mangouste.

Installer Mongoose Accédez à la base de votre application et installez mongoose par

```
npm install mongoose
```

Ensuite, nous nous connectons à la base de données.

```
var mongoose = require('mongoose');

//connect to the test database running on default mongod port of localhost
mongoose.connect('mongodb://localhost/test');



//Connecting with custom credentials
mongoose.connect('mongodb://USER:PASSWORD@HOST:PORT/DATABASE');


//Using Pool Size to define the number of connections opening
//Also you can use a call back function for error handling
mongoose.connect('mongodb://localhost:27017/consumers',
                 {server: { poolSize: 50 }},
                 function(err) {
                    if(err) {
                        console.log('error in this')
                        console.log(err);
```

```
                    // Do whatever to handle the error
                } else {
                    console.log('Connected to the database');
                }
            });
```

Créer une nouvelle collection

Avec Mongoose, tout est dérivé d'un schéma. Permet de créer un schéma.

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var AutoSchema = new Schema({
    name : String,
    countOf: Number,
});
// defining the document structure

// by default the collection created in the db would be the first parameter we use (or the
plural of it)
module.exports = mongoose.model('Auto', AutoSchema);

// we can over write it and define the collection name by specifying that in the third
parameters.
module.exports = mongoose.model('Auto', AutoSchema, 'collectionName');


// We can also define methods in the models.
AutoSchema.methods.speak = function () {
  var greeting = this.name
    ? "Hello this is  " + this.name+ " and I have counts of "+ this.countOf
    : "I don't have a name";
  console.log(greeting);
}
mongoose.model('Auto', AutoSchema, 'collectionName');
```

Rappelez-vous que les méthodes doivent être ajoutées au schéma avant de le compiler avec mongoose.model () comme ci-dessus.

Insérer des documents

Pour insérer un nouveau document dans la collection, nous créons un objet du schéma.

```
var Auto = require('models/auto')
var autoObj = new Auto({
    name: "NewName",
    countOf: 10
});
```

Nous sauvegardons comme ci-dessous

```
autoObj.save(function(err, insertedAuto) {
    if (err) return console.error(err);
```

```
    insertedAuto.speak();
    // output: Hello this is NewName and I have counts of 10
});
```

Cela va insérer un nouveau document dans la collection

En train de lire

La lecture des données de la collection est très facile. Obtenir toutes les données de la collection.

```
var Auto = require('models/auto')
Auto.find({}, function (err, autos) {
      if (err) return console.error(err);
       // will return a json array of all the documents in the collection
      console.log(autos);
})
```

Lecture de données avec une condition

```
Auto.find({countOf: {$gte: 5}}, function (err, autos) {
      if (err) return console.error(err);
       // will return a json array of all the documents in the collection whose count is
greater than 5
      console.log(autos);
})
```

Vous pouvez également spécifier le second paramètre comme objet de tous les champs dont vous avez besoin

```
Auto.find({},{name:1}, function (err, autos) {
      if (err) return console.error(err);
       // will return a json array of name field of all the documents in the collection
      console.log(autos);
})
```

Trouver un document dans une collection.

```
Auto.findOne({name:"newName"}, function (err, auto) {
      if (err) return console.error(err);
     //will return the first object of the document whose name is "newName"
      console.log(auto);
})
```

Trouver un document dans une collection par identifiant.

```
Auto.findById(123, function (err, auto) {
      if (err) return console.error(err);
     //will return the first json object of the document whose id is 123
      console.log(auto);
})
```

Mise à jour

Pour mettre à jour des collections et des documents, nous pouvons utiliser l'une des méthodes suivantes:

Les méthodes

- mettre à jour()
- updateOne ()
- updateMany ()
- replacene un ()

Mettre à jour()

La méthode update () modifie un ou plusieurs documents (paramètres de mise à jour)

```
db.lights.update(
   { room: "Bedroom" },
   { status: "On" }
)
```

Cette opération recherche la collection 'lights' pour un document où `room` est **Bedroom** *(1er paramètre)* . Il met ensuite à jour la propriété d' `status` documents correspondants sur **On** *(2nd paramètre)* et renvoie un objet WriteResult qui ressemble à ceci:

```
{ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 }
```

UpdateOne

La méthode UpdateOne () modifie UN document (paramètres de mise à jour)

```
db.countries.update(
   { country: "Sweden" },
   { capital: "Stockholm" }
)
```

Cette opération recherche la collection «pays» pour un document dont le `country` est la **Suède** *(1er paramètre)* . Il met ensuite à jour le `capital` propriété des documents correspondants à **Stockholm** *(2ème paramètre)* et renvoie un objet WriteResult qui ressemble à ceci:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

UpdateMany

La méthode UpdateMany () modifie les documents multiples (paramètres de mise à jour)

```
db.food.updateMany(
   { sold: { $lt: 10 } },
   { $set: { sold: 55 } }
)
```

Cette opération met à jour tous les documents *(dans une collection «alimentaire»)* où le produit `sold` est **inférieur à 10** * (1er paramètre) en définissant le produit `sold` à **55** . Il retourne ensuite un objet WriteResult qui ressemble à ceci:

```
{ "acknowledged" : true, "matchedCount" : a, "modifiedCount" : b }
```

a = Nombre de documents correspondants
b = Nombre de documents modifiés

ReplaceOne

Remplace le premier document correspondant (document de remplacement)

Cet exemple de collection appelé **Pays** contient 3 documents:

```
{ "_id" : 1, "country" : "Sweden" }
{ "_id" : 2, "country" : "Norway" }
{ "_id" : 3, "country" : "Spain" }
```

L'opération suivante remplace le document `{ country: "Spain" }` par le document `{ country: "Finland" }`

```
db.countries.replaceOne(
   { country: "Spain" },
   { country: "Finland" }
)
```

Et revient:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

L'exemple de **pays de** collecte contient maintenant:

```
{ "_id" : 1, "country" : "Sweden" }
{ "_id" : 2, "country" : "Norway" }
{ "_id" : 3, "country" : "Finland" }
```

Effacer

La suppression de documents d'une collection en mangouste se fait de la manière suivante.

```
Auto.remove({_id:123}, function(err, result){
    if (err) return console.error(err);
    console.log(result); // this will specify the mongo default delete result.
});
```

Lire Node.JS et MongoDB. en ligne: https://riptutorial.com/fr/node-js/topic/7505/node-js-et-mongodb-

Chapitre 80: Node.js Gestion des erreurs

Introduction

Nous allons apprendre à créer des objets Error et à lancer et gérer des erreurs dans Node.js

Modifications futures liées aux meilleures pratiques en matière de gestion des erreurs.

Examples

Création d'un objet d'erreur

nouvelle erreur (message)

Crée un nouvel objet d'erreur, où le `message` valeur est défini sur la propriété `message` de l'objet créé. En général, les arguments du `message` sont transmis au constructeur en tant que chaîne. Toutefois, si l'argument de `message` est object et non une chaîne, le constructeur Error appelle la méthode `.toString()` de l'objet passé et définit cette valeur sur la propriété `message` de l'objet d'erreur créé.

```
var err = new Error("The error message");
console.log(err.message); //prints: The error message
console.log(err);
//output
//Error: The error message
//    at ...
```

Chaque objet d'erreur a une trace de pile. La trace de pile contient les informations du message d'erreur et indique où l'erreur s'est produite (la sortie ci-dessus montre la pile d'erreur). Une fois l'objet d'erreur créé, le système capture la trace de la pile de l'erreur sur la ligne en cours. Pour obtenir la trace de la pile, utilisez la propriété stack de tout objet d'erreur créé. En dessous de deux lignes sont identiques:

```
console.log(err);
console.log(err.stack);
```

Erreur de lancer

L'erreur de lancer signifie une exception si une exception n'est pas gérée, le serveur de noeud se bloque.

La ligne suivante génère une erreur:

```
throw new Error("Some error occurred");
```

ou

```
var err = new Error("Some error occurred");
throw err;
```

ou

```
throw "Some error occurred";
```

Le dernier exemple (lancer des chaînes) n'est pas une bonne pratique et n'est pas recommandé (lancez toujours des erreurs qui sont des instances d'objet Error).

Notez que si vous `throw` une erreur dans votre système, le système se bloquera sur cette ligne (s'il n'y a pas de gestionnaire d'exceptions), aucun code ne sera exécuté après cette ligne.

```
var a = 5;
var err = new Error("Some error message");
throw err; //this will print the error stack and node server will stop
a++; //this line will never be executed
console.log(a); //and this one also
```

Mais dans cet exemple:

```
var a = 5;
var err = new Error("Some error message");
console.log(err); //this will print the error stack
a++;
console.log(a); //this line will be executed and will print 6
```

essayer ... attraper un bloc

try ... catch block est pour gérer les exceptions, mémoriser l'exception signifie que l'erreur renvoyée n'est pas l'erreur.

```
try {
    var a = 1;
    b++; //this will cause an error because be is undefined
    console.log(b); //this line will not be executed
} catch (error) {
    console.log(error); //here we handle the error caused in the try block
}
```

Dans le bloc `try` , `b++` provoque une erreur et cette erreur transmise au bloc `catch` qui peut être manipulé ou même peut être renvoyée à la même erreur dans le bloc catch ou faire une petite modification, puis lancer. Voyons l'exemple suivant.

```
try {
    var a = 1;
    b++;
    console.log(b);
} catch (error) {
    error.message = "b variable is undefined, so the undefined can't be incremented"
    throw error;
}
```

Dans l'exemple ci-dessus, nous avons modifié la propriété `message` de l'objet `error` , puis lancé l' `error` modifiée.

Vous pouvez par toute erreur dans votre bloc try et le gérer dans le bloc catch:

```
try {
    var a = 1;
    throw new Error("Some error message");
    console.log(a); //this line will not be executed;
} catch (error) {
    console.log(error); //will be the above thrown error
}
```

Lire Node.js Gestion des erreurs en ligne: https://riptutorial.com/fr/node-js/topic/8590/node-js-gestion-des-erreurs

Chapitre 81: Node.js v6 Nouvelles fonctionnalités et améliorations

Introduction

Le nœud 6 devenant la nouvelle version LTS du nœud. Nous pouvons voir un certain nombre d'améliorations apportées au langage grâce aux nouvelles normes ES6. Nous allons passer en revue certaines des nouvelles fonctionnalités introduites et des exemples de leur mise en œuvre.

Examples

Paramètres de fonction par défaut

```
function addTwo(a, b = 2) {
    return a + b;
}

addTwo(3) // Returns the result 5
```

Avec l'ajout de paramètres de fonction par défaut, vous pouvez désormais rendre les arguments facultatifs et leur donner une valeur par défaut.

Paramètres de repos

```
function argumentLength(...args) {
    return args.length;
}

argumentLength(5) // returns 1
argumentLength(5, 3) //returns 2
argumentLength(5, 3, 6) //returns 3
```

En préfaçant le dernier argument de votre fonction avec ... tous les arguments passés à la fonction sont lus comme un tableau. Dans cet exemple, nous obtenons plusieurs arguments et obtenons la longueur du tableau créé à partir de ces arguments.

Opérateur d'épandage

```
function myFunction(x, y, z) { }
var args = [0, 1, 2];
myFunction(...args);
```

La syntaxe de propagation permet à une expression d'être développée dans des endroits où plusieurs arguments (pour les appels de fonction) ou plusieurs éléments (pour les littéraux de tableau) ou plusieurs variables sont attendus. Tout comme les paramètres de repos, préférez

simplement votre tableau avec . . .

Fonctions de flèche

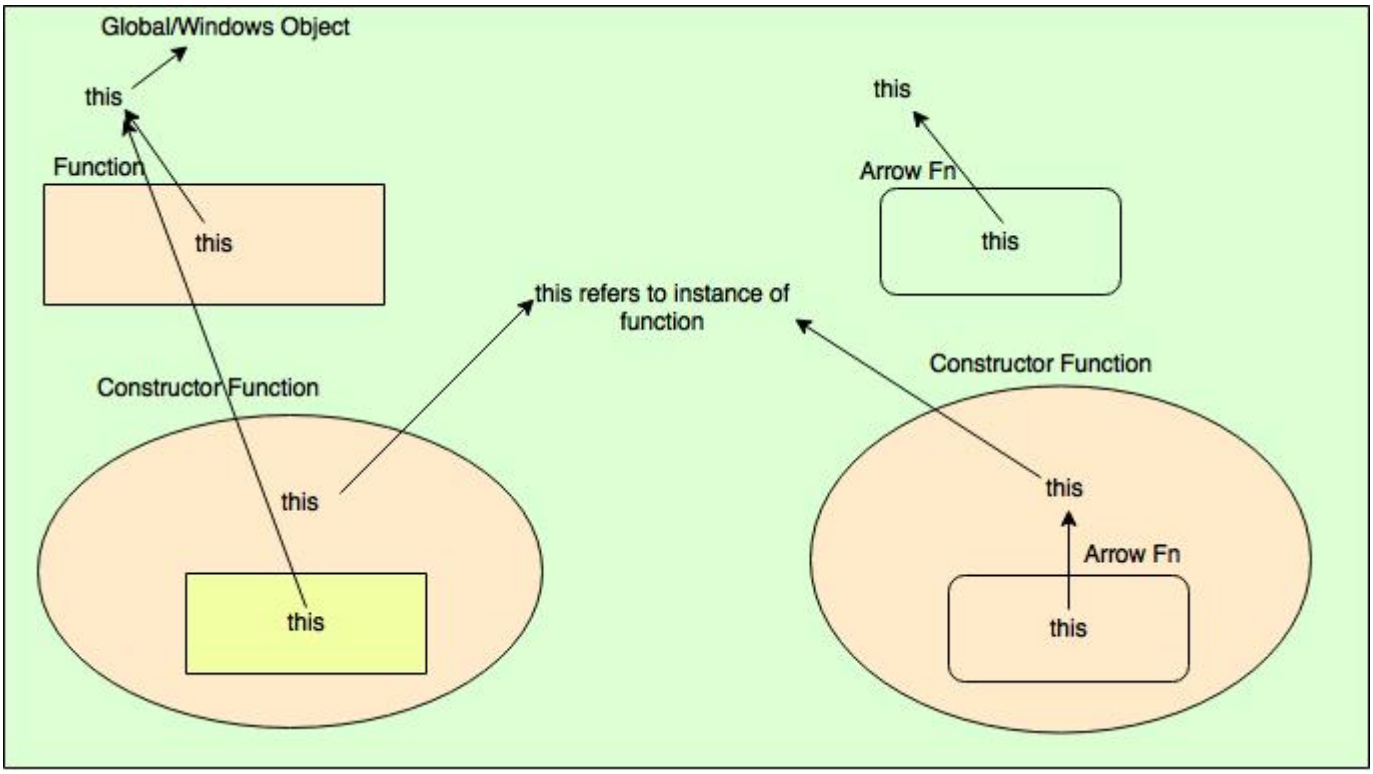
La fonction de flèche est la nouvelle façon de définir une fonction dans ECMAScript 6.

```
// traditional way of declaring and defining function
var sum = function(a,b)
{
    return a+b;
}

// Arrow Function
let sum = (a, b)=> a+b;

//Function defination using multiple lines
let checkIfEven = (a) => {
    if( a % 2 == 0 )
        return true;
    else
        return false;
}
```

"this" in Arrow Function

Cette fonction in fait référence à un objet d'instance utilisé pour appeler cette fonction mais ***cette*** fonction en flèche est égale à *celle* de la fonction dans laquelle la fonction de flèche est définie.

Comprenons en utilisant le diagramme



Comprendre à l'aide d'exemples.

```
var normalFn = function(){
   console.log(this) // refers to global/window object.
}

var arrowFn = () => console.log(this); // refers to window or global object as function is
defined in scope of global/window object

var service = {

    constructorFn : function(){

        console.log(this); //  refers to service as service object used to call method.

        var nestedFn = function(){
            console.log(this); // refers window or global object because no instance object
was used to call this method.
        }
        nestedFn();
    },

    arrowFn : function(){
        console.log(this); // refers to service as service object was used to call method.
        let fn = () => console.log(this); // refers to service object as arrow function
defined in function which is called using instance object.
        fn();
    }
}

// calling defined functions
constructorFn();
arrowFn();
service.constructorFn();
service.arrowFn();
```

Dans la fonction flèche, *il* s'agit de la portée lexicale qui est la portée de la fonction où la fonction flèche est définie.
Le premier exemple est la manière traditionnelle de définir les fonctions et, par conséquent, *cela* fait référence à *l'* objet *global / window* .
Dans le deuxième exemple, *ceci* est utilisé à l'intérieur de la fonction flèche, *c'est-à-dire qu'il* fait référence à la portée où il est défini (qui est Windows ou objet global). Dans le troisième exemple, *il* s'agit d'un objet de service, car l'objet de service est utilisé pour appeler la fonction.
Dans le quatrième exemple, la fonction de flèche dans défini et appelé à partir de la fonction dont la portée est le *service* , par conséquent, il imprime *un* objet de *service* .

Remarque: - L'objet global est imprimé dans Node.Js et l'objet Windows dans le navigateur.

Lire Node.js v6 Nouvelles fonctionnalités et améliorations en ligne: https://riptutorial.com/fr/node-js/topic/8593/node-js-v6-nouvelles-fonctionnalites-et-ameliorations

Chapitre 82: NodeJS avec Redis

Remarques

Nous avons couvert les opérations de base et les plus couramment utilisées dans node_redis. Vous pouvez utiliser ce module pour exploiter toute la puissance de Redis et créer des applications Node.js vraiment sophistiquées. Vous pouvez créer beaucoup de choses intéressantes avec cette bibliothèque, par exemple une couche de mise en cache puissante, un système de messagerie Pub / Sub puissant, etc. Pour en savoir plus sur la bibliothèque, consultez leur documentation .

Examples

Commencer

Comme vous l'avez peut-être deviné, node_redis est le client Redis pour Node.js. Vous pouvez l'installer via npm en utilisant la commande suivante.

```
npm install redis
```

Une fois que vous avez installé le module node_redis, vous êtes prêt à partir. Créons un fichier simple, app.js, et voyons comment se connecter avec Redis à partir de Node.js.

app.js

```
var redis = require('redis');
client = redis.createClient(); //creates a new client
```

Par défaut, redis.createClient () utilisera 127.0.0.1 et 6379 respectivement comme nom d'hôte et port. Si vous avez un hôte / port différent, vous pouvez les fournir comme suit:

```
var client = redis.createClient(port, host);
```

Maintenant, vous pouvez effectuer une action une fois la connexion établie. En gros, il vous suffit d'écouter les événements de connexion comme indiqué ci-dessous.

```
client.on('connect', function() {
    console.log('connected');
});
```

Ainsi, l'extrait suivant va dans app.js:

```
var redis = require('redis');
var client = redis.createClient();

client.on('connect', function() {
```

```
    console.log('connected');
});
```

Maintenant, tapez application de noeud dans le terminal pour exécuter l'application. Assurez-vous que votre serveur Redis est opérationnel avant d'exécuter cet extrait de code.

Stocker des paires clé-valeur

Maintenant que vous savez comment vous connecter à Redis à partir de Node.js, voyons comment stocker des paires clé-valeur dans le stockage Redis.

Stockage de chaînes

Toutes les commandes Redis sont exposées en tant que fonctions différentes sur l'objet client. Pour stocker une chaîne simple, utilisez la syntaxe suivante:

```
client.set('framework', 'AngularJS');
```

Ou

```
client.set(['framework', 'AngularJS']);
```

Les extraits ci-dessus stockent une chaîne simple AngularJS dans le cadre de la clé. Vous devez noter que les deux extraits de code font la même chose. La seule différence est que le premier transmet un nombre variable d'arguments tandis que le dernier transmet un tableau args à la fonction `client.set()` . Vous pouvez également transmettre un rappel facultatif pour obtenir une notification lorsque l'opération est terminée:

```
client.set('framework', 'AngularJS', function(err, reply) {
  console.log(reply);
});
```

Si l'opération échoue pour une raison quelconque, l'argument `err` du rappel représente l'erreur. Pour récupérer la valeur de la clé, procédez comme suit:

```
client.get('framework', function(err, reply) {
    console.log(reply);
});
```

`client.get()` vous permet de récupérer une clé stockée dans Redis. La valeur de la clé est accessible via la réponse d'argument de rappel. Si la clé n'existe pas, la valeur de la réponse sera vide.

Stockage de hachage

Plusieurs fois, stocker des valeurs simples ne résoudra pas votre problème. Vous devez stocker les hashes (objets) dans Redis. Pour cela, vous pouvez utiliser la fonction `hmset()` comme suit:

```
client.hmset('frameworks', 'javascript', 'AngularJS', 'css', 'Bootstrap', 'node', 'Express');

client.hgetall('frameworks', function(err, object) {
    console.log(object);
});
```

L'extrait de code ci-dessus stocke un hash dans Redis qui associe chaque technologie à sa structure. Le premier argument de `hmset()` est le nom de la clé. Les arguments suivants représentent des paires clé-valeur. De même, `hgetall()` est utilisé pour récupérer la valeur de la clé. Si la clé est trouvée, le deuxième argument du rappel contiendra la valeur qui est un objet.

Notez que Redis ne prend pas en charge les objets imbriqués. Toutes les valeurs de propriété de l'objet seront forcées dans des chaînes avant d'être stockées. Vous pouvez également utiliser la syntaxe suivante pour stocker des objets dans Redis:

```
client.hmset('frameworks', {
    'javascript': 'AngularJS',
    'css': 'Bootstrap',
    'node': 'Express'
});
```

Un rappel facultatif peut également être passé pour savoir quand l'opération est terminée.

Toutes les fonctions (commandes) peuvent être appelées avec des équivalents majuscules / minuscules. Par exemple, `client.hmset()` et `client.HMSET()` sont identiques. Stockage des listes

Si vous souhaitez stocker une liste d'éléments, vous pouvez utiliser les listes Redis. Pour stocker une liste, utilisez la syntaxe suivante:

```
client.rpush(['frameworks', 'angularjs', 'backbone'], function(err, reply) {
    console.log(reply); //prints 2
});
```

L'extrait de code ci-dessus crée une liste appelée frameworks et y insère deux éléments. La longueur de la liste est maintenant de deux. Comme vous pouvez le voir, j'ai passé un tableau `args` pour `rpush` . Le premier élément du tableau représente le nom de la clé tandis que le reste représente les éléments de la liste. Vous pouvez également utiliser `lpush()` au lieu de `rpush()` pour pousser les éléments vers la gauche.

Pour récupérer les éléments de la liste, vous pouvez utiliser la fonction `lrange()` comme suit:

```
client.lrange('frameworks', 0, -1, function(err, reply) {
    console.log(reply); // ['angularjs', 'backbone']
});
```

Notez simplement que vous obtenez tous les éléments de la liste en passant -1 comme troisième argument à `lrange()` . Si vous voulez un sous-ensemble de la liste, vous devez passer l'index final ici.

Jeux de stockage

Les ensembles sont similaires aux listes, mais la différence est qu'ils ne permettent pas les doublons. Donc, si vous ne voulez pas d'éléments en double dans votre liste, vous pouvez utiliser un ensemble. Voici comment nous pouvons modifier notre extrait de code précédent pour utiliser un ensemble au lieu de la liste.

```
client.sadd(['tags', 'angularjs', 'backbonejs', 'emberjs'], function(err, reply) {
    console.log(reply); // 3
});
```

Comme vous pouvez le voir, la fonction `sadd()` crée un nouvel ensemble avec les éléments spécifiés. Ici, la longueur de l'ensemble est de trois. Pour récupérer les membres de l'ensemble, utilisez la fonction `smembers()` comme suit:

```
client.smembers('tags', function(err, reply) {
    console.log(reply);
});
```

Cet extrait récupère tous les membres de l'ensemble. Notez simplement que la commande n'est pas conservée lors de la récupération des membres.

Ceci était une liste des structures de données les plus importantes trouvées dans chaque application propulsée par Redis. Mis à part les chaînes, les listes, les ensembles et les hachages, vous pouvez stocker des ensembles triés, hyperLogLogs et plus encore dans Redis. Si vous souhaitez une liste complète des commandes et des structures de données, consultez la documentation officielle Redis. N'oubliez pas que presque toutes les commandes Redis sont exposées sur l'objet client offert par le module node_redis.

Quelques opérations plus importantes prises en charge par node_redis.

Vérification de l'existence des clés

Parfois, vous devrez peut-être vérifier si une clé existe déjà et procéder en conséquence. Pour ce faire, vous pouvez utiliser la fonction `exists()` comme indiqué ci-dessous:

```
client.exists('key', function(err, reply) {
    if (reply === 1) {
        console.log('exists');
    } else {
        console.log('doesn\'t exist');
    }
});
```

Suppression et expiration des clés

Parfois, vous devrez effacer certaines clés et les réinitialiser. Pour effacer les touches, vous pouvez utiliser la commande del comme indiqué ci-dessous:

```
client.del('frameworks', function(err, reply) {
    console.log(reply);
});
```

Vous pouvez également donner une heure d'expiration à une clé existante comme suit:

```
client.set('key1', 'val1');
client.expire('key1', 30);
```

L'extrait de code ci-dessus attribue un délai d'expiration de 30 secondes à la touche clé1.

Incrémenter et Décrémenter

Redis prend également en charge l'incrémentation et la décrémentation des clés. Pour incrémenter une touche, utilisez la fonction `incr()` comme indiqué ci-dessous:

```
client.set('key1', 10, function() {
    client.incr('key1', function(err, reply) {
        console.log(reply); // 11
    });
});
```

La fonction `incr()` incrémente une valeur de clé de 1. Si vous devez incrémenter d'une valeur différente, vous pouvez utiliser la fonction `incrby()` . De même, pour décrémenter une touche, vous pouvez utiliser les fonctions comme `decr()` et `decrby()` .

Lire NodeJS avec Redis en ligne: https://riptutorial.com/fr/node-js/topic/7107/nodejs-avec-redis

Chapitre 83: NodeJs Routing

Introduction

Comment configurer le serveur Web Express de base sous le noeud js et Explorer le routeur Express.

Remarques

Enfin, en utilisant Express Router, vous pouvez utiliser la fonction de routage dans votre application et la mettre en œuvre facilement.

Examples

Routage Express Web Server

Création d'Express Web Server

Le serveur Express est très pratique et traverse de nombreux utilisateurs et communautés. Il devient populaire.

Permet de créer un serveur Express. Pour la gestion des packages et la flexibilité pour la dépendance Nous utiliserons NPM (Node Package Manager).

1. Accédez au répertoire du projet et créez le fichier package.json. ***package.json*** {"name": "expressRouter", "version": "0.0.1", "scripts": {"start": "node Server.js"}, "dependencies": {"express": "^ 4.12.3 "}}

2. Enregistrez le fichier et installez la dépendance express à l'aide de la commande suivante *npm install* . Cela créera node_modules dans votre répertoire de projet avec les dépendances requises.

3. Créons Express Web Server. Accédez au répertoire du projet et créez le fichier server.js. ***server.js***

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

// Création de l'objet Routeur ()

var router = express.Router ();

// Fournit tous les itinéraires ici, c'est pour la page d'accueil.

```
router.get("/",function(req,res){
res.json({"message" : "Hello World"});
```

})

app.use ("/ api", routeur);

// écoute ce port

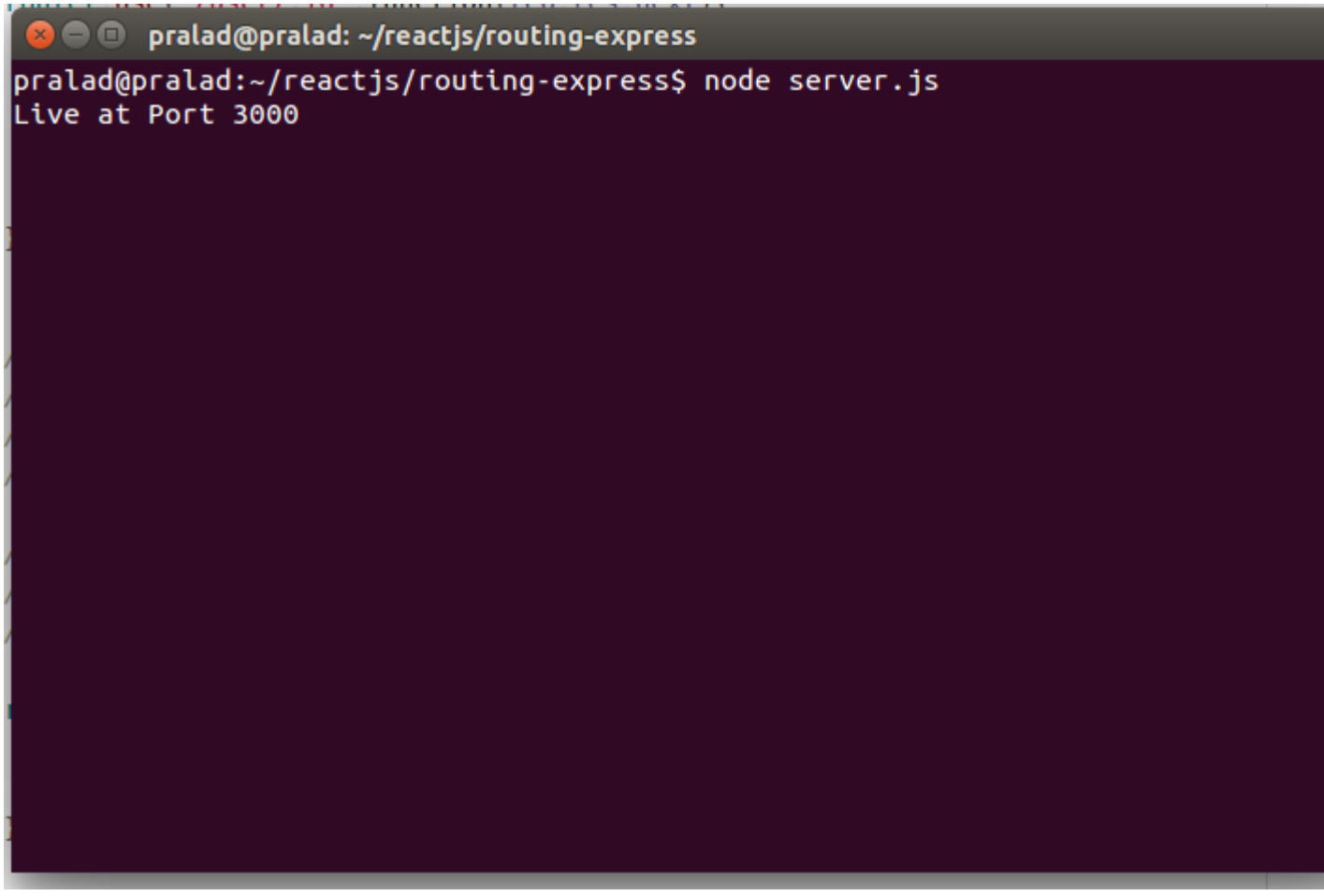
app.listen (3000, function () {console.log ("Live at Port 3000");});

```
For more detail on setting node server you can see [here][1].
```

4. Exécutez le serveur en tapant la commande suivante.

   node server.js

   Si le serveur fonctionne correctement, vous obtiendrez quelque chose comme ceci.



   .

5. Maintenant, allez au navigateur ou au postier et faites une demande

   http: // localhost: 3000 / api /

   La sortie sera

.

C'est tout, la base du routage Express.

Maintenant, gérons le GET, le POST, etc.

Changer le fichier yous server.js comme

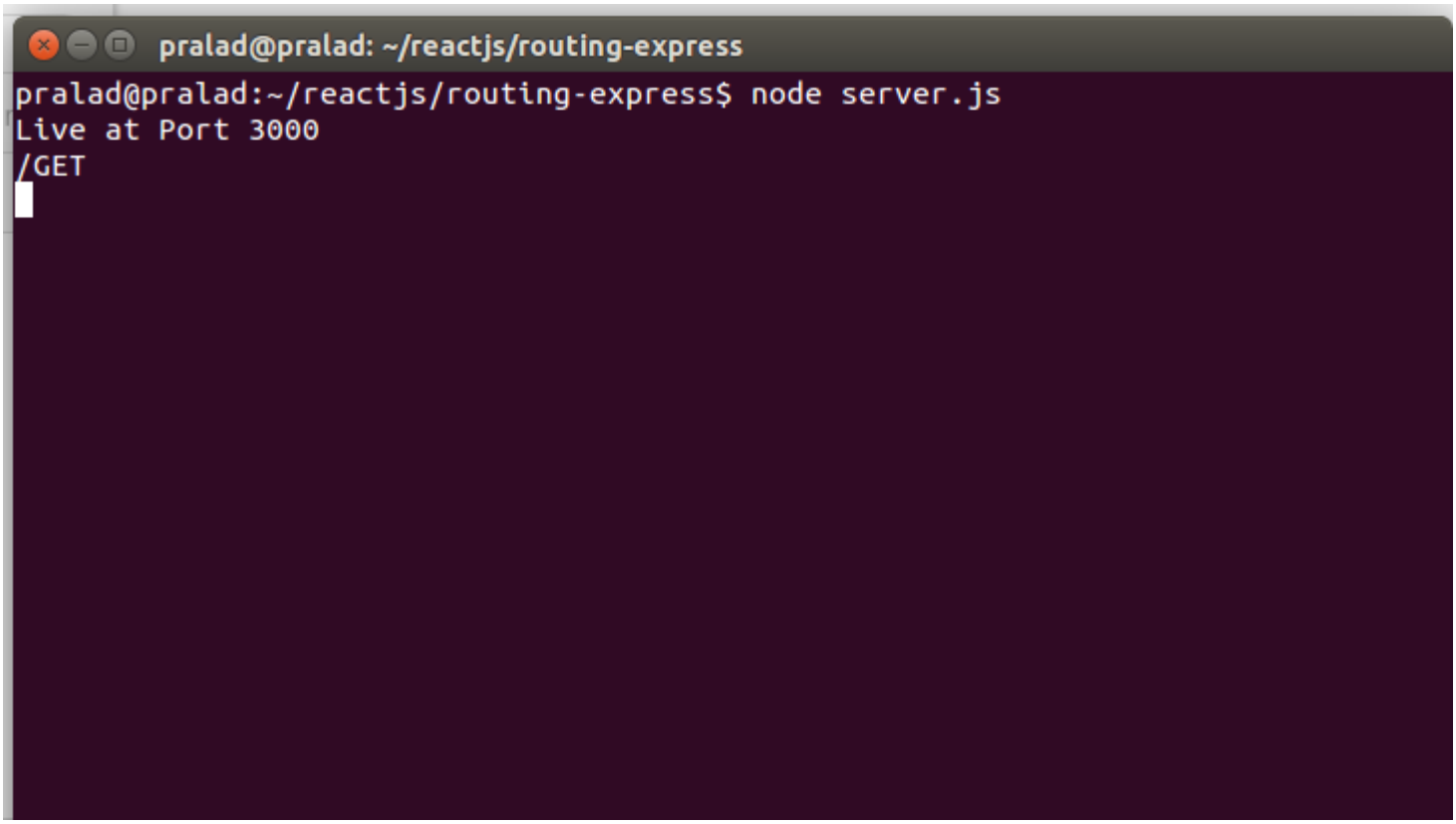
```
var express = require("express");
var app = express();

//Creating Router() object

var router = express.Router();

// Router middleware, mentioned it before defining routes.

router.use(function(req,res,next) {
  console.log("/" + req.method);
  next();
});

// Provide all routes here, this is for Home page.

router.get("/",function(req,res){
  res.json({"message" : "Hello World"});
});


app.use("/api",router);


app.listen(3000,function(){
  console.log("Live at Port 3000");
```

```
});
```

Maintenant, si vous redémarrez le serveur et faites la demande à

```
http://localhost:3000/api/
```

Vous verrez quelque chose comme

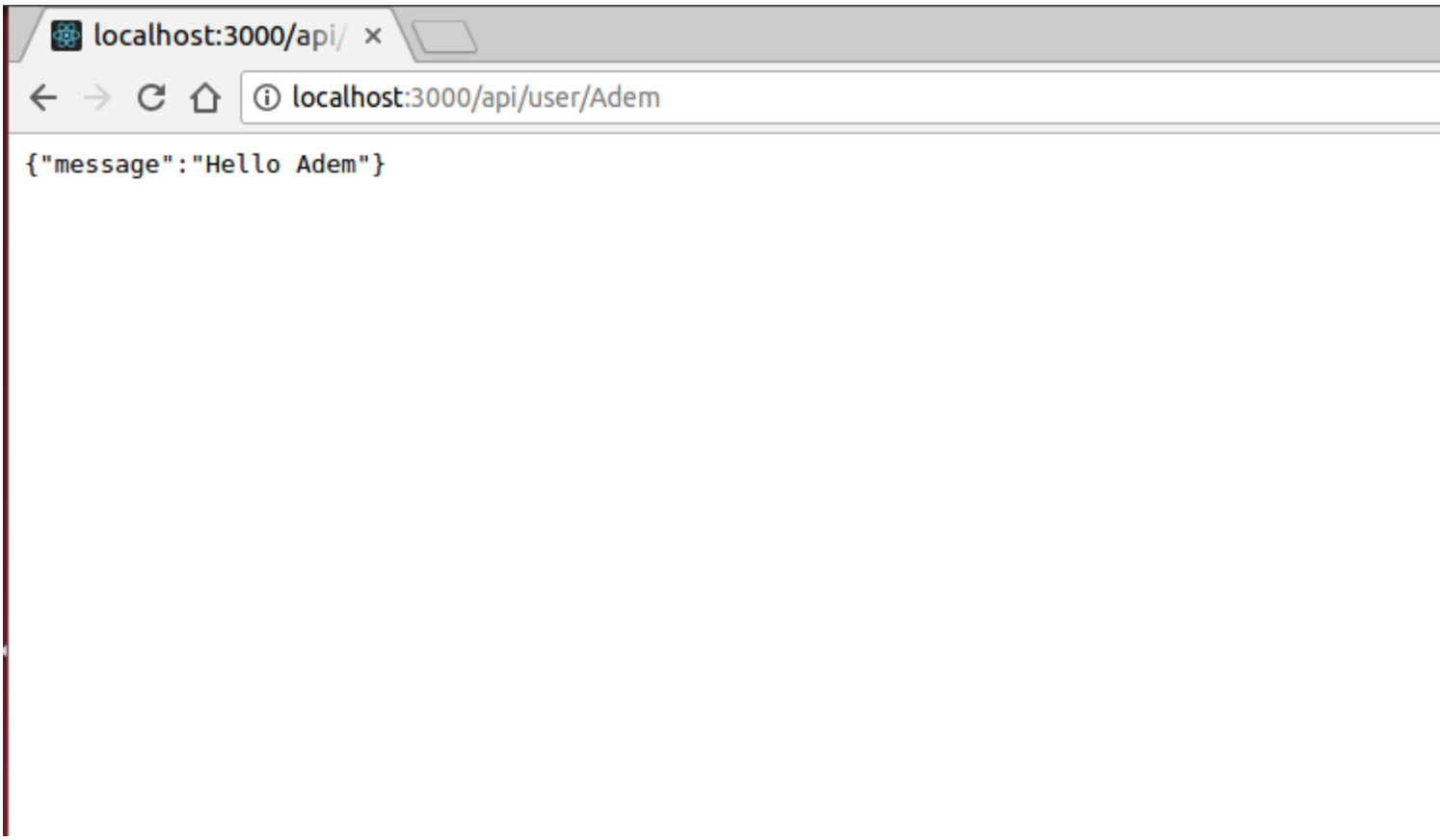


Accès au paramètre dans le routage

Vous pouvez également accéder au paramètre depuis l'URL, comme *http://example.com/api/:name/* . Donc, le paramètre name peut être un accès. Ajoutez le code suivant dans votre server.js

```
router.get("/user/:id",function(req,res){
  res.json({"message" : "Hello "+req.params.id});
});
```

Maintenant redémarrez le serveur et allez à [http: // localhost: 3000 / api / user / Adem] [4] , le résultat sera comme



.

Lire NodeJs Routing en ligne: https://riptutorial.com/fr/node-js/topic/9846/nodejs-routing

Chapitre 84: Nœud serveur sans framework

Remarques

Bien que Node dispose de nombreux frameworks pour vous aider à faire fonctionner votre serveur, principalement:

Express : le framework le plus utilisé

Total : Le framework UN-UN-EN-UN-UN, qui a tout et ne dépend d'aucun autre framework ou module.

Mais, il n'y a toujours pas de taille unique, donc le développeur peut avoir besoin de créer son propre serveur, sans aucune autre dépendance.

Si l'application accessible via un serveur externe, CORS, pouvait constituer un problème, un code pour l'éviter avait été fourni.

Examples

Serveur de noeud sans structure

```
var http = require('http');
var fs = require('fs');
var path = require('path');

http.createServer(function (request, response) {
console.log('request ', request.url);

var filePath = '.' + request.url;
if (filePath == './')
    filePath = './index.html';

var extname = String(path.extname(filePath)).toLowerCase();
var contentType = 'text/html';
var mimeTypes = {
    '.html': 'text/html',
    '.js': 'text/javascript',
    '.css': 'text/css',
    '.json': 'application/json',
    '.png': 'image/png',
    '.jpg': 'image/jpg',
    '.gif': 'image/gif',
    '.wav': 'audio/wav',
    '.mp4': 'video/mp4',
    '.woff': 'application/font-woff',
    '.ttf': 'applilcation/font-ttf',
    '.eot': 'application/vnd.ms-fontobject',
    '.otf': 'application/font-otf',
    '.svg': 'application/image/svg+xml'
};
```

```
contentType = mimeTypes[extname] || 'application/octect-stream';

fs.readFile(filePath, function(error, content) {
    if (error) {
        if(error.code == 'ENOENT'){
            fs.readFile('./404.html', function(error, content) {
                response.writeHead(200, { 'Content-Type': contentType });
                response.end(content, 'utf-8');
            });
        }
        else {
            response.writeHead(500);
            response.end('Sorry, check with the site admin for error: '+error.code+' ..\n');
            response.end();
        }
    }
    else {
        response.writeHead(200, { 'Content-Type': contentType });
        response.end(content, 'utf-8');
    }
 });

}).listen(8125);
console.log('Server running at http://127.0.0.1:8125/');
```

Surmonter les problèmes de la SCRO

```
// Website you wish to allow to connect to
response.setHeader('Access-Control-Allow-Origin', '*');

// Request methods you wish to allow
response.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE');

// Request headers you wish to allow
response.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');

// Set to true if you need the website to include cookies in the requests sent
// to the API (e.g. in case you use sessions)
response.setHeader('Access-Control-Allow-Credentials', true);
```

Lire Nœud serveur sans framework en ligne: https://riptutorial.com/fr/node-js/topic/5910/noud-serveur-sans-framework

Chapitre 85: Notifications push

Introduction

Donc, si vous voulez faire une notification d'application Web, je vous suggère d'utiliser Push.js ou SoneSignal pour l'application Web / mobile.

Push est le moyen le plus rapide de se familiariser avec les notifications Javascript. Nouvel ajout à la spécification officielle, l'API de notification permet aux navigateurs modernes tels que Chrome, Safari, Firefox et IE 9+ d'envoyer des notifications au bureau d'un utilisateur.

Vous devrez utiliser Socket.io et un framework backend, je vais utiliser Express pour cet exemple.

Paramètres

module / framework	la description
node.js / express	Simple framework back-end pour l'application Node.js, très facile à utiliser et extrêmement puissant
Socket.io	Socket.IO permet une communication en temps réel bidirectionnelle basée sur des événements. Il fonctionne sur toutes les plates-formes, navigateurs ou appareils, en mettant l'accent sur la fiabilité et la rapidité.
Push.js	Le cadre de notification de bureau le plus polyvalent au monde
OneSignal	Juste une autre forme de notifications push pour les appareils Apple
Firebase	Firebase est la plate-forme mobile de Google qui vous aide à développer rapidement des applications de haute qualité et à développer votre activité.

Examples

Notification Web

Tout d'abord, vous devrez installer le module Push.js.

```
$ npm install push.js --save
```

Ou importez-le sur votre application frontale via CDN

```
<script src="./push.min.js"></script> <!-- CDN link -->
```

Une fois que vous avez terminé avec cela, vous devriez être bon pour aller. Voici comment cela devrait ressembler si vous voulez faire une simple notification:

```
Push.create('Hello World!')
```

Je suppose que vous savez comment configurer Socket.io avec votre application. Voici un exemple de code de mon application backend avec express:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);

server.listen(80);

app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});

io.on('connection', function (socket) {

  socket.emit('pushNotification', { success: true, msg: 'hello' });

});
```

Une fois que votre serveur est configuré, vous devriez pouvoir passer à la configuration initiale. Il ne reste plus qu'à importer Socket.io CDN et à ajouter ce code dans mon fichier *index.html* :

```
<script src="../socket.io.js"></script> <!-- CDN link -->
<script>
  var socket = io.connect('http://localhost');
  socket.on('pushNotification', function (data) {
    console.log(data);
    Push.create("Hello world!", {
        body: data.msg, //this should print "hello"
        icon: '/icon.png',
        timeout: 4000,
        onClick: function () {
            window.focus();
            this.close();
        }
    });
  });
</script>
```

Voilà, vous devriez maintenant être en mesure d'afficher votre notification, cela fonctionne également sur tout appareil Android, et si vous souhaitez utiliser la messagerie cloud Firebase , vous pouvez l'utiliser avec ce module, voici le lien de cet exemple écrit par Nick (créateur de Push.js)

Pomme

Gardez à l'esprit que cela ne fonctionnera pas sur les appareils Apple (je ne les ai pas tous testés), mais si vous voulez faire des notifications push, vérifiez le plug- in OneSignal .

Lire Notifications push en ligne: https://riptutorial.com/fr/node-js/topic/10892/notifications-push

Chapitre 86: npm

Introduction

Node Package Manager (npm) offre les deux fonctionnalités principales suivantes: Référentiels en ligne pour les packages / modules node.js qui peuvent être recherchés sur search.nodejs.org. Utilitaire de ligne de commande pour installer les packages Node.js, faire la gestion des versions et la gestion des dépendances des packages Node.js.

Syntaxe

- npm <command> où <command> est l'un des suivants:
  - ajouter un utilisateur
  - adduser
  - apihelp
  - auteur
  - poubelle
  - bogues
  - c
  - cache
  - achèvement
  - config
  - ddp
  - déduplication
  - désapprouver
  - docs
  - modifier
  - explorer
  - FAQ
  - trouver
  - trouver des dupes
  - obtenir
  - Aidez-moi
  - aide-recherche
  - maison
  - je
  - installer
  - Info
  - init
  - isntall
  - problèmes
  - la
  - lien
  - liste

- ll
- ln
- s'identifier
- ls
- dépassé
- propriétaire
- pack
- préfixe
- prune
- publier
- r
- rb
- reconstruire
- retirer
- repo
- redémarrer
- rm
- racine
- script de lancement
- s
- se
- chercher
- ensemble
- montrer
- film rétractable
- étoile
- étoiles
- début
- Arrêtez
- sous-module
- marque
- tester
- tst
- ONU
- désinstaller
- dissocier
- non publié
- unstar
- en haut
- mettre à jour
- v
- version
- vue
- qui suis je

Paramètres

Paramètre	Exemple
accès	`npm publish --access=public`
poubelle	`npm bin -g`
modifier	`npm edit connect`
Aidez-moi	`npm help init`
init	`npm init`
installer	`npm install`
lien	`npm link`
prune	`npm prune`
publier	`npm publish ./`
redémarrer	`npm restart`
début	`npm start`
Arrêtez	`npm start`
mettre à jour	`npm update`
version	`npm version`

Examples

Installer des paquets

introduction

Package est un terme utilisé par npm pour désigner les outils que les développeurs peuvent utiliser pour leurs projets. Cela inclut tout, des bibliothèques et des frameworks tels que jQuery et AngularJS aux exécuteurs de tâches tels que Gulp.js. Les paquets seront fournis dans un dossier généralement appelé `node_modules` , qui contiendra également un fichier `package.json` . Ce fichier contient des informations concernant tous les packages, y compris les dépendances, qui sont des modules supplémentaires nécessaires pour utiliser un package particulier.

Npm utilise la ligne de commande pour installer et gérer les packages, de sorte que les utilisateurs essayant d'utiliser npm doivent être familiarisés avec les commandes de base de leur système d'exploitation: parcourir les répertoires et voir le contenu des répertoires.

Installation de NPM

Notez que pour installer des packages, NPM doit être installé.

La méthode recommandée pour installer NPM consiste à utiliser l'un des programmes d'installation de la page de téléchargement de Node.js. Vous pouvez vérifier si vous avez déjà installé node.js en exécutant la commande `npm -v` ou `npm version` .

Après avoir installé NPM via le programme d'installation de Node.js, assurez-vous de vérifier les mises à jour. Cela est dû au fait que NPM est mis à jour plus fréquemment que le programme d'installation de Node.js. Pour vérifier les mises à jour, exécutez la commande suivante:

```
npm install npm@latest -g
```

Comment installer les paquets

Pour installer un ou plusieurs packages, procédez comme suit:

```
npm install <package-name>
# or
npm i <package-name>...

# e.g. to install lodash and express
npm install lodash express
```

> **Note** : Ceci installera le paquet dans le répertoire dans lequel se trouve la ligne de commande, il est donc important de vérifier si le répertoire approprié a été choisi

Si vous avez déjà un fichier `package.json` dans votre répertoire de travail actuel et que des dépendances y sont définies, alors `npm install` résoudra et installera automatiquement toutes les dépendances répertoriées dans le fichier. Vous pouvez également utiliser la version abrégée de la commande `npm install` : `npm i`

Si vous souhaitez installer une version spécifique d'un package, utilisez:

```
npm install <name>@<version>

# e.g. to install version 4.11.1 of the package lodash
npm install lodash@4.11.1
```

Si vous souhaitez installer une version correspondant à une plage de version spécifique, utilisez:

```
npm install <name>@<version range>

# e.g. to install a version which matches "version >= 4.10.1" and "version < 4.11.1"
# of the package lodash
npm install lodash@">=4.10.1 <4.11.1"
```

Si vous souhaitez installer la dernière version, utilisez:

```
npm install <name>@latest
```

Les commandes ci-dessus rechercheront les paquets dans le dépôt central `npm` à npmjs.com . Si vous ne souhaitez pas installer depuis le registre `npm` , d'autres options sont prises en charge, telles que:

```
# packages distributed as a tarball
npm install <tarball file>
npm install <tarball url>

# packages available locally
npm install <local path>

# packages available as a git repository
npm install <git remote url>

# packages available on GitHub
npm install <username>/<repository>

# packages available as gist (need a package.json)
npm install gist:<gist-id>

# packages from a specific repository
npm install --registry=http://myreg.mycompany.com <package name>

# packages from a related group of packages
# See npm scope
npm install @<scope>/<name>(@<version>)

# Scoping is useful for separating private packages hosted on private registry from
# public ones by setting registry for specific scope
npm config set @mycompany:registry http://myreg.mycompany.com
npm install @mycompany/<package name>
```

Généralement, les modules seront installés localement dans un dossier nommé `node_modules` , qui se trouve dans votre répertoire de travail actuel. C'est le répertoire que `require()` pour charger les modules afin de les mettre à votre disposition.

Si vous avez déjà créé un fichier `package.json` , vous pouvez utiliser l' `--save` (`--save -S`) ou l'une de ses variantes pour ajouter automatiquement le package installé à votre `package.json` tant que dépendance. Si quelqu'un installe votre paquet, `npm` lira automatiquement les dépendances du fichier `package.json` et installera les versions listées. Notez que vous pouvez toujours ajouter et gérer vos dépendances en éditant le fichier ultérieurement. Il est donc recommandé de suivre les dépendances, par exemple en utilisant:

```
npm install --save <name> # Install dependencies
# or
npm install -S <name> # shortcut version --save
# or
npm i -S <name>
```

Pour installer les packages et les enregistrer uniquement s'ils sont nécessaires au

développement, pas pour les exécuter, pas s'ils sont nécessaires à l'exécution de l'application, suivez la commande suivante:

```
npm install --save-dev <name> # Install dependencies for development purposes
# or
npm install -D <name> # shortcut version --save-dev
# or
npm i -D <name>
```

Installation de dépendances

Certains modules fournissent non seulement une bibliothèque à utiliser, mais fournissent également un ou plusieurs fichiers binaires destinés à être utilisés via la ligne de commande. Bien que vous puissiez toujours installer ces packages localement, il est souvent préférable de les installer globalement pour que les outils de ligne de commande puissent être activés. Dans ce cas, `npm` liera automatiquement les fichiers binaires aux chemins appropriés (par exemple `/usr/local/bin/<name>`) afin qu'ils puissent être utilisés à partir de la ligne de commande. Pour installer un package globalement, utilisez:

```
npm install --global <name>
# or
npm install -g <name>
# or
npm i -g <name>

# e.g. to install the grunt command line tool
npm install -g grunt-cli
```

Si vous souhaitez voir une liste de tous les packages installés et de leurs versions associées dans l'espace de travail actuel, utilisez:

```
npm list
npm list <name>
```

L'ajout d'un argument de nom facultatif permet de vérifier la version d'un package spécifique.

Remarque: Si vous rencontrez des problèmes de permission en essayant d'installer un module npm globalement, résistez à la tentation de `sudo npm install -g ...` un `sudo npm install -g ...` pour résoudre le problème. Accorder des scripts tiers pour s'exécuter sur votre système avec des privilèges élevés est dangereux. Le problème de la permission peut signifier que vous avez un problème avec la façon dont `npm` été installé. Si vous êtes intéressé par l'installation de Node dans des environnements utilisateur en bac à sable, vous pouvez essayer d'utiliser nvm .

Si vous disposez d'outils de génération ou d'autres dépendances de développement uniquement (par exemple Grunt), vous ne souhaiterez peut-être pas les regrouper avec l'application que vous déployez. Si tel est le cas, vous voudrez l'avoir comme une dépendance de développement, qui est répertoriée dans le `package.json` sous `devDependencies` . Pour installer un package en tant que

dépendance de développement uniquement, utilisez `--save-dev` (ou `-D`).

```
npm install --save-dev <name> // Install development dependencies which is not included in
production
# or
npm install -D <name>
```

Vous verrez que le paquet est ensuite ajouté aux `devDependencies` de votre `package.json` .

Pour installer des dépendances d'un projet node.js téléchargé / cloné, vous pouvez simplement utiliser

```
npm install
# or
npm i
```

npm lit automatiquement les dépendances à partir de `package.json` et les installe.

MNP derrière un serveur proxy

Si votre accès Internet se fait via un serveur proxy, vous devrez peut-être modifier les commandes npm install qui accèdent aux référentiels distants. npm utilise un fichier de configuration qui peut être mis à jour via la ligne de commande:

```
npm config set
```

Vous pouvez localiser vos paramètres de proxy depuis le panneau de configuration de votre navigateur. Une fois que vous avez obtenu les paramètres de proxy (URL du serveur, port, nom d'utilisateur et mot de passe); vous devez configurer vos configurations npm comme suit.

```
$ npm config set proxy http://<username>:<password>@<proxy-server-url>:<port>
$ npm config set https-proxy http://<username>:<password>@<proxy-server-url>:<port>
```

`username` , `password` , `port` champs de `port` sont facultatifs. Une fois ces paramètres définis, votre `npm install` `npm i -g` , `npm i -g` etc. fonctionnerait correctement.

Portées et référentiels

```
# Set the repository for the scope "myscope"
npm config set @myscope:registry http://registry.corporation.com

# Login at a repository and associate it with the scope "myscope"
npm adduser --registry=http://registry.corporation.com --scope=@myscope

# Install a package "mylib" from the scope "myscope"
npm install @myscope/mylib
```

Si le nom de votre propre package commence par `@myscope` et que la portée "myscope" est

associée à un autre référentiel, `npm publish` plutôt votre package sur ce référentiel.

Vous pouvez également conserver ces paramètres dans un fichier `.npmrc` :

```
@myscope:registry=http://registry.corporation.com
//registry.corporation.com/:_authToken=xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxxxxx
```

Ceci est utile lors de l'automatisation de la construction sur un serveur CI

Désinstallation des paquets

Pour désinstaller un ou plusieurs packages installés localement, utilisez:

```
npm uninstall <package name>
```

La commande de désinstallation pour npm comporte cinq alias pouvant également être utilisés:

```
npm remove <package name>
npm rm <package name>
npm r <package name>

npm unlink <package name>
npm un <package name>
```

Si vous souhaitez supprimer le package du fichier `package.json` dans le cadre de la désinstallation, utilisez l'indicateur `--save` (raccourci: `-S`):

```
npm uninstall --save <package name>
npm uninstall -S <package name>
```

Pour une dépendance de développement, utilisez le drapeau `--save-dev` (raccourci: `-D`):

```
npm uninstall --save-dev <package name>
npm uninstall -D <package name>
```

Pour une dépendance facultative, utilisez l' `--save-optional` (raccourci: `-O`):

```
npm uninstall --save-optional <package name>
npm uninstall -O <package name>
```

Pour les packages installés globalement, utilisez le drapeau `--global` (raccourci: `-g`):

```
npm uninstall -g <package name>
```

Versioning sémantique de base

Avant de publier un package, vous devez le mettre à jour. npm prend en charge la gestion des versions sémantiques , ce qui signifie qu'il existe des versions **correctives, mineures et majeures** .

Par exemple, si votre package est à la version 1.2.3 pour modifier la version, vous devez:

1. `npm version patch` : `npm version patch` => 1.2.4
2. `npm version minor` : `npm version minor` => 1.3.0
3. `npm version major` : `npm version major` => 2.0.0

Vous pouvez également spécifier une version directement avec:

`npm version 3.1.4` => 3.1.4

Lorsque vous définissez une version de package utilisant l'une des commandes npm ci-dessus, npm modifie le champ de version du fichier package.json, le valide et crée également un nouveau tag Git avec la version précédée d'un "v", comme si vous 'ai émis la commande:

```
git tag v3.1.4
```

Contrairement à d'autres gestionnaires de paquetages comme Bower, le registre npm ne repose pas sur la création de balises Git pour chaque version. Mais si vous aimez utiliser les balises, n'oubliez pas de pousser la balise nouvellement créée après avoir heurté la version du paquet:

`git push origin master` (pour pousser le changement vers package.json)

`git push origin v3.1.4` (pour pousser la nouvelle balise)

Ou vous pouvez le faire d'un seul coup avec:

```
git push origin master --tags
```

Configuration d'une configuration de package

Les configurations de package Node.js sont contenues dans un fichier appelé `package.json` que vous pouvez trouver à la racine de chaque projet. Vous pouvez configurer un nouveau fichier de configuration en appelant:

```
npm init
```

Cela va essayer de lire le répertoire de travail actuel pour les informations du référentiel Git (s'il existe) et les variables d'environnement pour essayer de compléter automatiquement certaines des valeurs des espaces réservés pour vous. Sinon, il fournira une boîte de dialogue de saisie pour les options de base.

Si vous souhaitez créer un `package.json` avec les valeurs par défaut, utilisez:

```
npm init --yes
# or
npm init -y
```

Si vous créez un `package.json` pour un projet que vous ne publierez pas en tant que package npm (uniquement dans le but de rassembler vos dépendances), vous pouvez transmettre cette intention dans votre fichier `package.json` :

1. Définissez éventuellement la propriété `private` sur true pour empêcher toute publication accidentelle.
2. Définissez éventuellement la propriété de `license` sur "UNLICENSED" pour refuser aux autres le droit d'utiliser votre package.

Pour installer un paquet et l'enregistrer automatiquement dans votre `package.json` , utilisez:

```
npm install --save <package>
```

Le package et les métadonnées associées (telles que la version du package) apparaîtront dans vos dépendances. Si vous enregistrez en tant que dépendance de développement (en utilisant `--save-dev`), le package apparaîtra plutôt dans vos `devDependencies` .

Avec ce `package.json` bare-bones, vous rencontrerez des messages d'avertissement lors de l'installation ou de la mise à niveau des packages, vous indiquant qu'il vous manque une description et le champ du référentiel. Bien qu'il soit prudent d'ignorer ces messages, vous pouvez les supprimer en ouvrant le package.json dans n'importe quel éditeur de texte et en ajoutant les lignes suivantes à l'objet JSON:

```
[...]
"description": "No description",
"repository": {
  "private": true
},
[...]
```

Publication d'un package

Tout d'abord, assurez-vous d'avoir configuré votre paquet (comme indiqué dans Configuration d'une configuration de paquet). Ensuite, vous devez être connecté à npmjs.

Si vous avez déjà un utilisateur npm

```
npm login
```

Si vous n'avez pas d'utilisateur

```
npm adduser
```

Pour vérifier que votre utilisateur est enregistré dans le client actuel

```
npm config ls
```

Après cela, quand votre paquet est prêt à être publié, utilisez

```
npm publish
```

Et vous avez terminé.

Si vous devez publier une nouvelle version, assurez-vous de mettre à jour la version de votre package, comme indiqué dans la section Gestion des versions sémantiques . Sinon, `npm` ne vous laissera pas publier le paquet.

```
{
    name: "package-name",
    version: "1.0.4"
}
```

Exécution de scripts

Vous pouvez définir des scripts dans votre `package.json` , par exemple:

```
{
  "name": "your-package",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "echo": "echo hello!"
  }
}
```

Pour exécuter le script `echo` , exécutez `npm run echo` partir de la ligne de commande. Les scripts arbitraires, tels que `echo` ci-dessus, doivent être exécutés avec `npm run <script name>` . npm dispose également d'un certain nombre de scripts officiels qu'il exécute à certaines étapes de la vie du paquet (comme la `preinstall`). Voir ici pour la présentation complète de la façon dont npm gère les champs de script.

Les scripts npm sont utilisés le plus souvent pour démarrer un serveur, créer un projet et exécuter des tests. Voici un exemple plus réaliste:

```
  "scripts": {
    "test": "mocha tests",
    "start": "pm2 start index.js"
  }
```

Dans les entrées de `scripts` , les programmes en ligne de commande tels que `mocha` fonctionneront lorsqu'ils seront installés globalement ou localement. Si l'entrée de ligne de commande n'existe pas dans le système PATH, npm vérifie également vos packages installés localement.

Si vos scripts deviennent très longs, ils peuvent être divisés en plusieurs parties, comme ceci:

```
  "scripts": {
    "very-complex-command": "npm run chain-1 && npm run chain-2",
    "chain-1": "webpack",
```

```
    "chain-2": "node app.js"
  }
```

Enlever les paquets superflus

Pour supprimer les paquets superflus (packages installés mais pas dans la liste des dépendances), exécutez la commande suivante:

```
npm prune
```

Pour supprimer tous les paquets de `dev` , ajoutez `--production` flag de production:

```
npm prune --production
```

Plus à ce sujet

Liste des paquets actuellement installés

Pour générer une liste (arborescence) des packages actuellement installés, utilisez

```
npm list
```

ls , **la** et **ll** sont des alias de commande **list** . Les commandes la et ll affichent des informations étendues telles que la description et le référentiel.

Les options

Le format de réponse peut être modifié en passant des options.

```
npm list --json
```

- **json** - Affiche des informations au format json
- **long** - Affiche des informations étendues
- **analysable** - Affiche la liste analysable au lieu de l'arborescence
- **global** - Affiche les packages globalement installés
- **depth** - **Profondeur** maximale d'affichage de l'arbre de dépendance
- **dev** / **development** - Affiche les devDependencies
- **prod** / **production** - Affiche les dépendances

Si vous le souhaitez, vous pouvez également accéder à la page d'accueil du package.

```
npm home <package name>
```

Mise à jour des npm et des packages

Comme npm lui-même est un module Node.js, il peut être mis à jour lui-même.

Si le système d'exploitation est Windows, vous devez exécuter l'invite de commande en tant qu'administrateur

```
npm install -g npm@latest
```

Si vous voulez vérifier les versions mises à jour, vous pouvez faire:

```
npm outdated
```

Afin de mettre à jour un paquet spécifique:

```
npm update <package name>
```

> Cela mettra à jour le package à la dernière version en fonction des restrictions de package.json

Si vous souhaitez également verrouiller la version mise à jour dans package.json:

```
npm update <package name> --save
```

Verrouillage de modules sur des versions spécifiques

Par défaut, npm installe la dernière version disponible des modules en fonction de la version sémantique de chaque dépendance. Cela peut être problématique si un auteur de module n'adhère pas à semver et introduit des modifications de rupture dans une mise à jour de module, par exemple.

Pour verrouiller la version de chaque dépendance (et les versions de leurs dépendances, etc.) à la version spécifique installée localement dans le dossier `node_modules` , utilisez

```
npm shrinkwrap
```

Cela créera alors un `npm-shrinkwrap.json` côté de votre `package.json` qui répertorie les versions spécifiques des dépendances.

Mise en place de packages globalement installés

Vous pouvez utiliser `npm install -g` pour installer un package "globalement". Cela est généralement fait pour installer un exécutable que vous pouvez ajouter à votre chemin à exécuter. Par exemple:

```
npm install -g gulp-cli
```

Si vous mettez à jour votre chemin, vous pouvez appeler directement `gulp` .

Sur de nombreux systèmes d'exploitation, `npm install -g` tentera d'écrire dans un répertoire dans lequel votre utilisateur ne pourra peut-être pas écrire, tel que `/usr/bin` . Vous ne devez **pas** utiliser

`sudo npm install` dans ce cas, car il existe un risque de sécurité lié à l'exécution de scripts arbitraires avec `sudo` et l'utilisateur root peut créer des répertoires dans lesquels vous ne pouvez pas écrire, ce qui complique les futures installations.

Vous pouvez indiquer à `npm` où installer les modules globaux via votre fichier de configuration, `~/.npmrc` . Cela s'appelle le `prefix` que vous pouvez afficher avec le `npm prefix` .

```
prefix=~/.npm-global-modules
```

Cela utilisera le préfixe chaque fois que vous exécuterez `npm install -g` . Vous pouvez également utiliser `npm install --prefix ~/.npm-global-modules` pour définir le préfixe lors de l'installation. Si le préfixe est identique à votre configuration, vous n'avez pas besoin d'utiliser `-g` .

Pour utiliser le module installé globalement, il doit être sur votre chemin:

```
export PATH=$PATH:~/.npm-global-modules/bin
```

Maintenant, lorsque vous exécutez `npm install -g gulp-cli` vous pourrez utiliser `gulp` .

Remarque: Lorsque vous `npm install` (sans `-g`), le préfixe sera le répertoire contenant `package.json` ou le répertoire en cours si aucun élément n'est trouvé dans la hiérarchie. Cela crée également un répertoire `node_modules/.bin` les exécutables. Si vous souhaitez utiliser un exécutable spécifique à un projet, il n'est pas nécessaire d'utiliser `npm install -g` . Vous pouvez utiliser celui de `node_modules/.bin` .

Lier des projets pour un débogage et un développement plus rapides

La création de dépendances de projet peut parfois être une tâche fastidieuse. Au lieu de publier une version de package sur NPM et d'installer la dépendance pour tester les modifications, utilisez le `npm link` . `npm link` crée un lien symbolique afin que le dernier code puisse être testé dans un environnement local. Cela facilite le test des outils globaux et des dépendances de projet en autorisant le dernier code exécuté avant de publier une version.

Texte d'aide

```
NAME
       npm-link - Symlink a package folder

SYNOPSIS
         npm link (in package dir)
         npm link [<@scope>/]<pkg>[@<version>]

         alias: npm ln
```

Étapes pour lier les dépendances du projet

Lors de la création du lien de dépendance, notez que le nom du package est ce qui va être référencé dans le projet parent.

1. CD dans un répertoire de dépendances (ex: `cd ../my-dep`)
2. `npm link`
3. CD dans le projet qui va utiliser la dépendance
4. `npm link my-dep` ou si namespaced `npm link @namespace/my-dep`

Étapes pour lier un outil global

1. CD dans le répertoire du projet (ex: `cd eslint-watch`)
2. `npm link`
3. Utiliser l'outil
4. `esw --quiet`

Problèmes pouvant survenir

La liaison de projets peut parfois provoquer des problèmes si la dépendance ou l'outil global est déjà installé. `npm uninstall (-g) <pkg>` , puis en exécutant le `npm link` résout normalement tous les problèmes pouvant survenir.

Lire npm en ligne: https://riptutorial.com/fr/node-js/topic/482/npm

Chapitre 87: nvm - Node Version Manager

Remarques

Les URL utilisées dans les exemples ci-dessus font référence à une version spécifique de Node Version Manager. Il est fort probable que la dernière version diffère de ce qui est référencé. Pour installer NVM en utilisant la dernière version, cliquez ici pour accéder à NVM sur GitHub, qui vous fournira les dernières URL.

Examples

Installez NVM

Vous pouvez utiliser `curl` :

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Ou vous pouvez utiliser `wget` :

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

Vérifiez la version de NVM

Pour vérifier que nvm a été installé, procédez comme suit:

```
command -v nvm
```

qui devrait sortir "nvm" si l'installation a réussi.

Installation d'une version de noeud spécifique

Liste des versions distantes disponibles pour l'installation

```
nvm ls-remote
```

Installation d'une version distante

```
nvm install <version>
```

Par exemple

```
nvm install 0.10.13
```

Utiliser une version de noeud déjà installée

Pour répertorier les versions locales disponibles du noeud via NVM:

```
nvm ls
```

Par exemple, si `nvm ls` renvoie:

```
$ nvm ls
     v4.3.0
     v5.5.0
```

Vous pouvez passer à la `v5.5.0` avec:

```
nvm use v5.5.0
```

Installez nvm sur Mac OSX

PROCESSUS D'INSTALLATION

Vous pouvez installer Node Version Manager en utilisant git, curl ou wget. Vous exécutez ces commandes dans **Terminal** sur **Mac OSX** .

exemple curl:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

wget exemple:

```
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.31.3/install.sh | bash
```

TESTEZ QUE NVM A ÉTÉ INSTALLÉ CORRECTEMENT

Pour tester que nvm a été correctement installé, fermez et `nvm` Terminal et entrez `nvm` . Si vous obtenez un **message nvm: commande introuvable** , votre système d'exploitation peut ne pas avoir le fichier **.bash_profile** nécessaire. Dans Terminal, entrez `touch ~/.bash_profile` et exécutez à nouveau le script d'installation ci-dessus.

Si vous avez toujours la **commande nvm: not found** , essayez ce qui suit:

- Dans Terminal, entrez `nano .bashrc` . Vous devriez voir un script d'exportation presque identique à celui-ci:

  export NVM_DIR = ”/ Users / johndoe / .nvm” [-s “$ NVM_DIR / nvm.sh”] &&. “$ NVM_DIR / nvm.sh”

- Copiez le script d'exportation et supprimez-le de **.bashrc**
- Enregistrez et fermez le fichier .bashrc (CTRL + O - Entrée - CTRL + X)
- Ensuite, entrez `nano .bash_profile` pour ouvrir le profil Bash

- Collez le script d'exportation que vous avez copié dans le profil Bash sur une nouvelle ligne
- Enregistrer et fermer le profil Bash (CTRL + O - Entrée - CTRL + X)
- Enfin, entrez `nano .bashrc` pour ré-ouvrir le fichier **.bashrc**
- Collez la ligne suivante dans le fichier:

  source ~ / .nvm / nvm.sh

- Enregistrer et fermer (CTRL + O - Entrée - CTRL + X)
- Redémarrez Terminal et entrez `nvm` pour tester si cela fonctionne

Définition de l'alias pour la version de noeud

Si vous souhaitez définir un nom d'alias pour la version de noeud installée, procédez comme suit:

```
nvm alias <name> <version>
```

Similaire à unalias, faites:

```
nvm unalias <name>
```

Si vous souhaitez définir une autre version que la version stable en tant qu'alias par défaut, vous pouvez utiliser une méthode appropriée. `default` versions avec alias `default` sont chargées sur la console par défaut.

Comme:

```
nvm alias default 5.0.1
```

Ensuite, chaque fois que la **console / le terminal** démarrera, la version 5.0.1 sera présente par défaut.

Remarque:

```
nvm alias # lists all aliases created on nvm
```

Exécuter une commande arbitraire dans un sous-shell avec la version de noeud souhaitée

Liste toutes les versions de nœuds installées

```
nvm ls
    v4.5.0
    v6.7.0
```

Exécuter la commande en utilisant n'importe quelle version de noeud installée

```
nvm run 4.5.0 --version or nvm exec 4.5.0 node --version
Running node v4.5.0 (npm v2.15.9)
```

```
v4.5.0
```

```
nvm run 6.7.0 --version or nvm exec 6.7.0 node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0
```

---

en utilisant un alias

```
nvm run default --version or nvm exec default node --version
Running node v6.7.0 (npm v3.10.3)
v6.7.0
```

Pour installer la version LTS du noeud

```
nvm install --lts
```

Changement de version

```
nvm use v4.5.0 or nvm use stable ( alias )
```

Lire nvm - Node Version Manager en ligne: https://riptutorial.com/fr/node-js/topic/2823/nvm---node-version-manager

Chapitre 88: OAuth 2.0

Examples

OAuth 2 avec l'implémentation de Redis - grant_type: password

Dans cet exemple, j'utiliserai api oauth2 in rest avec la base de données redis

> **Important:** Vous devez installer la base de données redis sur votre ordinateur, téléchargez-la à partir d' ici pour les utilisateurs Linux et à partir d' ici pour installer la version Windows, et nous utiliserons l'appli redis manager desktop, installez-la à partir d' ici .

Maintenant, nous devons configurer notre serveur node.js pour utiliser la base de données redis.

- **Création du fichier serveur: app.js**

```
    var express = require('express'),
  bodyParser = require('body-parser'),
  oauthserver = require('oauth2-server'); // Would be: 'oauth2-server'

var app = express();

app.use(bodyParser.urlencoded({ extended: true }));

app.use(bodyParser.json());

app.oauth = oauthserver({
  model: require('./routes/Oauth2/model'),
  grants: ['password', 'refresh_token'],
  debug: true
});

// Handle token grant requests
app.all('/oauth/token', app.oauth.grant());

app.get('/secret', app.oauth.authorise(), function (req, res) {
  // Will require a valid access_token
  res.send('Secret area');
});

app.get('/public', function (req, res) {
  // Does not require an access_token
  res.send('Public area');
});

// Error handling
app.use(app.oauth.errorHandler());

app.listen(3000);
```

- **Créer un modèle Oauth2 dans routes / Oauth2 / model.js**

```
 var model = module.exports,
  util = require('util'),
  redis = require('redis');

var db = redis.createClient();

var keys = {
  token: 'tokens:%s',
  client: 'clients:%s',
  refreshToken: 'refresh_tokens:%s',
  grantTypes: 'clients:%s:grant_types',
  user: 'users:%s'
};

model.getAccessToken = function (bearerToken, callback) {
  db.hgetall(util.format(keys.token, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      accessToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.getClient = function (clientId, clientSecret, callback) {
  db.hgetall(util.format(keys.client, clientId), function (err, client) {
    if (err) return callback(err);

    if (!client || client.clientSecret !== clientSecret) return callback();

    callback(null, {
      clientId: client.clientId,
      clientSecret: client.clientSecret
    });
  });
};

model.getRefreshToken = function (bearerToken, callback) {
  db.hgetall(util.format(keys.refreshToken, bearerToken), function (err, token) {
    if (err) return callback(err);

    if (!token) return callback();

    callback(null, {
      refreshToken: token.accessToken,
      clientId: token.clientId,
      expires: token.expires ? new Date(token.expires) : null,
      userId: token.userId
    });
  });
};

model.grantTypeAllowed = function (clientId, grantType, callback) {
  db.sismember(util.format(keys.grantTypes, clientId), grantType, callback);
};
```

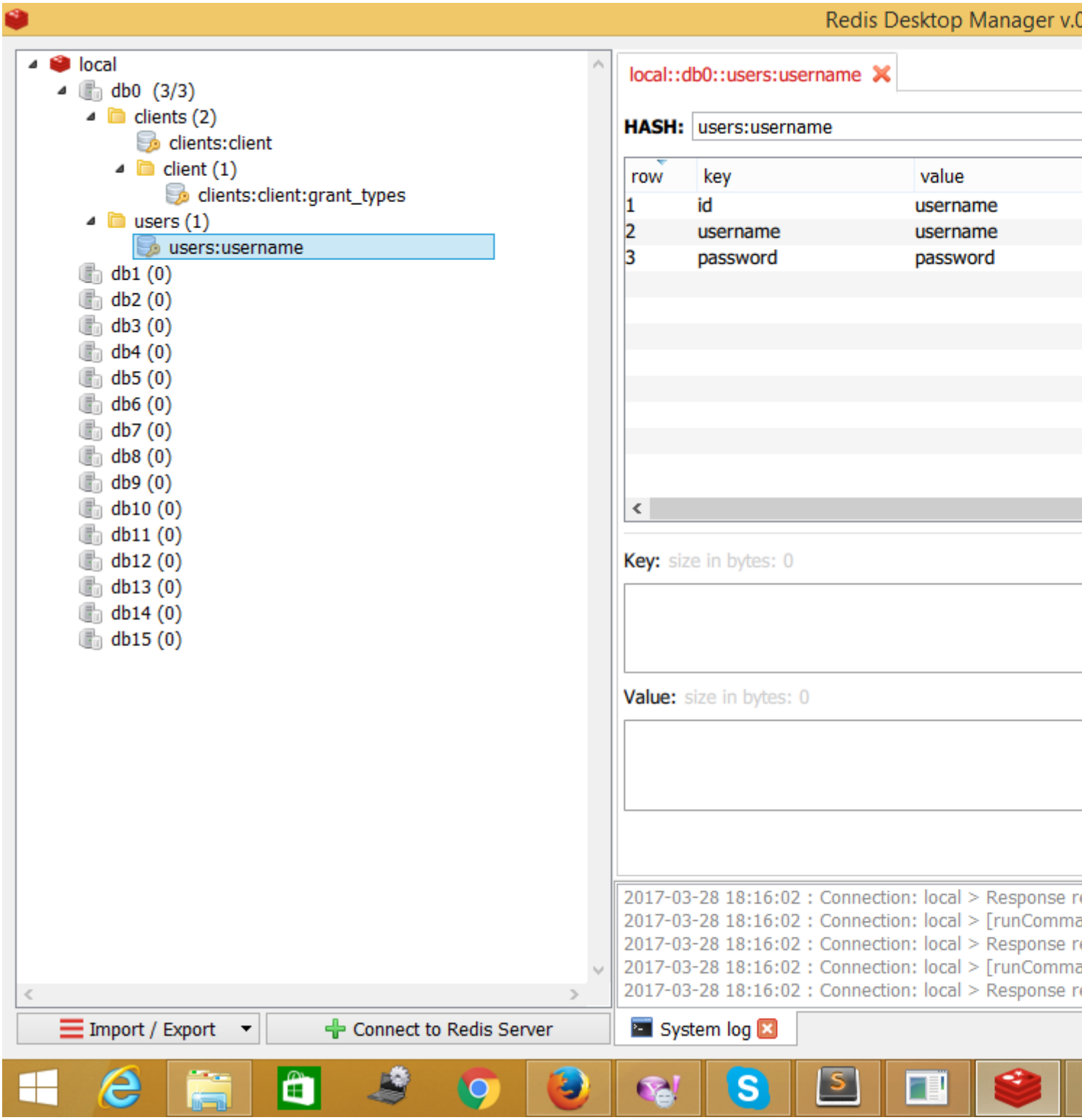
```
model.saveAccessToken = function (accessToken, clientId, expires, user, callback) {
  db.hmset(util.format(keys.token, accessToken), {
    accessToken: accessToken,
    clientId: clientId,
    expires: expires ? expires.toISOString() : null,
    userId: user.id
  }, callback);
};

model.saveRefreshToken = function (refreshToken, clientId, expires, user, callback) {
  db.hmset(util.format(keys.refreshToken, refreshToken), {
    refreshToken: refreshToken,
    clientId: clientId,
    expires: expires ? expires.toISOString() : null,
    userId: user.id
  }, callback);
};

model.getUser = function (username, password, callback) {
  db.hgetall(util.format(keys.user, username), function (err, user) {
    if (err) return callback(err);

    if (!user || password !== user.password) return callback();

    callback(null, {
      id: username
    });
  });
};
```

Vous devez uniquement installer redis sur votre machine et exécuter le fichier de nœud suivant

```
  #! /usr/bin/env node

var db = require('redis').createClient();

db.multi()
  .hmset('users:username', {
    id: 'username',
    username: 'username',
    password: 'password'
  })
  .hmset('clients:client', {
    clientId: 'client',
    clientSecret: 'secret'
  })//clientId + clientSecret to base 64 will generate Y2xpZW50OnNlY3JldA==
  .sadd('clients:client:grant_types', [
    'password',
    'refresh_token'
  ])
  .exec(function (errs) {
    if (errs) {
      console.error(errs[0].message);
      return process.exit(1);
    }

    console.log('Client and user added successfully');
    process.exit();
  });
```

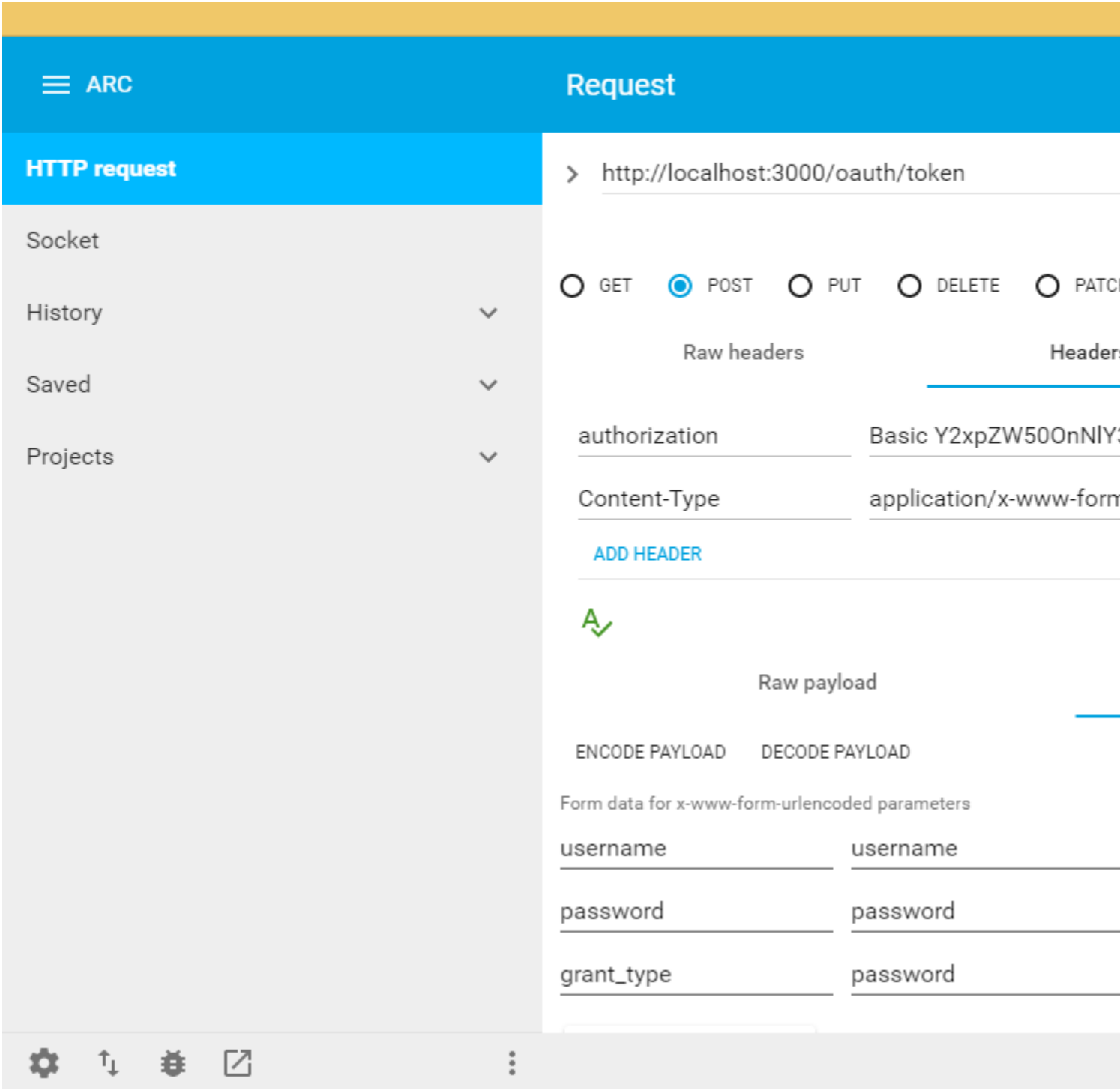
Remarque : Ce fichier définira les informations d'identification de votre interface pour demander un jeton. Votre demande auprès de

Exemple de base de données redis après l'appel du fichier ci-dessus:



La demande sera comme suit:

Exemple d'appel à api



Entête:

1. autorisation: Basic suivi du mot de passe défini lors de la première configuration de redis:

    une. clientId + secretId à base64

2. Formulaire de données:

    nom d'utilisateur: utilisateur de cette requête

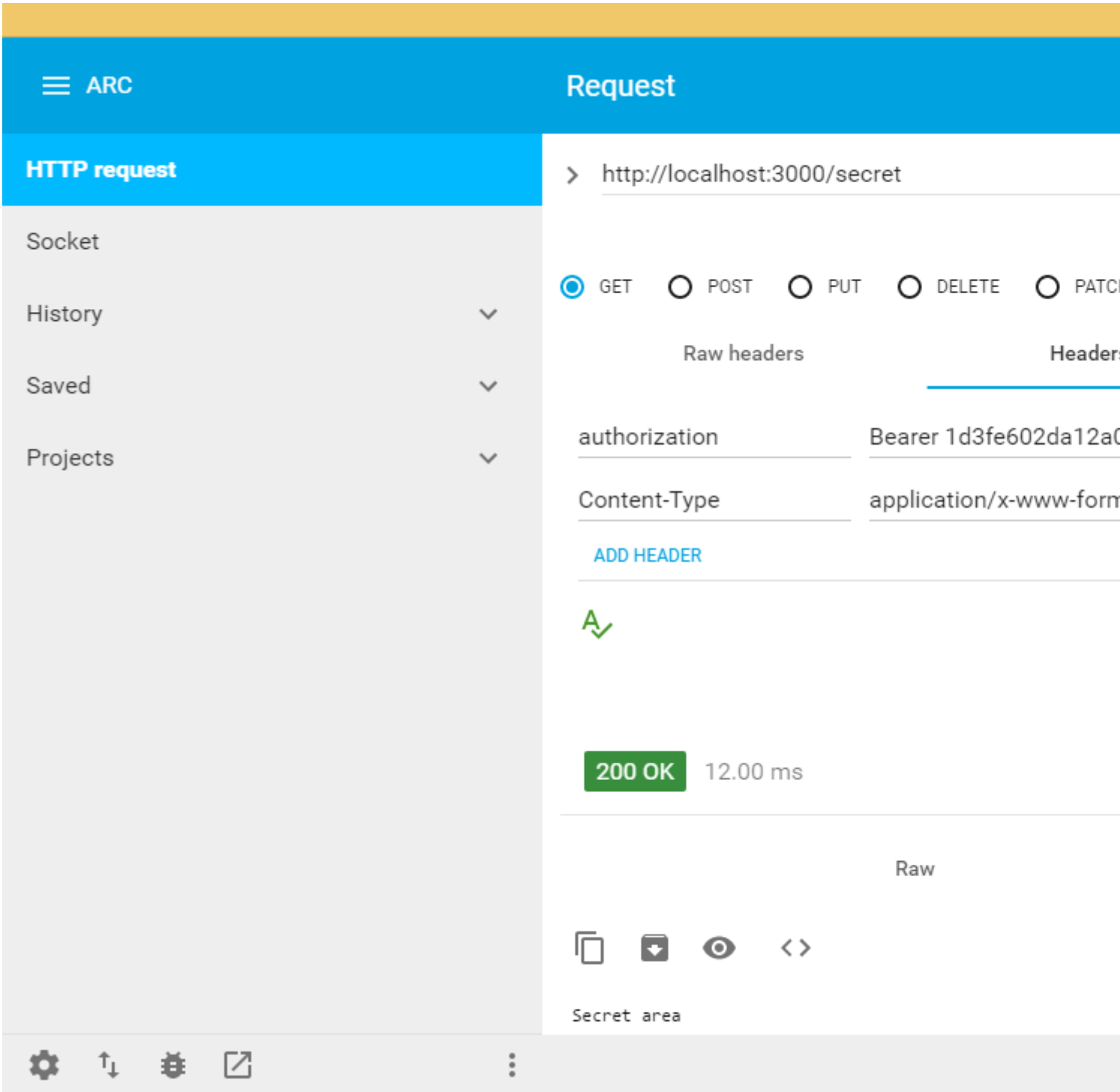
    mot de passe: mot de passe utilisateur

    grant_type: dépend des options que vous souhaitez, je choisis le mot de passe

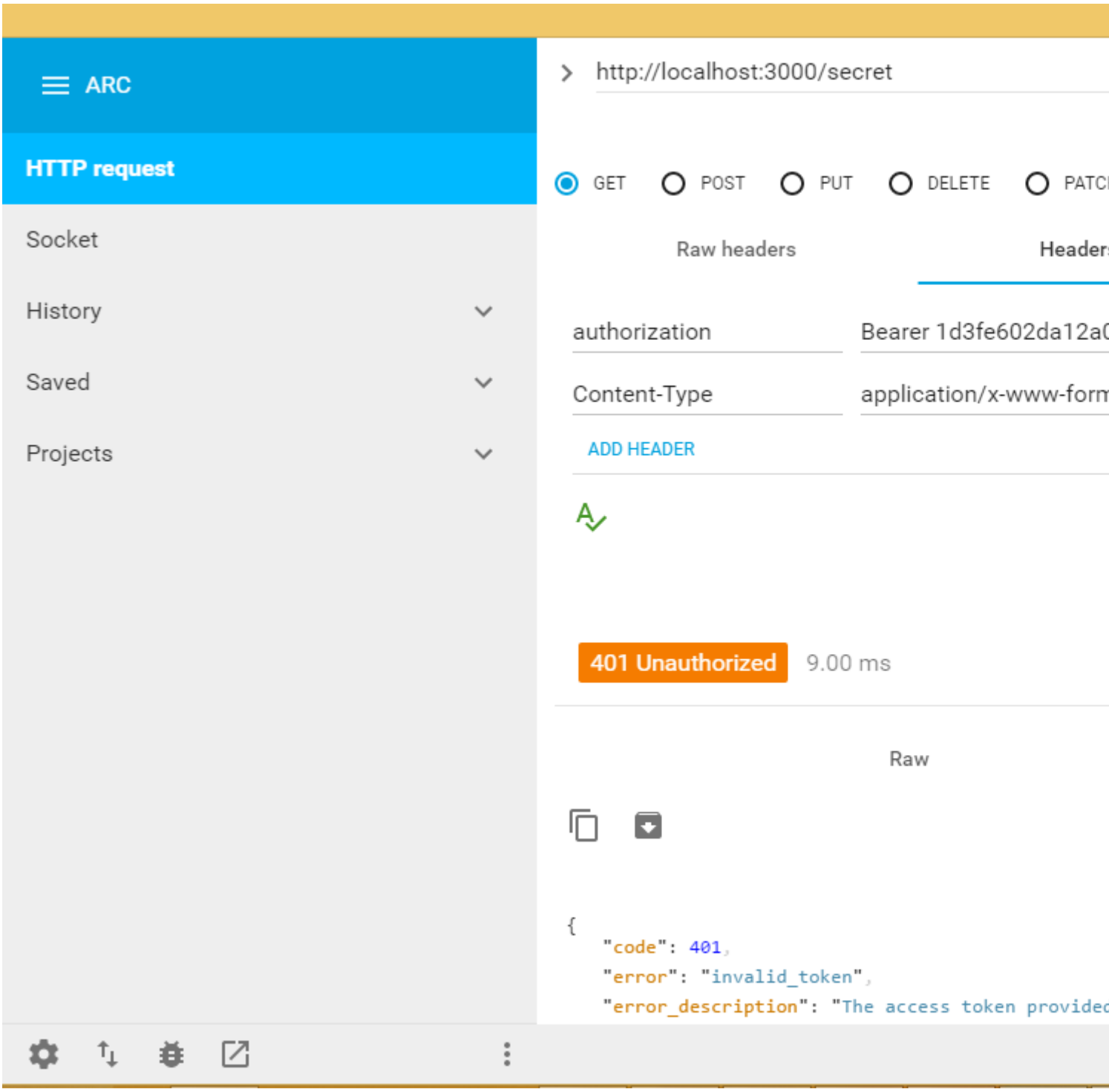
qui prend uniquement le nom d'utilisateur et le mot de passe à créer dans redis, les données sur redis seront comme ci-dessous:

```
{
  "access_token":"1d3fe602da12a086ecb2b996fd7b7ae874120c4f",
  "token_type":"bearer", // Will be used to access api + access+token e.g. bearer
1d3fe602da12a086ecb2b996fd7b7ae874120c4f
  "expires_in":3600,
  "refresh_token":"b6ad56e5c9aba63c85d7e21b1514680bbf711450"
}
```

Nous devons donc appeler notre API et récupérer des données sécurisées avec notre jeton d'accès que nous venons de créer, voir ci-dessous:

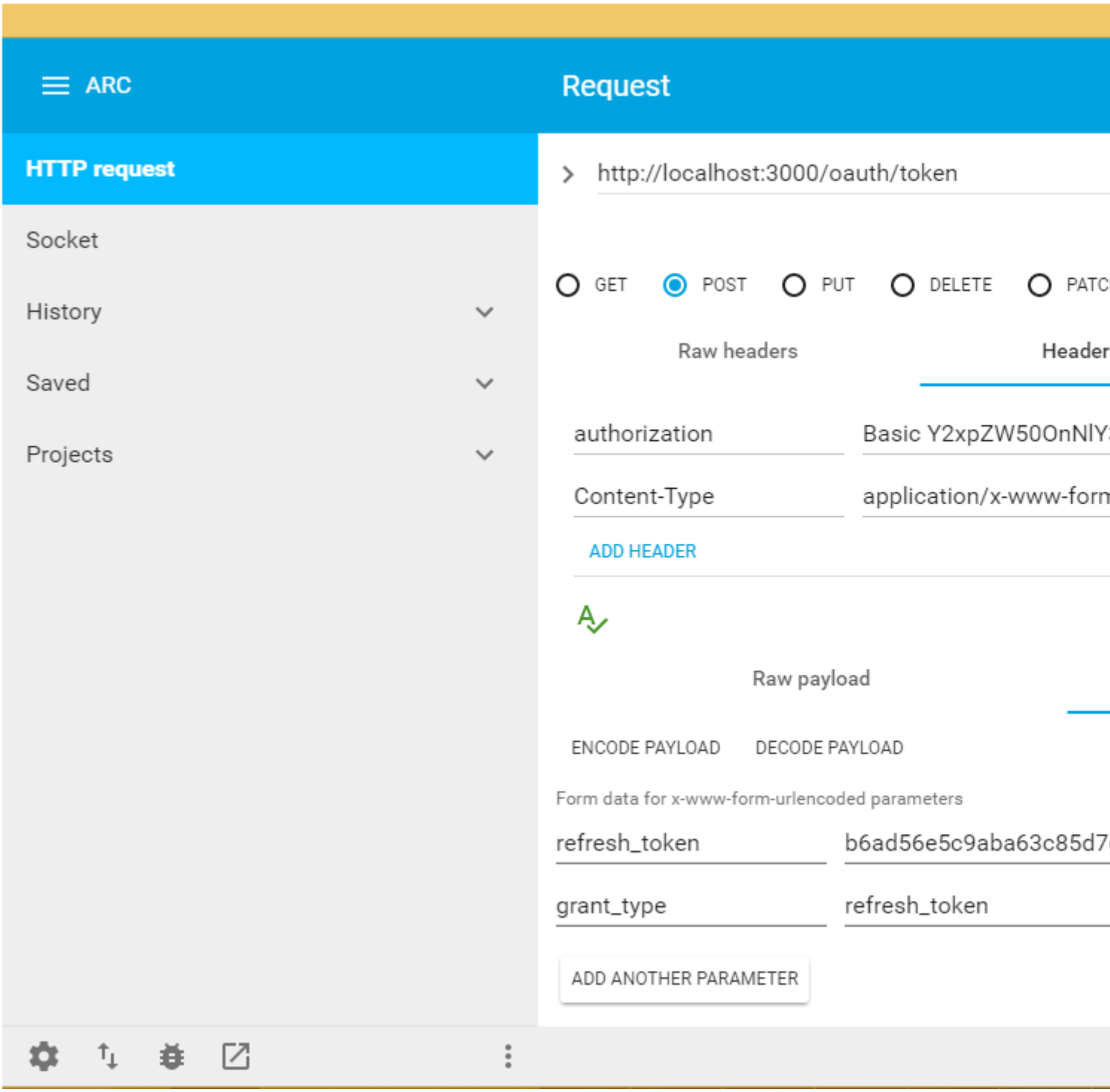


lorsque le jeton expire, api génère une erreur indiquant que le jeton expire et que vous ne pouvez accéder à aucun des appels d'API, voir l'image ci-dessous:



> Voyons ce qu'il faut faire si le jeton expire. Permettez-moi d'abord de vous l'expliquer, si le jeton d'accès expire, un élément refresh_token existe dans redis qui fait référence à access_token expiré. autorisation à l'ID de base du client: clientsecret (à la base 64!) et enfin envoyer le refresh_token, cela générera un nouvel access_token avec une nouvelle donnée d'expiration.

L'image suivante montre comment obtenir un nouveau jeton d'accès:



J'espère vous aider!

Lire OAuth 2.0 en ligne: https://riptutorial.com/fr/node-js/topic/9566/oauth-2-0

Chapitre 89: package.json

Remarques

Vous pouvez créer `package.json` avec

```
npm init
```

qui vous demandera des informations de base sur vos projets, y compris l' identificateur de licence .

Examples

Définition de base du projet

```
{
    "name": "my-project",
    "version": "0.0.1",
    "description": "This is a project.",
    "author": "Someone <someone@example.com>",
    "contributors": [{
        "name": "Someone Else",
        "email": "else@example.com"
    }],
    "keywords": ["improves", "searching"]
}
```

Champ	La description
prénom	un champ **obligatoire** pour un package à installer. Doit être en minuscule, un seul mot sans espaces. (Tirets et traits de soulignement autorisés)
version	un champ **obligatoire** pour la version du package utilisant le contrôle de version sémantique .
la description	une brève description du projet
auteur	spécifie l'auteur du paquet
contributeurs	un tableau d'objets, un pour chaque contributeur
mots clés	un tableau de chaînes, cela aidera les personnes à trouver votre paquet

Les dépendances

"dependencies": {"nom-module": "0.1.0"}

- **exact** : `0.1.0` va installer cette version spécifique du module.
- **Dernière version mineure** : `^0.1.0` installera la version mineure la plus récente, par exemple `0.2.0` , mais n'installera pas de module avec une version majeure supérieure, par exemple `1.0.0`
- **Le dernier patch** : `0.1.x` ou `~0.1.0` installera la version la plus récente du patch disponible, par exemple `0.1.4` , mais n'installe pas de module avec une version majeure ou mineure plus élevée, par exemple `0.2.0` ou `1.0.0` .
- **caractère générique** : `*` installera la dernière version du module.
- **git repository** : ce qui suit va installer une archive tar à partir de la branche principale d'un **dépôt** git. Un `#sha` , `#tag` ou `#branch` peut également être fourni:
  - **GitHub** : `user/project` ou `user/project#v1.0.0`
  - **url** : `git://gitlab.com/user/project.git` ou `git://gitlab.com/user/project.git#develop`
- **chemin local** : `file:../lib/project`

Après les avoir ajoutés à votre package.json, utilisez la commande `npm install` dans le répertoire de votre projet dans le terminal.

devDependencies

```
"devDependencies": {
    "module-name": "0.1.0"
}
```

Pour les dépendances requises uniquement pour le développement, comme le test des proxies de style ext. Ces dépendances dev ne seront pas installées lors de l'exécution de "npm install" en mode production.

Scripts

Vous pouvez définir des scripts pouvant être exécutés ou déclenchés avant ou après un autre script.

```
{
  "scripts": {
    "pretest": "scripts/pretest.js",
    "test": "scripts/test.js",
    "posttest": "scripts/posttest.js"
  }
}
```

Dans ce cas, vous pouvez exécuter le script en exécutant l'une de ces commandes:

```
$ npm run-script test
$ npm run test
$ npm test
$ npm t
```

Scripts prédéfinis

Nom du script	La description
prépublication	Exécuter avant la publication du paquet
publier, postpublier	Exécuter après la publication du package.
préinstaller	Exécuter avant l'installation du package.
installer, postinstaller	Exécuter après l'installation du package.
preuninstall, désinstaller	Exécuter avant la désinstallation du package.
post-install	Exécuter après la désinstallation du package.
preversion, version	Exécuter avant de cogner la version du package.
postversion	Exécuter après bump la version du package.
prétest, test, post-test	Exécuter par la commande de `npm test`
prestop, stop, poststop	Exécutée par la commande `npm stop`
prestart, start, poststart	Exécuter par la commande de `npm start`
prerestart, restart, postrestart	Exécuter par la commande `npm restart`

Scripts définis par l'utilisateur

Vous pouvez également définir vos propres scripts de la même manière que vous le faites avec les scripts prédéfinis:

```
{
  "scripts": {
    "preci": "scripts/preci.js",
    "ci": "scripts/ci.js",
    "postci": "scripts/postci.js"
  }
}
```

Dans ce cas, vous pouvez exécuter le script en exécutant l'une de ces commandes:

```
$ npm run-script ci
$ npm run ci
```

Les scripts définis par l'utilisateur prennent également en charge *les* scripts *pré* et *post* , comme le

montre l'exemple ci-dessus.

Définition étendue du projet

Certains des attributs supplémentaires sont analysés par le site Web npm, comme le `repository` , les `bugs` ou la `homepage` et affichés dans l'infobox pour ces paquets.

```
{
  "main": "server.js",
  "repository" :  {
    "type": "git",
    "url": "git+https://github.com/<accountname>/<repositoryname>.git"
  },
  "bugs": {
    "url": "https://github.com/<accountname>/<repositoryname>/issues"
  },
  "homepage": "https://github.com/<accountname>/<repositoryname>#readme",
  "files": [
    "server.js", // source files
    "README.md", // additional files
    "lib" // folder with all included files
  ]
}
```

Champ	La description
principale	Script d'entrée pour ce package. Ce script est renvoyé lorsqu'un utilisateur requiert le package.
dépôt	Emplacement et type du référentiel public
bogues	Bugtracker pour ce paquet (par exemple github)
page d'accueil	Page d'accueil pour ce package ou le projet général
des dossiers	Liste des fichiers et dossiers à télécharger lorsqu'un utilisateur effectue une `npm install <packagename>`

Explorer le package.json

Un fichier `package.json` , généralement présent dans la racine du projet, contient les métadonnées relatives à votre application ou à votre module, ainsi que la liste des dépendances à installer à partir de npm lors de l'exécution de `npm install` .

Pour initialiser un `package.json` tapez `npm init` dans votre invite de commande.

Pour créer un `package.json` avec des valeurs par défaut, utilisez:

```
npm init --yes
# or
npm init -y
```

Pour installer un package et l'enregistrer dans `package.json` utilisez:

```
npm install {package name} --save
```

Vous pouvez également utiliser la notation abrégée:

```
 npm i -S {package name}
```

Alias NPM `-S` à `--save` et `-D` à `--save-dev` pour enregistrer respectivement dans vos dépendances de production ou de développement.

Le paquet apparaîtra dans vos dépendances; Si vous utilisez `--save-dev` au lieu de `--save` , le paquet apparaîtra dans vos devDependencies.

Propriétés importantes de `package.json` :

```
{
  "name": "module-name",
  "version": "10.3.1",
  "description": "An example module to illustrate the usage of a package.json",
  "author": "Your Name <your.name@example.org>",
  "contributors": [{
    "name": "Foo Bar",
    "email": "foo.bar@example.com"
  }],
  "bin": {
    "module-name": "./bin/module-name"
  },
  "scripts": {
    "test": "vows --spec --isolate",
    "start": "node index.js",
    "predeploy": "echo About to deploy",
    "postdeploy": "echo Deployed",
    "prepublish": "coffee --bare --compile --output lib/foo src/foo/*.coffee"
  },
  "main": "lib/foo.js",
  "repository": {
    "type": "git",
    "url": "https://github.com/username/repo"
  },
  "bugs": {
    "url": "https://github.com/username/issues"
  },
  "keywords": [
    "example"
  ],
  "dependencies": {
    "express": "4.2.x"
  },
  "devDependencies": {
    "assume": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"
  },
  "peerDependencies": {
    "moment": ">2.0.0"
  },
  "preferGlobal": true,
  "private": true,
```

```
  "publishConfig": {
    "registry": "https://your-private-hosted-npm.registry.domain.com"
  },
  "subdomain": "foobar",
  "analyze": true,
  "license": "MIT",
  "files": [
    "lib/foo.js"
  ]
}
```

Informations sur certaines propriétés importantes:

```
name
```

Le nom unique de votre package et devrait être en minuscule. Cette propriété est requise et votre package ne sera pas installé sans lui.

1. Le nom doit être inférieur ou égal à 214 caractères.
2. Le nom ne peut pas commencer par un point ou un trait de soulignement.
3. Les nouveaux paquets ne doivent pas comporter de majuscules dans le nom.

```
version
```

La version du package est spécifiée par Semantic Versioning (semver). Ce qui suppose qu'un numéro de version est écrit en tant que MAJOR.MINOR.PATCH et que vous incrémentez le:

1. Version MAJOR lorsque vous apportez des modifications incompatibles à l'API
2. Version MINOR lorsque vous ajoutez des fonctionnalités de manière rétrocompatible
3. Version PATCH lorsque vous apportez des corrections de bogues rétro-compatibles

```
description
```

La description du projet. Essayez de le garder court et concis.

```
author
```

L'auteur de ce paquet.

```
bin
```

Un objet utilisé pour exposer des scripts binaires de votre package. L'objet suppose que la clé est le nom du script binaire et la valeur un chemin d'accès relatif au script.

Cette propriété est utilisée par les packages contenant une interface de ligne de commande (CLI).

```
script
```

Un objet qui expose des commandes npm supplémentaires. L'objet suppose que la clé est la

commande npm et que la valeur est le chemin du script. Ces scripts peuvent être exécutés lorsque vous exécutez `npm run {command name}` ou `npm run-script {command name}` .

Les packages contenant une interface de ligne de commande et installés localement peuvent être appelés sans chemin relatif. Ainsi , au lieu d'appeler `./node-modules/.bin/mocha` vous pouvez appeler directement `mocha` .

```
main
```

Le point d'entrée principal de votre forfait. Lors de l'appel de `require('{module name}')` dans le noeud, ce sera le fichier réel requis.

Il est fortement recommandé que le fait de demander le fichier principal ne génère aucun effet secondaire. Par exemple, exiger que le fichier principal ne démarre pas un serveur HTTP ou ne se connecte pas à une base de données. Au lieu de cela, vous devriez créer quelque chose comme `exports.init = function () {...}` dans votre script principal.

```
keywords
```

Un tableau de mots-clés décrivant votre package. Cela aidera les gens à trouver votre paquet.

```
devDependencies
```

Ce sont les dépendances qui sont uniquement destinées au développement et aux tests de votre module. Les dépendances seront installées automatiquement à moins que la variable d'environnement de `NODE_ENV=production` ait été définie. Si tel est le cas, vous pouvez toujours utiliser ces paquetages avec `npm install --dev`

```
peerDependencies
```

Si vous utilisez ce module, peerDependencies répertorie les modules à installer avec celui-ci. Par exemple, `moment-timezone` doit être installé à côté de l' `moment` car il s'agit d'un plugin pour l'instant, même s'il ne `require("moment")` pas directement `require("moment")` .

```
preferGlobal
```

Une propriété qui indique que cette page préfère être installée globalement à l'aide de `npm install -g {module-name}` . Cette propriété est utilisée par les packages contenant une interface de ligne de commande (CLI).

Dans toutes les autres situations, vous ne devez PAS utiliser cette propriété.

```
publishConfig
```

Le publishConfig est un objet avec des valeurs de configuration qui seront utilisées pour la publication des modules. Les valeurs de configuration définies remplacent votre configuration npm par défaut.

L'utilisation la plus courante de `publishConfig` consiste à publier votre package dans un registre privé npm, ce qui vous permet de bénéficier des avantages de npm, à l'exception des packages privés. Cela se fait simplement en définissant l'URL de votre npm privé comme valeur pour la clé de registre.

```
files
```

Ceci est un tableau de tous les fichiers à inclure dans le package publié. Un chemin de fichier ou un chemin de dossier peut être utilisé. Tout le contenu d'un chemin de dossier sera inclus. Cela réduit la taille totale de votre package en incluant uniquement les fichiers corrects à distribuer. Ce champ fonctionne conjointement avec un fichier de règles `.npmignore` .

La source

Lire package.json en ligne: https://riptutorial.com/fr/node-js/topic/1515/package-json

Chapitre 90: passport.js

Introduction

Passport est un module d'autorisation populaire pour les nœuds. En termes simples, il gère toutes les demandes d'autorisation sur votre application par les utilisateurs. Passport prend en charge plus de 300 stratégies afin que vous puissiez facilement intégrer la connexion à Facebook / Google ou à tout autre réseau social qui l'utilise. La stratégie que nous allons aborder ici est la section locale où vous authentifiez un utilisateur à l'aide de votre propre base de données d'utilisateurs enregistrés (en utilisant un nom d'utilisateur et un mot de passe).

Examples

Exemple de LocalStrategy dans passport.js

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;

passport.serializeUser(function(user, done) { //In serialize user you decide what to store in
the session. Here I'm storing the user id only.
  done(null, user.id);
});

passport.deserializeUser(function(id, done) { //Here you retrieve all the info of the user
from the session storage using the user id stored in the session earlier using serialize user.
  db.findById(id, function(err, user) {
    done(err, user);
    });
});

passport.use(new LocalStrategy(function(username, password, done) {
    db.findOne({'username':username},function(err,student){
        if(err)return done(err,{message:message});//wrong roll_number or password;
        var pass_retrieved = student.pass_word;
        bcrypt.compare(password, pass_retrieved, function(err3, correct) {
          if(err3){
            message = [{"msg": "Incorrect Password!"}];
            return done(null,false,{message:message});  // wrong password
          }
          if(correct){
              return done(null,student);
          }
        });
    });
}));

app.use(session({ secret: 'super secret' })); //to make passport remember the user on other
pages too.(Read about session store. I used express-sessions.)
app.use(passport.initialize());
app.use(passport.session());

app.post('/',passport.authenticate('local',{successRedirect:'/users' failureRedirect: '/'}),
    function(req,res,next){
```

```
});
```

Lire passport.js en ligne: https://riptutorial.com/fr/node-js/topic/8812/passport-js

Chapitre 91: Performances Node.js

Examples

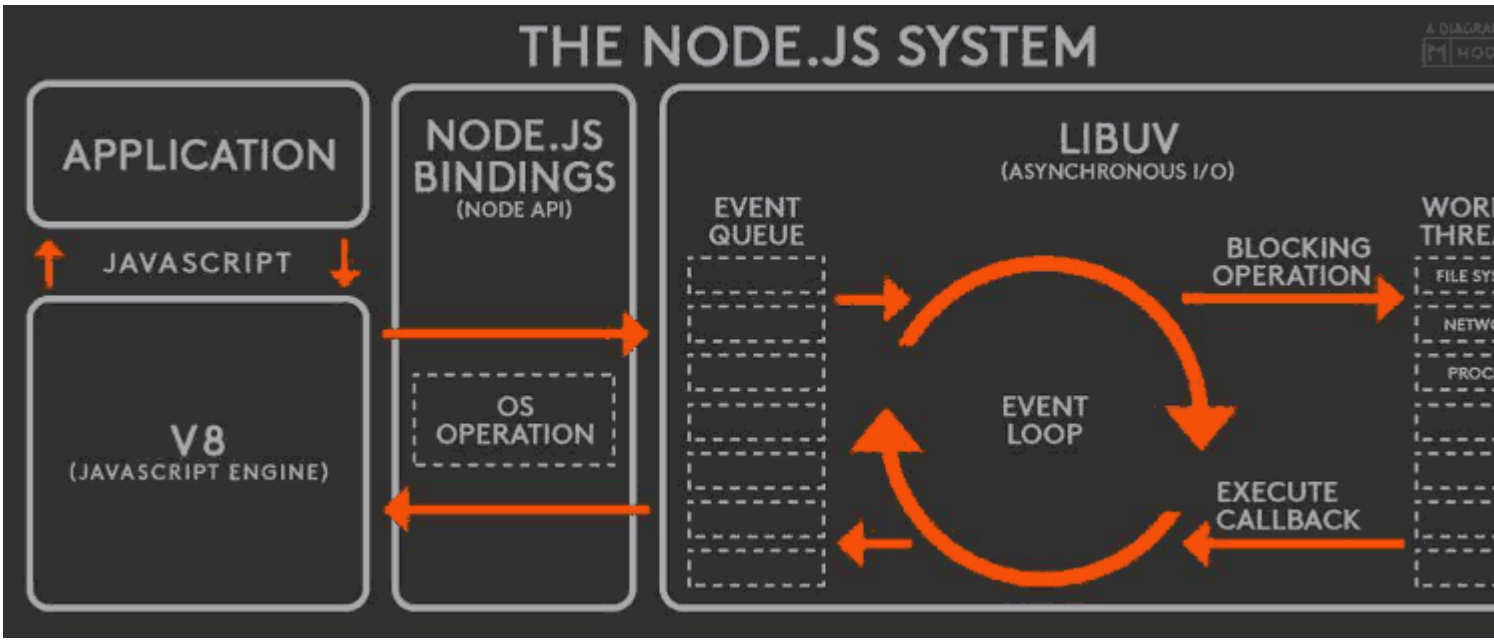
Boucle d'événement

Exemple d'opération de blocage

```
let loop = (i, max) => {
  while (i < max) i++
  return i
}

// This operation will block Node.js
// Because, it's CPU-bound
// You should be careful about this kind of code
loop(0, 1e+12)
```

Exemple d'opération IO non bloquante

```
let i = 0

const step = max => {
  while (i < max) i++
  console.log('i = %d', i)
}

const tick = max => process.nextTick(step, max)

// this will postpone tick run step's while-loop to event loop cycles
// any other IO-bound operation (like filesystem reading) can take place
// in parallel
tick(1e+6)
tick(1e+7)
console.log('this will output before all of tick operations. i = %d', i)
console.log('because tick operations will be postponed')
tick(1e+8)
```



En termes plus simples, Event Loop est un mécanisme de file d'attente à thread unique qui exécute votre code lié à l'UC jusqu'à la fin de son exécution et le code lié aux E / S de manière non bloquante.

Cependant, Node.js sous le tapis utilise plusieurs threads pour certaines de ses opérations via libuv Library.

Considérations de performance

- Les opérations non bloquantes ne bloqueront pas la file d'attente et n'affecteront pas les performances de la boucle.
- Cependant, les opérations liées au processeur bloquent la file d'attente, vous devez donc faire attention à ne pas effectuer d'opérations liées au processeur dans votre code Node.js.

Node.js non-blocs IO car il décharge le travail sur le noyau du système d'exploitation, et lorsque l'opération IO fournit des données (*en tant qu'événement*), il notifiera votre code avec les rappels fournis.

Augmenter les maxSockets

Les bases

```
require('http').globalAgent.maxSockets = 25

// You can change 25 to Infinity or to a different value by experimenting
```

Node.js utilise par défaut `maxSockets = Infinity` en même temps (depuis la v0.12.0). Jusqu'à Node v0.12.0, la valeur par défaut était `maxSockets = 5` (voir v0.11.0). Ainsi, après plus de 5 demandes, ils seront mis en file d'attente. Si vous voulez une concurrence, augmentez ce

nombre.

Définir votre propre agent

`http` API `http` utilise un " **Agent global** " . Vous pouvez fournir votre propre agent. Comme ça:

```
const http = require('http')
const myGloriousAgent = new http.Agent({ keepAlive: true })
myGloriousAgent.maxSockets = Infinity

http.request({ ..., agent: myGloriousAgent }, ...)
```

Désactiver complètement la mise en commun des sockets

```
const http = require('http')
const options = {.....}

options.agent = false

const request = http.request(options)
```

Pièges

- Vous devriez faire la même chose pour les API `https` si vous voulez les mêmes effets
- Attention, par exemple, AWS utilisera 50 au lieu d' `Infinity` .

Activer gzip

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

http.createServer((request, response) => {
  const stream          = fs.createReadStream('index.html')
  const acceptsEncoding = request.headers['accept-encoding']

  let encoder = {
    hasEncoder     : false,
    contentEncoding: {},
    createEncoder  : () => throw 'There is no encoder'
  }

  if (!acceptsEncoding) {
    acceptsEncoding = ''
  }
```

```
  if (acceptsEncoding.match(/\bdeflate\b/)) {
    encoder = {
      hasEncoder     : true,
      contentEncoding: { 'content-encoding': 'deflate' },
      createEncoder  : zlib.createDeflate
    }
  } else if (acceptsEncoding.match(/\bgzip\b/)) {
    encoder = {
      hasEncoder     : true,
      contentEncoding: { 'content-encoding': 'gzip' },
      createEncoder  : zlib.createGzip
    }
  }

  response.writeHead(200, encoder.contentEncoding)

  if (encoder.hasEncoder) {
    stream = stream.pipe(encoder.createEncoder())
  }

  stream.pipe(response)

}).listen(1337)
```

Lire Performances Node.js en ligne: https://riptutorial.com/fr/node-js/topic/9410/performances-node-js

Chapitre 92: Pirater

Examples

Ajouter de nouvelles extensions à exiger ()

Vous pouvez ajouter de nouvelles extensions à `require()` en étendant `require.extensions` .

Pour un exemple **XML** :

```
// Add .xml for require()
require.extensions['.xml'] = (module, filename) => {
    const fs = require('fs')
    const xml2js = require('xml2js')

    module.exports = (callback) => {
        // Read required file.
        fs.readFile(filename, 'utf8', (err, data) => {
            if (err) {
                callback(err)
                return
            }
            // Parse it.
            xml2js.parseString(data, (err, result) => {
                callback(null, result)
            })
        })
    }
}
```

Si le contenu de `hello.xml` est le suivant:

```
<?xml version="1.0" encoding="UTF-8"?>
<foo>
    <bar>baz</bar>
    <qux />
</foo>
```

Vous pouvez le lire et l'analyser via `require()` :

```
require('./hello')((err, xml) {
    if (err)
        throw err;
    console.log(err);
})
```

Il affiche `{ foo: { bar: [ 'baz' ], qux: [ '' ] } }` .

Lire Pirater en ligne: https://riptutorial.com/fr/node-js/topic/6645/pirater

Chapitre 93: Pool de connexions Mysql

Examples

Utilisation d'un pool de connexions sans base de données

Atteindre le multitenancy sur le serveur de base de données avec plusieurs bases de données hébergées sur celui-ci.

La multitenancy est une exigence courante des applications d'entreprise et la création d'un pool de connexions pour chaque base de données sur le serveur de base de données n'est pas recommandée. Ainsi, ce que nous pouvons faire à la place, c'est créer un pool de connexions avec un serveur de base de données et basculer entre les bases de données hébergées sur un serveur de base de données à la demande.

Supposons que notre application dispose de différentes bases de données pour chaque entreprise hébergée sur un serveur de base de données. Nous allons nous connecter à la base de données de l'entreprise respective lorsque l'utilisateur accède à l'application. voici l'exemple sur comment faire cela: -

```
var pool  = mysql.createPool({
      connectionLimit : 10,
      host            : 'example.org',
      user            : 'bobby',
      password        : 'pass'
    });

pool.getConnection(function(err, connection){
    if(err){
        return cb(err);
    }
    connection.changeUser({database : "firm1"});
    connection.query("SELECT * from history", function(err, data){
        connection.release();
        cb(err, data);
    });
});
```

Laissez-moi décomposer l'exemple: -

Lors de la définition de la configuration du pool, je n'ai pas donné le nom de la base de données, mais uniquement le serveur de base de données, c.-à-d.

```
{
  connectionLimit : 10,
  host            : 'example.org',
  user            : 'bobby',
  password        : 'pass'
}
```

Ainsi, lorsque nous voulons utiliser la base de données spécifique sur le serveur de base de données, nous demandons à la connexion d'accéder à la base de données en utilisant: -

```
    connection.changeUser({database : "firm1"});
```

vous pouvez consulter la documentation officielle ici

Lire Pool de connexions Mysql en ligne: https://riptutorial.com/fr/node-js/topic/6353/pool-de-connexions-mysql

Chapitre 94: Prise en main du profilage de nœuds

Introduction

Le but de cet article est de commencer avec le profilage de l'application nodejs et de comprendre ce résultat pour capturer un bogue ou une fuite de mémoire. Une application en cours d'exécution sur nodejs n'est rien d'autre qu'un moteur v8 qui fonctionne de manière similaire à un site Web s'exécutant sur un navigateur et nous pouvons capturer toutes les métriques liées à un processus de site Web pour une application de noeud.

L'outil de mon choix est chrome devtools ou inspecteur chrome couplé à l'inspecteur de nœud.

Remarques

L'inspecteur de nœuds ne parvient parfois pas à joindre le processus bebug du nœud, auquel cas vous ne pourrez pas obtenir le point d'arrêt de débogage dans devtools.

Si ce n'est pas le cas, redémarrez l'inspecteur de noeud à partir de la ligne de commande.

Examples

Profilage d'une application de noeud simple

Étape 1 : Installez le paquet node-inspector en utilisant npm globalement sur votre machine

```
$ npm install -g node-inspector
```

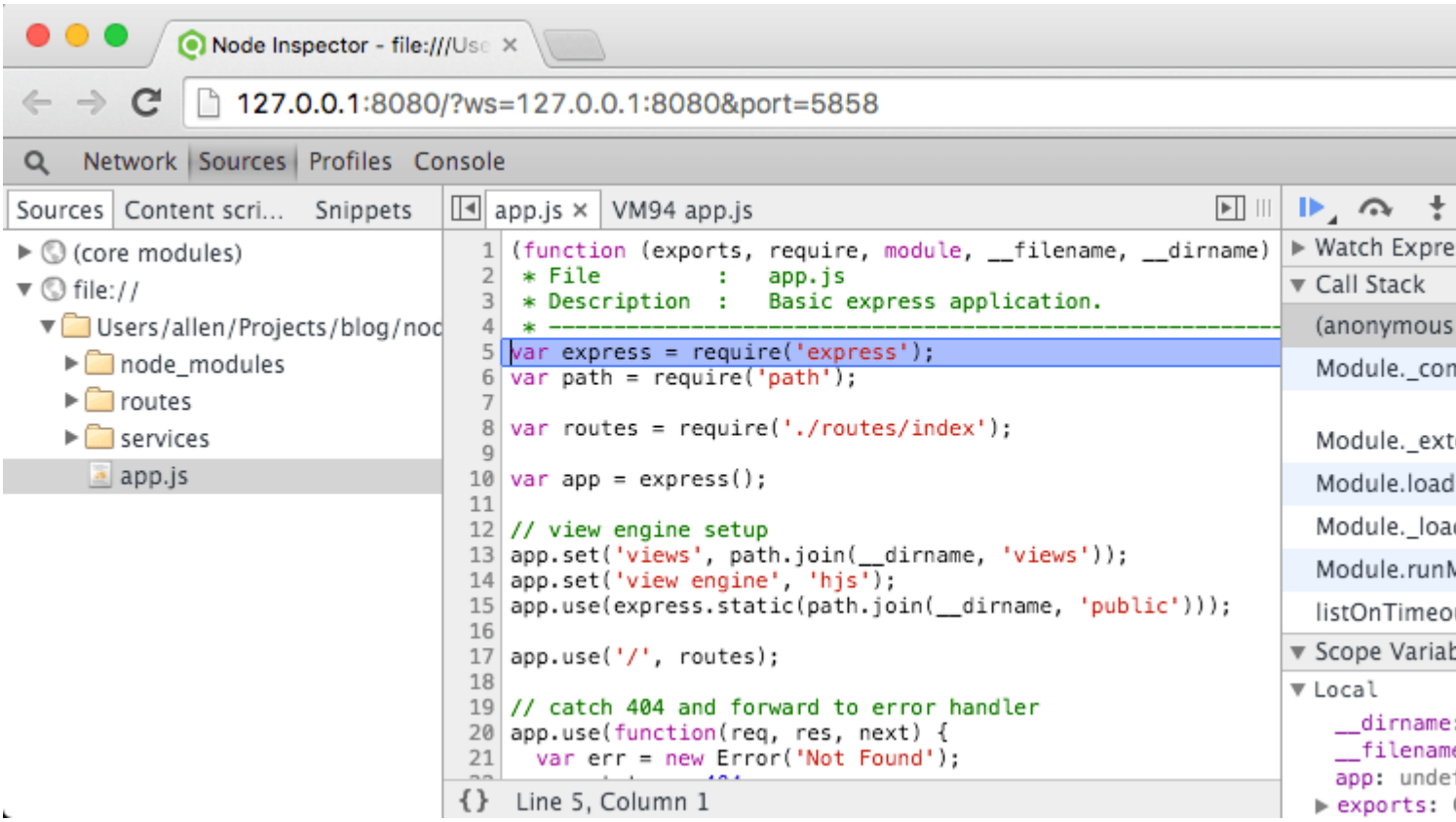
Étape 2 : Démarrez le serveur d'inspecteur de noeud

```
$ node-inspector
```

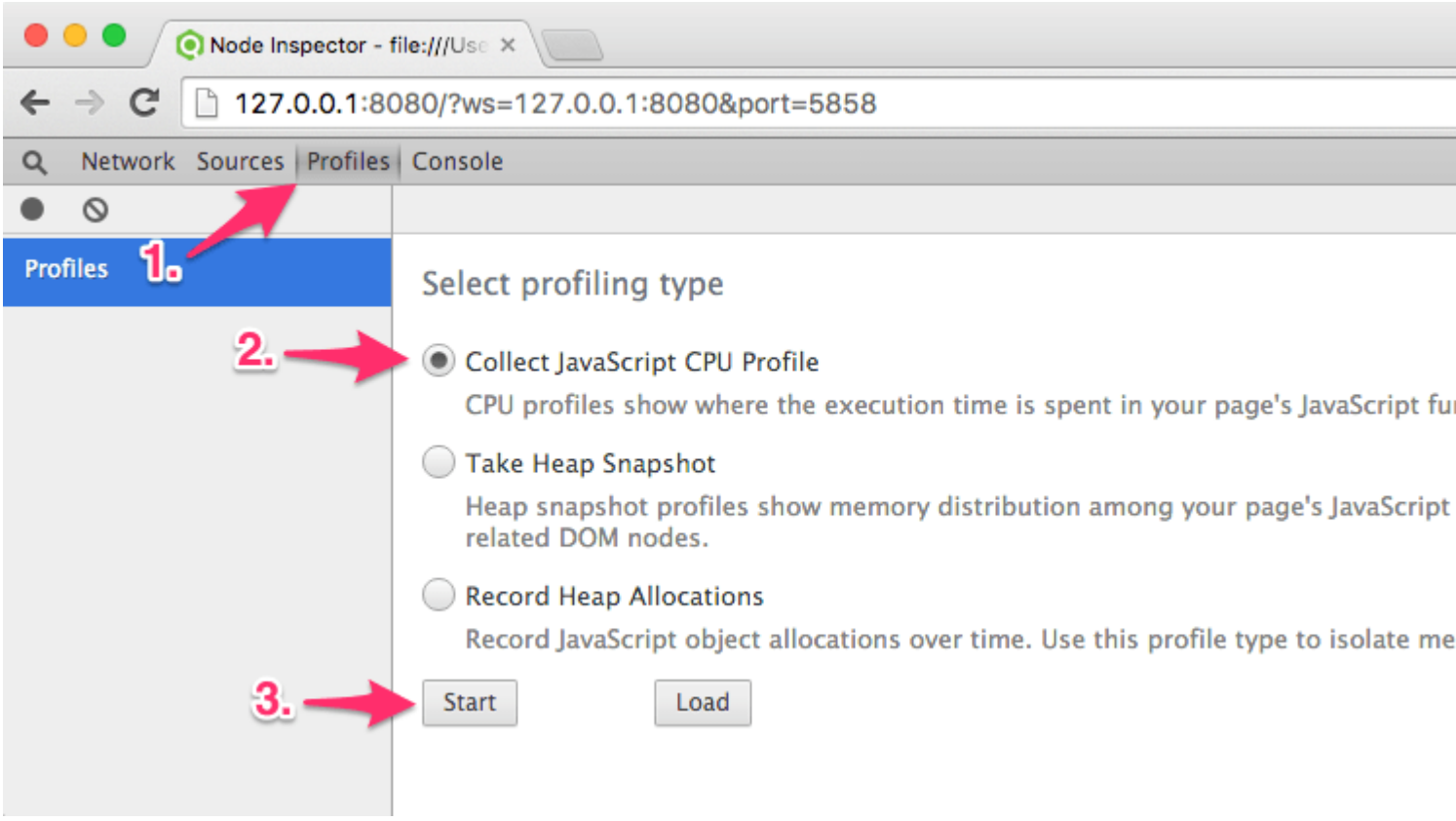
Étape 3 : Commencez à déboguer votre application de noeud

```
$ node --debug-brk your/short/node/script.js
```

Étape 4 : Ouvrez http://127.0.0.1:8080/?port=5858 dans le navigateur Chrome. Et vous verrez une interface d'outils chrom-dev avec le code source de votre application nodejs dans le panneau de gauche. Et comme nous avons utilisé l'option de rupture de débogage lors du débogage de l'application, l'exécution du code s'arrêtera à la première ligne de code.



Étape 5 : C'est la partie la plus simple où vous passez à l'onglet Profiling et commencez à profiler l'application. Si vous voulez obtenir le profil pour une méthode ou un flux particulier, assurez-vous que l'exécution du code est effectuée juste avant que ce morceau de code ne soit exécuté.



Étape 6 : Une fois que vous avez enregistré votre profil de processeur ou votre allocation de mémoire / snapshot ou de tas, vous pouvez afficher les résultats dans la même fenêtre ou les

enregistrer sur le disque local pour une analyse ultérieure ou une comparaison avec d'autres profils.

Vous pouvez utiliser ces articles pour savoir comment lire les profils:

- Lecture de profils de CPU
- Profileur de processeur Chrome et profileur Heap

Lire Prise en main du profilage de nœuds en ligne: https://riptutorial.com/fr/node-js/topic/9347/prise-en-main-du-profilage-de-nouds

Chapitre 95: Programmation asynchrone

Introduction

Node est un langage de programmation où tout peut être exécuté de manière asynchrone. Vous trouverez ci-dessous des exemples et des exemples typiques de travail asynchrone.

Syntaxe

- doQuelque chose ([args], function ([argsCB]) {/ * fait quelque chose quand c'est fait * /});
- doQuelque chose ([args], ([argsCB]) => {/ * fait quelque chose quand c'est fait * /});

Examples

Fonctions de rappel

Fonctions de rappel en JavaScript

Les fonctions de rappel sont communes en JavaScript. Les fonctions de rappel sont possibles en JavaScript car les fonctions sont des citoyens de premier ordre .

Rappels synchrones.

Les fonctions de rappel peuvent être synchrones ou asynchrones. Comme les fonctions de rappel asynchrones peuvent être plus complexes, voici un exemple simple de fonction de rappel synchrone.

```
// a function that uses a callback named `cb` as a parameter
function getSyncMessage(cb) {
    cb("Hello World!");
}

console.log("Before getSyncMessage call");
// calling a function and sending in a callback function as an argument.
getSyncMessage(function(message) {
    console.log(message);
});
console.log("After getSyncMessage call");
```

La sortie du code ci-dessus est:

```
> Before getSyncMessage call
> Hello World!
> After getSyncMessage call
```

Nous allons d'abord passer en revue comment le code ci-dessus est exécuté. C'est plus pour ceux qui ne comprennent pas déjà le concept des rappels si vous comprenez déjà que vous pouvez sauter ce paragraphe. Tout d'abord, le code est analysé, puis la première chose intéressante à faire est que la ligne 6 soit exécutée, ce qui `Before getSyncMessage call` à la console. Ensuite, la ligne 8 est exécutée et appelle la fonction `getSyncMessage` envoyant une fonction anonyme en tant qu'argument du paramètre nommé `cb` dans la fonction `getSyncMessage` . L'exécution est maintenant effectuée dans la fonction `getSyncMessage` de la ligne 3 qui exécute la fonction `cb` qui vient d'être transmise, cet appel envoie une chaîne d'argument "Hello World" pour le `message` nommé param dans la fonction anonyme passée. L'exécution va ensuite à la ligne 9 qui connecte `Hello World!` à la console. Ensuite, l'exécution passe par le processus de sortie de la pile d'appels (voir aussi) en frappant la ligne 10 puis la ligne 4 puis finalement en revenant à la ligne 11.

Quelques informations à connaître sur les rappels en général:

- La fonction que vous envoyez à une fonction en tant que rappel peut être appelée zéro fois, une fois ou plusieurs fois. Tout dépend de la mise en œuvre.
- La fonction de rappel peut être appelée de manière synchrone ou asynchrone et éventuellement de manière synchrone et asynchrone.
- Tout comme les fonctions normales, les noms que vous attribuez aux paramètres de votre fonction ne sont pas importants, mais l'ordre est. Ainsi, par exemple, à la ligne 8, le `message` paramètre pourrait avoir été nommé `statement` , `msg` ou si vous êtes insensé, comme `jellybean` . Vous devez donc savoir quels paramètres sont envoyés dans votre callback pour que vous puissiez les obtenir dans le bon ordre avec des noms propres.

Rappels asynchrones.

Une chose à noter à propos de JavaScript est qu'il est synchrone par défaut, mais il existe des API dans l'environnement (navigateur, Node.js, etc.) qui pourraient le rendre asynchrone (il y a plus à ce sujet ici).

Quelques éléments communs qui sont asynchrones dans les environnements JavaScript qui acceptent les rappels:

- Événements
- setTimeout
- mettreInterval
- l'API d'extraction
- Promesses

De plus, toute fonction qui utilise l'une des fonctions ci-dessus peut être entourée d'une fonction qui prend un rappel et le rappel serait alors un rappel asynchrone (bien que l'encapsulation d'une promesse avec une fonction prenant un rappel serait probablement considérée comme un anti-pattern comme il y a plus de moyens préférés pour gérer les promesses).

Donc, étant donné cette information, nous pouvons construire une fonction asynchrone similaire à celle ci-dessus synchrone.

```
// a function that uses a callback named `cb` as a parameter
function getAsyncMessage(cb) {
    setTimeout(function () { cb("Hello World!") }, 1000);
}

console.log("Before getSyncMessage call");
// calling a function and sending in a callback function as an argument.
getAsyncMessage(function(message) {
    console.log(message);
});
console.log("After getSyncMessage call");
```

Qui imprime les éléments suivants sur la console:

```
> Before getSyncMessage call
> After getSyncMessage call
// pauses for 1000 ms with no output
> Hello World!
```

L'exécution de ligne passe à la ligne 6 des journaux "Avant l'appel getSyncMessage". Ensuite, l'exécution va à la ligne 8 appelant getAsyncMessage avec un rappel pour le paramètre `cb` . La ligne 3 est alors exécutée et appelle setTimeout avec un rappel comme premier argument et le numéro 300 comme second argument. `setTimeout` fait tout ce qu'il fait et retient ce rappel pour pouvoir l'appeler plus tard en 1000 millisecondes, mais après avoir configuré le délai d'attente et avant de suspendre les 1000 millisecondes, il renvoie l'exécution à la ligne 4 , puis la ligne 11, puis s'arrête pendant 1 seconde et setTimeout appelle alors sa fonction de rappel qui `getAsyncMessages` exécution sur la ligne 3 où le rappel `getAsyncMessages` est appelé avec la valeur "Hello World" pour son `message` paramètre qui est ensuite consigné sur la ligne 9 .

Fonctions de rappel dans Node.js

NodeJS a des rappels asynchrones et fournit généralement deux paramètres à vos fonctions, parfois appelés de manière conventionnelle `err` et `data` . Un exemple avec la lecture d'un fichier texte.

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

fs.readFile("./test.txt", "utf8", function(err, data) {
    if(err) {
        // handle the error
    } else {
        // process the file text given with data
    }
});
```

Ceci est un exemple de rappel appelé une seule fois.

Il est recommandé de gérer l'erreur d'une manière ou d'une autre, même si vous ne faites que la connecter ou la lancer. Le reste n'est pas nécessaire si vous lancez ou renvoyez et que vous pouvez le supprimer pour diminuer l'indentation tant que vous arrêtez l'exécution de la fonction en cours dans le if en faisant quelque chose comme lancer ou retourner.

Bien qu'il soit fréquent de voir `err` , les `data` ne peut pas toujours être le cas que vos callbacks utiliseront ce modèle , il est préférable de regarder la documentation.

Un autre exemple de rappel provient de la bibliothèque express (express 4.x):

```
// this code snippet was on http://expressjs.com/en/4x/api.html
const express = require('express');
const app = express();

// this app.get method takes a url route to watch for and a callback
// to call whenever that route is requested by a user.
app.get('/', function(req, res){
  res.send('hello world');
});

app.listen(3000);
```

Cet exemple montre un rappel appelé plusieurs fois. Le rappel est fourni avec deux objets en tant que paramètres nommés ici comme `req` et `res` ces noms correspondent respectivement à la requête et à la réponse, et ils permettent de visualiser la requête et de définir la réponse qui sera envoyée à l'utilisateur.

Comme vous pouvez le voir, il existe différentes manières d’utiliser un rappel pour exécuter la synchronisation et le code asynchrone en JavaScript et les rappels sont très répandus dans JavaScript.

Exemple de code

Question: Quelle est la sortie du code ci-dessous et pourquoi?

```
setTimeout(function() {
    console.log("A");
}, 1000);

setTimeout(function() {
    console.log("B");
}, 0);

getDataFromDatabase(function(err, data) {
    console.log("C");
    setTimeout(function() {
        console.log("D");
    }, 1000);
});

console.log("E");
```

Sortie: Ceci est connu avec certitude: `EBAD . C` est inconnu quand il sera enregistré.

Explication: Le compilateur ne s'arrêtera pas sur les méthodes `setTimeout` et `getDataFromDatabase` . Donc, la première ligne qu'il enregistrera est `E` Les fonctions de rappel *(premier argument de `setTimeout`)* seront exécutées après le délai d'attente défini de manière asynchrone!

Plus de détails:

1. `E` n'a pas de `setTimeout`
2. `B` a un délai d'attente de 0 milliseconde
3. `A` a un délai d'attente de 1000 millisecondes
4. `D` doit demander une base de données, après il doit `D` attendre 1000 millisecondes il vient après `A` .
5. `C` est inconnu car il est inconnu lorsque les données de la base de données sont demandées. Cela pourrait être avant ou après `A`

Traitement d'erreur asynchrone

Essayer attraper

Les erreurs doivent toujours être traitées. Si vous utilisez une programmation synchrone, vous pouvez utiliser un `try catch` . Mais cela ne fonctionne pas si vous travaillez asynchrone! Exemple:

```
try {
    setTimeout(function() {
        throw new Error("I'm an uncaught error and will stop the server!");
    }, 100);
}
catch (ex) {
    console.error("This error will not be work in an asynchronous situation: " + ex);
}
```

Les erreurs asynchrones ne seront traitées que dans la fonction de rappel!

Possibilités de travail

v0.8

Gestionnaires d'événements

Les premières versions de Node.JS ont un gestionnaire d'événement.

```
process.on("UncaughtException", function(err, data) {
    if (err) {
        // error handling
    }
});
```

v0.8

Domaines

Dans un domaine, les erreurs sont libérées via les émetteurs d'événements. En utilisant ceci, il y a toutes les erreurs, les temporisateurs, les méthodes de rappel implicitement seulement

enregistrées dans le domaine. Par une erreur, être un événement d'erreur envoyé et ne pas planter l'application.

```
var domain = require("domain");
var d1 = domain.create();
var d2 = domain.create();

d1.run(function() {
    d2.add(setTimeout(function() {
        throw new Error("error on the timer of domain 2");
    }, 0));
});

d1.on("error", function(err) {
    console.log("error at domain 1: " + err);
});

d2.on("error", function(err) {
    console.log("error at domain 2: " + err);
});
```

Enfer de rappel

Callback hell (également une pyramide d'effet doom ou boomerang) se produit lorsque vous imbriquez trop de fonctions de rappel dans une fonction de rappel. Voici un exemple pour lire un fichier (dans ES6).

```
const fs = require('fs');
let filename = `${__dirname}/myfile.txt`;

fs.exists(filename, exists => {
    if (exists) {
        fs.stat(filename, (err, stats) => {
            if (err) {
                throw err;
            }
            if (stats.isFile()) {
                fs.readFile(filename, null, (err, data) => {
                    if (err) {
                        throw err;
                    }
                    console.log(data);
                });
            }
            else {
                throw new Error("This location contains not a file");
            }
        });
    }
    else {
        throw new Error("404: file not found");
    }
});
```

Comment éviter "Callback Hell"

Il est recommandé de ne pas imbriquer plus de 2 fonctions de rappel. Cela vous aidera à

maintenir la lisibilité du code et sera beaucoup plus facile à maintenir à l'avenir. Si vous devez imbriquer plus de 2 rappels, essayez plutôt d'utiliser des événements distribués .

Il existe également une bibliothèque appelée async qui aide à gérer les callbacks et leur exécution disponible sur npm. Il améliore la lisibilité du code de rappel et vous permet de mieux contrôler le flux de votre code de rappel, en vous permettant notamment de les exécuter en parallèle ou en série.

Promesses autochtones

v6.0.0

Les promesses sont un outil de programmation asynchrone. En JavaScript, les promesses sont connues pour leurs méthodes `then` . Les promesses ont deux états principaux «en attente» et «réglé». Une fois qu'une promesse est «réglée», elle ne peut plus être «en attente». Cela signifie que les promesses sont généralement bonnes pour les événements qui ne se produisent qu'une seule fois. L'état "réglé" a deux états aussi "résolu" et "rejeté". Vous pouvez créer une nouvelle promesse en utilisant le `new` mot-clé et en passant une fonction dans le constructeur `new Promise(function (resolve, reject) {})` .

La fonction transmise au constructeur Promise reçoit toujours un premier et un deuxième paramètre, généralement nommés `resolve` et `reject` respectivement. La désignation de ces deux paramètres est la convention, mais ils mettront la promesse soit dans l'état «résolu», soit dans l'état «rejeté». Lorsque l'une d'elles est appelée, la promesse passe de «en attente» à «réglé». `resolve` est appelée lorsque l'action désirée, qui est souvent asynchrone, a été effectuée et que le `reject` est utilisé si l'action a été erronée.

Dans le délai ci-dessous, une fonction renvoie une promesse.

```
function timeout (ms) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve("It was resolved!");
    }, ms)
  });
}

timeout(1000).then(function (dataFromPromise) {
  // logs "It was resolved!"
  console.log(dataFromPromise);
})

console.log("waiting...");
```

sortie de la console

```
waiting...
// << pauses for one second>>
It was resolved!
```

Lorsque la temporisation est appelée, la fonction transmise au constructeur Promise est exécutée

sans délai. Ensuite, la méthode setTimeout est exécutée et son rappel est défini pour se déclencher dans les `ms` millisecondes suivantes, dans ce cas, `ms=1000` . Comme le rappel de setTimeout n'est pas déclenché, la fonction de délai d'attente renvoie le contrôle à la portée de l'appel. La chaîne d' `then` les méthodes sont ensuite stockées à appeler plus tard , quand / si la promesse a résolu. S'il y avait `catch` méthodes de `catch` ici, elles seraient également stockées, mais seraient renvoyées quand / si la promesse est «rejetée».

Le script imprime alors "en attente ...". Une seconde plus tard, setTimeout appelle son rappel qui appelle la fonction resolve avec la chaîne "It a été résolue!". Cette chaîne est ensuite passé dans le `then` de rappel de la méthode et est ensuite connecté à l'utilisateur.

Dans le même sens, vous pouvez encapsuler la fonction asynchrone setTimeout qui nécessite un rappel, vous pouvez envelopper toute action asynchrone singulière avec une promesse.

En savoir plus sur les promesses dans la documentation JavaScript promesses .

Lire Programmation asynchrone en ligne: https://riptutorial.com/fr/node-js/topic/8813/programmation-asynchrone

Chapitre 96: Programmation synchrone vs asynchrone dans nodejs

Examples

Utiliser async

Le paquet asynchrone fournit des fonctions pour le code asynchrone.

En utilisant la fonction automatique , vous pouvez définir des relations asynchrones entre deux fonctions ou plus:

```
var async = require('async');

async.auto({
    get_data: function(callback) {
        console.log('in get_data');
        // async code to get some data
        callback(null, 'data', 'converted to array');
    },
    make_folder: function(callback) {
        console.log('in make_folder');
        // async code to create a directory to store a file in
        // this is run at the same time as getting the data
        callback(null, 'folder');
    },
    write_file: ['get_data', 'make_folder', function(results, callback) {
        console.log('in write_file', JSON.stringify(results));
        // once there is some data and the directory exists,
        // write the data to a file in the directory
        callback(null, 'filename');
    }],
    email_link: ['write_file', function(results, callback) {
        console.log('in email_link', JSON.stringify(results));
        // once the file is written let's email a link to it...
        // results.write_file contains the filename returned by write_file.
        callback(null, {'file':results.write_file, 'email':'user@example.com'});
    }]
}, function(err, results) {
    console.log('err = ', err);
    console.log('results = ', results);
});
```

Ce code aurait pu être fait de manière synchrone, en appelant simplement `get_data` , `make_folder` , `write_file` et `email_link` dans le bon ordre. Async garde une trace des résultats pour vous, et si une erreur survient (premier paramètre de `callback` inégal à `null`), elle arrête l'exécution des autres fonctions.

Lire Programmation synchrone vs asynchrone dans nodejs en ligne: https://riptutorial.com/fr/node-js/topic/8287/programmation-synchrone-vs-asynchrone-dans-nodejs

Chapitre 97: Rappel à la promesse

Examples

Promouvoir un rappel

Rappel basé sur:

```
db.notification.email.find({subject: 'promisify callback'}, (error, result) => {
   if (error) {
       console.log(error);
   }

   // normal code here
});
```

Ceci utilise la méthode promisifyAll de bluebird pour indiquer ce qui est classiquement un code basé sur le rappel comme ci-dessus. bluebird fera une version prometteuse de toutes les méthodes de l'objet, les noms des méthodes basées sur les promesses ont été ajoutés à Async:

```
let email = bluebird.promisifyAll(db.notification.email);

email.findAsync({subject: 'promisify callback'}).then(result => {

    // normal code here
})
.catch(console.error);
```

Si seules des méthodes spécifiques doivent être promises, il suffit d'utiliser sa promesse:

```
let find = bluebird.promisify(db.notification.email.find);

find({locationId: 168}).then(result => {

    // normal code here
});
.catch(console.error);
```

Certaines bibliothèques (par exemple, MassiveJS) ne peuvent pas être promises si l'objet immédiat de la méthode n'est pas transmis au second paramètre. Dans ce cas, il suffit de transmettre l'objet immédiat de la méthode qui doit être promis sur le deuxième paramètre et de le placer dans la propriété contextuelle.

```
let find = bluebird.promisify(db.notification.email.find, { context: db.notification.email });

find({locationId: 168}).then(result => {

    // normal code here
});
.catch(console.error);
```

Promouvoir manuellement un rappel

Parfois, il peut être nécessaire de promouvoir manuellement une fonction de rappel. Cela pourrait être pour un cas où le rappel ne suit pas le format standard erreur-first ou si une logique supplémentaire est nécessaire pour indiquer:

Exemple avec fs.exists (path, callback) :

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

var existsAsync = function(path) {
  return new Promise(function(resolve, reject) {
    fs.exists(path, function(exists) {
      // exists is a boolean
      if (exists) {
        // Resolve successfully
        resolve();
      } else {
        // Reject with error
        reject(new Error('path does not exist'));
      }
    });
});

// Use as a promise now
existsAsync('/path/to/some/file').then(function() {
  console.log('file exists!');
}).catch(function(err) {
  // file does not exist
  console.error(err);
});
```

setTimeout promis

```
function wait(ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(resolve, ms)
    })
}
```

Lire Rappel à la promesse en ligne: https://riptutorial.com/fr/node-js/topic/2346/rappel-a-la-promesse

Chapitre 98: Readline

Syntaxe

- const readline = exige ('readline')
- readline.close ()
- readline.pause ()
- readline.prompt ([preserveCursor])
- readline.question (requête, rappel)
- readline.resume ()
- readline.setPrompt (prompt)
- readline.write (data [, clé])
- readline.clearLine (stream, dir)
- readline.clearScreenDown (stream)
- readline.createInterface (options)
- readline.cursorTo (flux, x, y)
- readline.emitKeypressEvents (flux [, interface])
- readline.moveCursor (flux, dx, dy)

Examples

Lecture de fichier ligne par ligne

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

const rl = readline.createInterface({
    input: fs.createReadStream('text.txt')
});

// Each new line emits an event - every time the stream receives \r, \n, or \r\n
rl.on('line', (line) => {
    console.log(line);
});

rl.on('close', () => {
    console.log('Done reading file');
});
```

Inviter l'utilisateur à entrer via l'interface de ligne de commande

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

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question('What is your name?', (name) => {
```

```
    console.log(`Hello ${name}!`);

    rl.close();
});
```

Lire Readline en ligne: https://riptutorial.com/fr/node-js/topic/1431/readline

Chapitre 99: Routage des requêtes ajax avec Express.JS

Examples

Une implémentation simple d'AJAX

Vous devriez avoir le modèle de générateur express de base

Dans app.js, ajoutez (vous pouvez l'ajouter n'importe où après `var app = express.app()`):

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

Maintenant, dans votre fichier index.js (ou sa correspondance respective), ajoutez:

```
router.get('/ajax', function(req, res){
    res.render('ajax', {title: 'An Ajax Example', quote: "AJAX is great!"});
});
router.post('/ajax', function(req, res){
    res.render('ajax', {title: 'An Ajax Example', quote: req.body.quote});
});
```

Créez un `ajax.jade` / `ajax.pug` ou `ajax.ejs` dans le répertoire `/views` , ajoutez:

Pour Jade / PugJS:

```
extends layout
script(src="http://code.jquery.com/jquery-3.1.0.min.js")
script(src="/magic.js")
h1 Quote: !{quote}
form(method="post" id="changeQuote")
    input(type='text', placeholder='Set quote of the day', name='quote')
    input(type="submit", value="Save")
```

Pour EJS:

```
<script src="http://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="/magic.js"></script>
<h1>Quote: <%=quote%> </h1>
<form method="post" id="changeQuote">
    <input type="text" placeholder="Set quote of the day" name="quote"/>
    <input type="submit" value="Save">
</form>
```

Maintenant, créez un fichier dans `/public` appelé `magic.js`

```
$(document).ready(function(){
```

```
    $("form#changeQuote").on('submit', function(e){
        e.preventDefault();
        var data = $('input[name=quote]').val();
        $.ajax({
            type: 'post',
            url: '/ajax',
            data: data,
            dataType: 'text'
        })
        .done(function(data){
            $('h1').html(data.quote);
        });
    });
});
```

Et voila! Lorsque vous cliquez sur Enregistrer, le devis change!

Lire Routage des requêtes ajax avec Express.JS en ligne: https://riptutorial.com/fr/node-js/topic/6738/routage-des-requetes-ajax-avec-express-js

Chapitre 100: Sécurisation des applications Node.js

Examples

Prévention de la contrefaçon de requêtes inter-sites (CSRF)

CSRF est une attaque qui oblige l'utilisateur final à exécuter des actions indésirables sur une application Web dans laquelle il est actuellement authentifié.

Cela peut se produire parce que les cookies sont envoyés avec chaque demande à un site Web - même lorsque ces demandes proviennent d'un autre site.

Nous pouvons utiliser le module `csurf` pour créer un jeton csrf et le valider.

Exemple

```
var express = require('express')
var cookieParser = require('cookie-parser')    //for cookie parsing
var csrf = require('csurf')    //csrf module
var bodyParser = require('body-parser')    //for body parsing

// setup route middlewares
var csrfProtection = csrf({ cookie: true })
var parseForm = bodyParser.urlencoded({ extended: false })

// create express app
var app = express()

// parse cookies
app.use(cookieParser())

app.get('/form', csrfProtection, function(req, res) {
  // generate and pass the csrfToken to the view
  res.render('send', { csrfToken: req.csrfToken() })
})

app.post('/process', parseForm, csrfProtection, function(req, res) {
  res.send('data is being processed')
})
```

Ainsi, lorsque nous `csrfToken` à `GET /form` , il transmettra le jeton `csrfToken` à la vue.

Maintenant, dans la vue, définissez la valeur csrfToken comme valeur d'un champ d'entrée masqué nommé `_csrf` .

par exemple pour les modèles de `handlebar`

```
<form action="/process" method="POST">
    <input type="hidden" name="_csrf" value="{{csrfToken}}">
    Name: <input type="text" name="name">
```

```
    <button type="submit">Submit</button>
</form>
```

par exemple pour les modèles de `jade`

```
form(action="/process" method="post")
    input(type="hidden", name="_csrf", value=csrfToken)

    span Name:
    input(type="text", name="name", required=true)
    br

    input(type="submit")
```

par exemple pour les modèles `ejs`

```
<form action="/process" method="POST">
    <input type="hidden" name="_csrf" value="<%= csrfToken %>">
    Name: <input type="text" name="name">
    <button type="submit">Submit</button>
</form>
```

SSL / TLS dans Node.js

Si vous choisissez de gérer SSL / TLS dans votre application Node.js, considérez que vous êtes également responsable du maintien de la prévention des attaques SSL / TLS à ce stade. Dans de nombreuses architectures serveur-client, SSL / TLS se termine sur un proxy inverse, à la fois pour réduire la complexité des applications et pour réduire la portée de la configuration de la sécurité.

Si votre application Node.js doit gérer SSL / TLS, elle peut être sécurisée en chargeant les fichiers clé et cert.

Si votre fournisseur de certificats requiert une chaîne d'autorité de certification, il peut être ajouté dans l'option `ca` tant que tableau. Une chaîne avec plusieurs entrées dans un seul fichier doit être divisée en plusieurs fichiers et saisie dans le même ordre dans le tableau car Node.js ne prend actuellement pas en charge plusieurs entrées dans un fichier. Un exemple est fourni dans le code ci-dessous pour les fichiers `1_ca.crt` et `2_ca.crt` . Si le tableau `ca` est requis et n'est pas défini correctement, les navigateurs clients peuvent afficher des messages qu'ils n'ont pas pu vérifier l'authenticité du certificat.

Exemple

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

const options = {
  key: fs.readFileSync('privatekey.pem'),
  cert: fs.readFileSync('certificate.pem'),
  ca: [fs.readFileSync('1_ca.crt'), fs.readFileSync('2_ca.crt')]
};

https.createServer(options, (req, res) => {
```

```
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

Utiliser HTTPS

La configuration minimale pour un serveur HTTPS dans Node.js serait la suivante:

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

const httpsOptions = {
    key: fs.readFileSync('path/to/server-key.pem'),
    cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

https.createServer(httpsOptions, app).listen(4433);
```

Si vous souhaitez également prendre en charge les requêtes http, vous devez apporter cette petite modification:

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

const httpsOptions = {
    key: fs.readFileSync('path/to/server-key.pem'),
    cert: fs.readFileSync('path/to/server-crt.pem')
};

const app = function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}

http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4433);
```

Configuration d'un serveur HTTPS

Une fois que node.js est installé sur votre système, suivez la procédure ci-dessous pour obtenir un serveur Web de base compatible avec HTTP et HTTPS!

Étape 1: créer une autorité de certification

1. Créez le dossier dans lequel vous souhaitez stocker votre clé et votre certificat:

   ```
   mkdir conf
   ```

2. allez dans ce répertoire:

```
cd conf
```

---

3. récupérer ce fichier `ca.cnf` à utiliser comme raccourci de configuration:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-clients/master/keys/ca.cnf
```

---

4. créer une nouvelle autorité de certification en utilisant cette configuration:

```
openssl req -new -x509 -days 9999 -config ca.cnf -keyout ca-key.pem -out ca-cert.pem
```

---

5. Maintenant que nous avons notre autorité de certification dans `ca-key.pem` et `ca-cert.pem` , générons une clé privée pour le serveur:

```
openssl genrsa -out key.pem 4096
```

---

6. récupérer ce fichier `server.cnf` à utiliser comme raccourci de configuration:

```
wget https://raw.githubusercontent.com/anders94/https-authorized-
clients/master/keys/server.cnf
```

---

7. générer la demande de signature de certificat en utilisant cette configuration:

```
openssl req -new -config server.cnf -key key.pem -out csr.pem
```

---

8. signer la demande:

```
openssl x509 -req -extfile server.cnf -days 999 -passin "pass:password" -in csr.pem -CA ca-
cert.pem -CAkey ca-key.pem -CAcreateserial -out cert.pem
```

Étape 2: installez votre certificat en tant que certificat racine

1. copiez votre certificat dans le dossier de vos certificats racine:

```
sudo cp ca-crt.pem /usr/local/share/ca-certificates/ca-crt.pem
```

---

2. mettre à jour le magasin CA:

```
sudo update-ca-certificates
```

Application express.js sécurisée 3

La configuration pour établir une connexion sécurisée avec express.js (depuis la version 3):

```
var fs = require('fs');
var http = require('http');
```

```
var https = require('https');
var privateKey  = fs.readFileSync('sslcert/server.key', 'utf8');
var certificate = fs.readFileSync('sslcert/server.crt', 'utf8');

// Define your key and cert

var credentials = {key: privateKey, cert: certificate};
var express = require('express');
var app = express();

// your express configuration here

var httpServer = http.createServer(app);
var httpsServer = https.createServer(credentials, app);

// Using port 8080 for http and 8443 for https

httpServer.listen(8080);
httpsServer.listen(8443);
```

De cette manière, vous fournissez un middleware express au serveur natif http / https

Si vous souhaitez que votre application s'exécute sur des ports inférieurs à 1024, vous devrez utiliser la commande sudo (non recommandée) ou utiliser un proxy inverse (par exemple, nginx, haproxy).

Lire Sécurisation des applications Node.js en ligne: https://riptutorial.com/fr/node-js/topic/3473/securisation-des-applications-node-js

Chapitre 101: Sequelize.js

Examples

Installation

Assurez-vous que Node.js et npm sont installés en premier. Ensuite, installez sequelize.js avec npm

```
npm install --save sequelize
```

Vous devrez également installer les modules de base de données Node.js pris en charge. Il vous suffit d'installer celui que vous utilisez

Pour `MYSQL` et `Mariadb`

```
npm install --save mysql
```

Pour `PostgreSQL`

```
npm install --save pg pg-hstore
```

Pour `SQLite`

```
 npm install --save sqlite
```

Pour `MSSQL`

```
npm install --save tedious
```

Une fois que vous avez configuré l'installation, vous pouvez inclure et créer une nouvelle instance Sequalize comme celle-ci.

Syntaxe ES5

```
var Sequelize = require('sequelize');
var sequelize = new Sequelize('database', 'username', 'password');
```

ES6 stage-0 syntaxe Babel

```
import Sequelize from 'sequelize';
const sequelize = new Sequelize('database', 'username', 'password');
```

Vous avez maintenant une instance de sequelize disponible. Vous pourriez si vous vous sentez tellement enclin à l'appeler un autre nom tel que

```
var db = new Sequelize('database', 'username', 'password');
```

ou

```
var database = new Sequelize('database', 'username', 'password');
```

cette partie est votre prérogative. Une fois que vous avez installé cela, vous pouvez l'utiliser à l'intérieur de votre application selon la documentation de l'API http://docs.sequelizejs.com/en/v3/api/sequelize/

Votre prochaine étape après l'installation serait de configurer votre propre modèle

Définir des modèles

Il y a deux manières de définir des modèles dans la suite; avec `sequelize.define(...)` , ou `sequelize.import(...)` . Les deux fonctions renvoient un objet modèle sequelize.

1. sequelize.define (modelName, attributes, [options])

C'est la voie à suivre si vous souhaitez définir tous vos modèles dans un seul fichier ou si vous souhaitez contrôler davantage la définition de votre modèle.

```
/* Initialize Sequelize */
const config = {
    username: "database username",
    password: "database password",
    database: "database name",
    host: "database's host URL",
    dialect: "mysql" // Other options are postgres, sqlite, mariadb and mssql.
}
var Sequelize = require("sequelize");
var sequelize = new Sequelize(config);

/* Define Models */
sequelize.define("MyModel", {
    name: Sequelize.STRING,
    comment: Sequelize.TEXT,
    date: {
        type: Sequelize.DATE,
        allowNull: false
    }
});
```

Pour la documentation et d'autres exemples, consultez la documentation de doclets ou la documentation de sequelize.com .

2. sequelize.import (chemin)

Si les définitions de votre modèle sont divisées en un fichier pour chacune, alors l' `import` est votre ami. Dans le fichier où vous initialisez Sequelize, vous devez appeler l'importation comme suit:

```
/* Initialize Sequelize */
// Check previous code snippet for initialization

/* Define Models */
sequelize.import("./models/my_model.js"); // The path could be relative or absolute
```

Ensuite, dans vos fichiers de définition de modèle, votre code ressemblera à ceci:

```
module.exports = function(sequelize, DataTypes) {
    return sequelize.define("MyModel", {
        name: DataTypes.STRING,
        comment: DataTypes.TEXT,
        date: {
            type: DataTypes.DATE,
            allowNull: false
        }
    });
};
```

Pour plus d'informations sur l'utilisation de l' `import` , consultez l' exemple express de sequelize sur GitHub .

Lire Sequelize.js en ligne: https://riptutorial.com/fr/node-js/topic/7705/sequelize-js

Chapitre 102: Socket.io communication

Examples

"Bonjour le monde!" avec des messages de socket.

Installer des modules de noeud

```
npm install express
npm install socket.io
```

Serveur Node.js

```
const express = require('express');
const app = express();
const server = app.listen(3000,console.log("Socket.io Hello World server started!"));
const io = require('socket.io')(server);

io.on('connection', (socket) => {
    //console.log("Client connected!");
    socket.on('message-from-client-to-server', (msg) => {
        console.log(msg);
    })
    socket.emit('message-from-server-to-client', 'Hello World!');
});
```

Client de navigateur

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Hello World with Socket.io</title>
  </head>
  <body>
    <script src="https://cdn.socket.io/socket.io-1.4.5.js"></script>
    <script>
      var socket = io("http://localhost:3000");
      socket.on("message-from-server-to-client", function(msg) {
          document.getElementById('message').innerHTML = msg;
      });
      socket.emit('message-from-client-to-server', 'Hello World!');
    </script>
    <p>Socket.io Hello World client started!</p>
    <p id="message"></p>
  </body>
</html>
```

Lire Socket.io communication en ligne: https://riptutorial.com/fr/node-js/topic/4261/socket-io-communication

Chapitre 103: Sockets TCP

Examples

Un simple serveur TCP

```
// Include Nodejs' net module.
const Net = require('net');
// The port on which the server is listening.
const port = 8080;

// Use net.createServer() in your code. This is just for illustration purpose.
// Create a new TCP server.
const server = new Net.Server();
// The server listens to a socket for a client to make a connection request.
// Think of a socket as an end point.
server.listen(port, function() {
    console.log(`Server listening for connection requests on socket localhost:${port}`.);
});

// When a client requests a connection with the server, the server creates a new
// socket dedicated to that client.
server.on('connection', function(socket) {
    console.log('A new connection has been established.');

    // Now that a TCP connection has been established, the server can send data to
    // the client by writing to its socket.
    socket.write('Hello, client.');

    // The server can also receive data from the client by reading from its socket.
    socket.on('data', function(chunk) {
        console.log(`Data received from client: ${chunk.toString()`.});
    });

    // When the client requests to end the TCP connection with the server, the server
    // ends the connection.
    socket.on('end', function() {
        console.log('Closing connection with the client');
    });

    // Don't forget to catch error, for your own sake.
    socket.on('error', function(err) {
        console.log(`Error: ${err}`);
    });
});
```

Un simple client TCP

```
// Include Nodejs' net module.
const Net = require('net');
// The port number and hostname of the server.
const port = 8080;
const host = 'localhost';

// Create a new TCP client.
```

```
const client = new Net.Socket();
// Send a connection request to the server.
client.connect({ port: port, host: host }), function() {
    // If there is no error, the server has accepted the request and created a new
    // socket dedicated to us.
    console.log('TCP connection established with the server.');

    // The client can now send data to the server by writing to its socket.
    client.write('Hello, server.');
});

// The client can also receive data from the server by reading from its socket.
client.on('data', function(chunk) {
    console.log(`Data received from the server: ${chunk.toString()}.`);

    // Request an end to the connection after the data has been received.
    client.end();
});

client.on('end', function() {
    console.log('Requested an end to the TCP connection');
});
```

Lire Sockets TCP en ligne: https://riptutorial.com/fr/node-js/topic/6545/sockets-tcp

Chapitre 104: Structure du projet

Introduction

La structure du projet nodejs est influencée par les préférences personnelles, l'architecture du projet et la stratégie d'injection de module utilisée. Également sur l'événement basé sur un mécanisme d'instanciation de module dynamique. Pour avoir une structure MVC, il est impératif de séparer le code source côté serveur et le code source côté client car le code côté client sera probablement réduit et envoyé au navigateur et est de nature publique. Et le côté serveur ou back-end fournira une API pour effectuer des opérations CRUD

Remarques

Le projet ci-dessus utilise les modules browserify et vue.js comme vue de base de l'application et comme bibliothèques de minification. Ainsi, la structure du projet peut être modifiée avec précision en fonction du framework mvc que vous utilisez, par exemple le répertoire de compilation en public devra contenir tout le code mvc. Vous pouvez avoir une tâche qui le fait pour vous.

Examples

Une simple application nodejs avec MVC et API

- La première distinction majeure se situe entre les répertoires générés dynamiquement qui seront utilisés pour l'hébergement et les répertoires sources.
- Les répertoires source auront un fichier ou un dossier de configuration en fonction de la quantité de configuration que vous pourriez avoir. Cela inclut la configuration de l'environnement et la configuration de la logique métier que vous pouvez choisir de placer dans le répertoire de configuration.

```
|-- Config
       |-- config.json
       |-- appConfig
           |-- pets.config
           |-- payment.config
```

- Maintenant, les répertoires les plus vitaux où l'on distingue le côté serveur / backend et les modules frontend. Les 2 répertoires *serveur* et *webapp* représentent respectivement le backend et le frontend que nous pouvons choisir de placer dans un répertoire source à savoir. *src* .

  > Vous pouvez choisir des noms différents pour chaque serveur ou application Web en fonction de ce qui vous convient. Assurez-vous que vous ne voulez pas le rendre trop long ou trop complexe car il est en fin de compte dans la structure interne du projet.

- Dans le répertoire du *serveur* , vous pouvez avoir le contrôleur, le fichier App.js / index.js qui sera votre fichier principal nodejs et le point de départ. Le répertoire du serveur. peut également avoir le *répertoire dto* qui contient tous les objets de transfert de données qui seront utilisés par les contrôleurs API.

```
|-- server
     |-- dto
         |-- pet.js
         |-- payment.js
     |-- controller
         |-- PetsController.js
         |-- PaymentController.js
     |-- App.js
```

- Le répertoire webapp peut être divisé en deux parties: *public* et *mvc* , ceci est à nouveau influencé par la stratégie de construction que vous souhaitez utiliser. Nous utilisons browserfiy pour créer la partie MVC de webapp et minimiser le contenu du répertoire *mvc* simplement.

  | - webapp | - public | - mvc

- Maintenant, le répertoire public peut contenir toutes les ressources statiques, images, css (vous pouvez également avoir des fichiers saas) et surtout les fichiers HTML.

```
|-- public
    |-- build  // will contianed minified scripts(mvc)
    |-- images
        |-- mouse.jpg
        |-- cat.jpg
    |-- styles
        |-- style.css
    |-- views
        |-- petStore.html
        |-- paymentGateway.html
        |-- header.html
        |-- footer.html
    |-- index.html
```

- Le répertoire *mvc* contiendra la logique frontale incluant les *modèles* , les *contrôleurs de vue* et tout autre module *utils dont* vous pourriez avoir besoin dans le cadre de l'interface utilisateur. De même, index.js ou shell.js, selon votre choix, font également partie de ce répertoire.

```
|-- mvc
    |-- controllers
        |-- Dashborad.js
        |-- Help.js
        |-- Login.js
    |-- utils
    |-- index.js
```

Donc , en conclusion toute la structure du projet ressemblera below.And une tâche simple de construction comme *gulp browserify* va rapetisser les scripts et publier dans mvc répertoire *public.*

Nous pouvons alors fournir ce répertoire public en tant que ressource statique via ***express.use (satic ('public'))*** api.

```
    |-- node_modules
    |-- src
        |-- server
            |-- controller
            |-- App.js   // node app
        |-- webapp
            |-- public
                |-- styles
                |-- images
                |-- index.html
            |-- mvc
                |-- controller
                |-- shell.js  // mvc shell
    |-- config
    |-- Readme.md
    |-- .gitignore
    |-- package.json
```

Lire Structure du projet en ligne: https://riptutorial.com/fr/node-js/topic/9935/structure-du-projet

Chapitre 105: Structure Route-Controller-Service pour ExpressJS

Examples

Structure du répertoire des modèles de routes, des contrôleurs et des services

```
├──models
|   ├──user.model.js
├──routes
|   ├──user.route.js
├──services
|   ├──user.service.js
├──controllers
|   ├──user.controller.js
```

Pour la structure de code modulaire, la logique doit être divisée en ces répertoires et fichiers.

Models - La définition de schéma du modèle

Routes - L'API route les cartes vers les contrôleurs

Contrôleurs - Les contrôleurs gèrent toute la logique de la validation des paramètres de requête, de la requête, des réponses d'envoi avec les codes corrects.

Services - Les services contiennent les requêtes de la base de données et le retour des objets ou des erreurs de lancement

Ce codeur finira par écrire plus de codes. Mais à la fin, les codes seront beaucoup plus faciles à maintenir et à séparer.

Structure du code Model-Routes-Controllers-Services

user.model.js

```
var mongoose = require('mongoose')

const UserSchema  = new mongoose.Schema({
    name: String
})

const User = mongoose.model('User', UserSchema)

module.exports = User;
```

user.routes.js

```
var express = require('express');
var router = express.Router();

var UserController = require('../controllers/user.controller')

router.get('/', UserController.getUsers)

module.exports = router;
```

user.controllers.js

```
var UserService = require('../services/user.service')

exports.getUsers = async function (req, res, next) {
    // Validate request parameters, queries using express-validator

    var page = req.params.page ? req.params.page : 1;
    var limit = req.params.limit ? req.params.limit : 10;
    try {
        var users = await UserService.getUsers({}, page, limit)
        return res.status(200).json({ status: 200, data: users, message: "Succesfully Users
Retrieved" });
    } catch (e) {
        return res.status(400).json({ status: 400, message: e.message });
    }
}
```

user.services.js

```
var User = require('../models/user.model')

exports.getUsers = async function (query, page, limit) {

    try {
        var users = await User.find(query)
        return users;
    } catch (e) {
        // Log Errors
        throw Error('Error while Paginating Users')
    }
}
```

Lire Structure Route-Controller-Service pour ExpressJS en ligne: https://riptutorial.com/fr/node-js/topic/10785/structure-route-controller-service-pour-expressjs

Chapitre 106: Système de fichiers I / O

Remarques

Dans Node.js, les opérations gourmandes en ressources telles que les E / S sont effectuées de *manière asynchrone* , mais ont un pendant *synchrone* (par exemple, il existe un `fs.readFile` et son homologue est `fs.readFileSync`). Étant donné que Node est mono-thread, vous devez faire attention lorsque vous utilisez *des* opérations *synchrones* , car elles bloqueront tout le processus.

Si un processus est bloqué par une opération synchrone, tout le cycle d'exécution (y compris la boucle d'événement) est arrêté. Cela signifie qu'aucun autre code asynchrone, y compris les événements et les gestionnaires d'événements, ne s'exécutera pas et que votre programme continuera d'attendre la fin de l'opération de blocage unique.

Il existe des utilisations appropriées pour les opérations synchrones et asynchrones, mais il faut veiller à les utiliser correctement.

Examples

Écrire dans un fichier en utilisant writeFile ou writeFileSync

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

// Save the string "Hello world!" in a file called "hello.txt" in
// the directory "/tmp" using the default encoding (utf8).
// This operation will be completed in background and the callback
// will be called when it is either done or failed.
fs.writeFile('/tmp/hello.txt', 'Hello world!', function(err) {
  // If an error occurred, show it and return
  if(err) return console.error(err);
  // Successfully wrote to the file!
});

// Save binary data to a file called "binary.txt" in the current
// directory. Again, the operation will be completed in background.
var buffer = new Buffer([ 0x48, 0x65, 0x6c, 0x6c, 0x6f ]);
fs.writeFile('binary.txt', buffer, function(err) {
  // If an error occurred, show it and return
  if(err) return console.error(err);
  // Successfully wrote binary contents to the file!
});
```

`fs.writeFileSync` se comporte de la même manière que `fs.writeFile` , mais ne prend pas de rappel car il se termine de manière synchrone et bloque donc le thread principal. La plupart des développeurs node.js préfèrent les variantes asynchrones qui ne causent pratiquement aucun retard dans l'exécution du programme.

Remarque: le blocage du thread principal est une mauvaise pratique dans node.js. La fonction synchrone ne doit être utilisée que lors du débogage ou lorsque aucune autre option n'est

disponible.

```
// Write a string to another file and set the file mode to 0755
try {
  fs.writeFileSync('sync.txt', 'anni', { mode: 0o755 });
} catch(err) {
  // An error occurred
  console.error(err);
}
```

Lecture asynchrone à partir de fichiers

Utilisez le module de système de fichiers pour toutes les opérations sur les fichiers:

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

Avec encodage

Dans cet exemple, lisez `hello.txt` partir du répertoire `/tmp` . Cette opération sera terminée en arrière-plan et le rappel se produit à la fin ou en cas d'échec:

```
fs.readFile('/tmp/hello.txt', { encoding: 'utf8' }, (err, content) => {
  // If an error occurred, output it and return
  if(err) return console.error(err);

  // No error occurred, content is a string
  console.log(content);
});
```

Sans codage

Lisez le fichier binaire `binary.txt` du répertoire en cours, de manière asynchrone en arrière-plan. Notez que nous ne définissons pas l'option "encoding" - cela empêche Node.js de décoder le contenu en une chaîne:

```
fs.readFile('binary', (err, binaryContent) => {
  // If an error occurred, output it and return
  if(err) return console.error(err);

  // No error occurred, content is a Buffer, output it in
  // hexadecimal representation.
  console.log(content.toString('hex'));
});
```

Chemins relatifs

Gardez à l'esprit que, dans le cas général, votre script peut être exécuté avec un répertoire de travail courant arbitraire. Pour adresser un fichier par rapport au script en cours, utilisez `__dirname`

ou `__filename` :

```
fs.readFile(path.resolve(__dirname, 'someFile'), (err, binaryContent) => {
  //Rest of Function
}
```

Liste du contenu du répertoire avec readdir ou readdirSync

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

// Read the contents of the directory /usr/local/bin asynchronously.
// The callback will be invoked once the operation has either completed
// or failed.
fs.readdir('/usr/local/bin', (err, files) => {
  // On error, show it and return
  if(err) return console.error(err);

  // files is an array containing the names of all entries
  // in the directory, excluding '.' (the directory itself)
  // and '..' (the parent directory).

  // Display directory entries
  console.log(files.join(' '));
});
```

Une variante synchrone est disponible sous la forme `readdirSync` qui bloque le thread principal et empêche donc l'exécution du code asynchrone en même temps. La plupart des développeurs évitent les fonctions d'E / S synchrones pour améliorer les performances.

```
let files;

try {
  files = fs.readdirSync('/var/tmp');
} catch(err) {
  // An error occurred
  console.error(err);
}
```

En utilisant un générateur

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

// Iterate through all items obtained via
// 'yield' statements
// A callback is passed to the generator function because it is required by
// the 'readdir' method
function run(gen) {
  var iter = gen((err, data) => {
    if (err) { iter.throw(err); }

    return iter.next(data);
  });

  iter.next();
```

```
}

const dirPath = '/usr/local/bin';

// Execute the generator function
run(function* (resume) {
  // Emit the list of files in the directory from the generator
  var contents = yield fs.readdir(dirPath, resume);
  console.log(contents);
});
```

Lecture d'un fichier de manière synchrone

Pour toute opération de fichier, vous aurez besoin du module de système de fichiers:

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

Lire une chaîne

`fs.readFileSync` se comporte de la même manière que `fs.readFile` , mais ne prend pas de rappel car il se termine de manière synchrone et bloque donc le thread principal. La plupart des développeurs node.js préfèrent les variantes asynchrones qui ne causent pratiquement aucun retard dans l'exécution du programme.

Si une option de `encoding` est spécifiée, une chaîne sera renvoyée, sinon un `Buffer` sera renvoyé.

```
// Read a string from another file synchronously
let content;
try {
  content = fs.readFileSync('sync.txt', { encoding: 'utf8' });
} catch(err) {
  // An error occurred
  console.error(err);
}
```

Supprimer un fichier en utilisant unlink ou unlinkSync

Supprimer un fichier de manière asynchrone:

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

fs.unlink('/path/to/file.txt', function(err) {
  if (err) throw err;

  console.log('file deleted');
});
```

Vous pouvez également le supprimer de manière synchrone *:

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

```
fs.unlinkSync('/path/to/file.txt');
console.log('file deleted');
```

* éviter les méthodes synchrones car elles bloquent tout le processus jusqu'à la fin de l'exécution.

Lecture d'un fichier dans un tampon à l'aide de flux

Bien que la lecture du contenu d'un fichier soit déjà asynchrone à l'aide de la méthode `fs.readFile()` , nous souhaitons parfois obtenir les données dans un flux plutôt que dans un simple rappel. Cela nous permet de canaliser ces données vers d'autres emplacements ou de les traiter en même temps qu'à la fin.

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

// Store file data chunks in this array
let chunks = [];
// We can use this variable to store the final data
let fileBuffer;

// Read file into stream.Readable
let fileStream = fs.createReadStream('text.txt');

// An error occurred with the stream
fileStream.once('error', (err) => {
    // Be sure to handle this properly!
    console.error(err);
});

// File is done being read
fileStream.once('end', () => {
    // create the final data Buffer from data chunks;
    fileBuffer = Buffer.concat(chunks);

    // Of course, you can do anything else you need to here, like emit an event!
});

// Data is flushed from fileStream in chunks,
// this callback will be executed for each chunk
fileStream.on('data', (chunk) => {
    chunks.push(chunk); // push data chunk to array

    // We can perform actions on the partial data we have so far!
});
```

Vérifier les autorisations d'un fichier ou d'un répertoire

`fs.access()` détermine si un chemin existe et quelles autorisations un utilisateur a au fichier ou au répertoire sur ce chemin. `fs.access` ne `fs.access` pas un résultat, mais s'il ne renvoie pas d'erreur, le chemin existe et l'utilisateur dispose des autorisations souhaitées.

Les modes d'autorisation sont disponibles en tant que propriété sur l'objet `fs` , `fs.constants`

- `fs.constants.F_OK` - A des autorisations de lecture / écriture / exécution (si aucun mode n'est fourni, c'est le mode par défaut)

- `fs.constants.R_OK` - A des autorisations de lecture
- `fs.constants.W_OK` - A des autorisations d'écriture
- `fs.constants.X_OK` - Possède des autorisations d'exécution (Fonctionne comme `fs.constants.F_OK` sous Windows)

Asynchrone

```
var fs = require('fs');
var path = '/path/to/check';

// checks execute permission
fs.access(path, fs.constants.X_OK, (err) => {
    if (err) {
        console.log("%s doesn't exist", path);
    } else {
        console.log('can execute %s', path);
    }
});
// Check if we have read/write permissions
// When specifying multiple permission modes
// each mode is separated by a pipe : `|`
fs.access(path, fs.constants.R_OK | fs.constants.W_OK, (err) => {
    if (err) {
        console.log("%s doesn't exist", path);
    } else {
        console.log('can read/write %s', path);
    }
});
```

De manière synchrone

`fs.access` également une version synchrone `fs.accessSync` . Lorsque vous utilisez `fs.accessSync` vous devez le `fs.accessSync` dans un bloc try / catch.

```
// Check write permission
try {
    fs.accessSync(path, fs.constants.W_OK);
    console.log('can write %s', path);
}
catch (err) {
    console.log("%s doesn't exist", path);
}
```

Éviter les conditions de course lors de la création ou de l'utilisation d'un répertoire existant

En raison de la nature asynchrone de Node, créer ou utiliser un répertoire en premier:

1. vérifier son existence avec `fs.stat()` , puis
2. en créant ou en l'utilisant en fonction des résultats du contrôle d'existence,

peut conduire à une condition de concurrence si le dossier est créé entre l'heure de la vérification et l'heure de la création. La méthode ci-dessous `fs.mkdir()` et `fs.mkdirSync()` dans des wrappers de capture d'erreur qui laissent passer l'exception si son code est `EEXIST` (existe déjà). Si l'erreur est autre chose, comme `EPERM` (pemission denied), lancez ou passez une erreur comme le font les fonctions natives.

Version asynchrone avec `fs.mkdir()`

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

function mkdir (dirPath, callback) {
  fs.mkdir(dirPath, (err) => {
    callback(err && err.code !== 'EEXIST' ? err : null);
  });
}

mkdir('./existingDir', (err) => {

  if (err)
    return console.error(err.code);

  // Do something with `./existingDir` here

});
```

Version synchrone avec `fs.mkdirSync()`

```
function mkdirSync (dirPath) {
  try {
    fs.mkdirSync(dirPath);
  } catch(e) {
    if ( e.code !== 'EEXIST' ) throw e;
  }
}

mkdirSync('./existing-dir');
// Do something with `./existing-dir` now
```

Vérifier si un fichier ou un répertoire existe

Asynchrone

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

fs.stat('path/to/file', function(err) {
    if (!err) {
        console.log('file or directory exists');
    }
    else if (err.code === 'ENOENT') {
        console.log('file or directory does not exist');
    }
});
```

De manière synchrone

ici, nous devons envelopper l'appel de fonction dans un bloc `try/catch` pour gérer l'erreur.

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

try {
    fs.statSync('path/to/file');
    console.log('file or directory exists');
}
catch (err) {
  if (err.code === 'ENOENT') {
    console.log('file or directory does not exist');
  }
}
```

Cloner un fichier en utilisant des flux

Ce programme illustre comment on peut copier un fichier en utilisant des flux lisibles et inscriptibles en utilisant les fonctions `createReadStream()` et `createWriteStream()` fournies par le module de système de fichiers.

```
//Require the file System module
var fs = require('fs');

/*
  Create readable stream to file in current directory (__dirname) named 'node.txt'
  Use utf8 encoding
  Read the data in 16-kilobyte chunks
*/
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark:
16 * 1024 });

// create writable stream
var writable = fs.createWriteStream(__dirname + '/nodeCopy.txt');

// Write each chunk of data to the writable stream
readable.on('data', function(chunk) {
    writable.write(chunk);
});
```

Copie de fichiers par flux de tuyauterie

Ce programme copie un fichier en utilisant un flux lisible et un flux accessible en écriture avec la fonction `pipe()` fournie par la classe de flux

```
// require the file system module
var fs = require('fs');

/*
    Create readable stream to file in current directory named 'node.txt'
    Use utf8 encoding
    Read the data in 16-kilobyte chunks
*/
```

```
var readable = fs.createReadStream(__dirname + '/node.txt', { encoding: 'utf8', highWaterMark:
16 * 1024 });

// create writable stream
var writable = fs.createWriteStream(__dirname + '/nodePipe.txt');

// use pipe to copy readable to writable
readable.pipe(writable);
```

Changer le contenu d'un fichier texte

Exemple. Il remplacera le mot `email` par un `name` dans un fichier texte `index.txt` avec un simple `replace(/email/gim, 'name')` RegExp `replace(/email/gim, 'name')`

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

fs.readFile('index.txt', 'utf-8', function(err, data) {
    if (err) throw err;

    var newValue = data.replace(/email/gim, 'name');

    fs.writeFile('index.txt', newValue, 'utf-8', function(err, data) {
        if (err) throw err;
        console.log('Done!');
    })
})
```

Détermination du nombre de lignes d'un fichier texte

app.js

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

var file = 'path.to.file';
var linesCount = 0;
var rl = readline.createInterface({
    input: fs.createReadStream(file),
    output: process.stdout,
    terminal: false
});
rl.on('line', function (line) {
    linesCount++; // on each linebreak, add +1 to 'linesCount'
});
rl.on('close', function () {
    console.log(linesCount); // print the result when the 'close' event is called
});
```

Usage:

application de noeud

Lecture d'un fichier ligne par ligne

app.js

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

var file = 'path.to.file';
var rl = readline.createInterface({
    input: fs.createReadStream(file),
    output: process.stdout,
    terminal: false
});

rl.on('line', function (line) {
    console.log(line) // print the content of the line on each linebreak
});
```

Usage:

application de noeud

Lire Système de fichiers I / O en ligne: https://riptutorial.com/fr/node-js/topic/489/systeme-de-fichiers-i---o

Chapitre 107: Téléchargement de fichiers

Examples

Téléchargement de fichier unique avec multer

Se souvenir de

- créer un dossier pour le téléchargement (`uploads` dans l'exemple).
- installer multer `npm i -S multer`

`server.js` :

```
var express =   require("express");
var multer  =   require('multer');
var app         =   express();
var fs = require('fs');

app.get('/',function(req,res){
      res.sendFile(__dirname + "/index.html");
});

var storage =   multer.diskStorage({
  destination: function (req, file, callback) {
    fs.mkdir('./uploads', function(err) {
        if(err) {
            console.log(err.stack)
        } else {
            callback(null, './uploads');
        }
    })
  },
  filename: function (req, file, callback) {
    callback(null, file.fieldname + '-' + Date.now());
  }
});

app.post('/api/file',function(req,res){
    var upload = multer({ storage : storage}).single('userFile');
    upload(req,res,function(err) {
        if(err) {
            return res.end("Error uploading file.");
        }
        res.end("File is uploaded");
    });
});

app.listen(3000,function(){
    console.log("Working on port 3000");
});
```

`index.html` :

```
<form id        =  "uploadForm"
```

```
     enctype   =  "multipart/form-data"
     action    =  "/api/file"
     method    =  "post"
>
<input type="file" name="userFile" />
<input type="submit" value="Upload File" name="submit">
</form>
```

Remarque:

Pour télécharger un fichier avec une extension, vous pouvez utiliser la bibliothèque intégrée du chemin Node.js

Pour cela, il suffit de rechercher le `path` du fichier `server.js` :

```
var path = require('path');
```

et changer:

```
callback(null, file.fieldname + '-' + Date.now());
```

ajouter une extension de fichier de la manière suivante:

```
callback(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname));
```

Comment filtrer le téléchargement par extension:

Dans cet exemple, affichez comment télécharger des fichiers pour autoriser uniquement certaines extensions.

Par exemple, uniquement les extensions d'images. Ajoutez simplement à `var upload = multer({ storage : storage}).single('userFile');` condition du fichierFiltre

```
var upload = multer({
    storage: storage,
    fileFilter: function (req, file, callback) {
        var ext = path.extname(file.originalname);
        if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext !== '.jpeg') {
            return callback(new Error('Only images are allowed'))
        }
        callback(null, true)
    }
}).single('userFile');
```

Maintenant, vous ne pouvez télécharger que des fichiers image avec `jpeg` extensions `png` , `jpg` , `gif` ou `jpeg`

Utiliser un module formidable

Installer le module et lire les **documents**

```
npm i formidable@latest
```

Exemple de serveur sur le port 8080

```
var formidable = require('formidable'),
    http = require('http'),
    util = require('util');

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();

    form.parse(req, function(err, fields, files) {
      if (err)
          do-smth; // process error

      // Copy file from temporary place
      // var fs = require('fs');
      // fs.rename(file.path, <targetPath>, function (err) { ... });

      // Send result on client
      res.writeHead(200, {'content-type': 'text/plain'});
      res.write('received upload:\n\n');
      res.end(util.inspect({fields: fields, files: files}));
    });

    return;
  }

  // show a file upload form
  res.writeHead(200, {'content-type': 'text/html'});
  res.end(
    '<form action="/upload" enctype="multipart/form-data" method="post">'+
    '<input type="text" name="title"><br>'+
    '<input type="file" name="upload" multiple="multiple"><br>'+
    '<input type="submit" value="Upload">'+
    '</form>'
  );
}).listen(8080);
```

Lire Téléchargement de fichiers en ligne: https://riptutorial.com/fr/node-js/topic/4080/telechargement-de-fichiers

Chapitre 108: Utilisation d'IISNode pour héberger les applications Web Node.js dans IIS

Remarques

Répertoire virtuel / Application imbriquée avec des débordements de vues

Si vous envisagez d'utiliser Express pour afficher des vues à l'aide d'un moteur de visualisation, vous devez transmettre la valeur `virtualDirPath` à vos vues.

```
`res.render('index', { virtualDirPath: virtualDirPath });`
```

La raison en est que vos liens hypertexte vers d'autres vues sont hébergés par votre application et que les chemins de ressources statiques vous permettent de savoir où le site est hébergé sans devoir modifier toutes les vues après le déploiement. C'est l'un des pièges les plus ennuyeux et fastidieux de l'utilisation des répertoires virtuels avec IISNode.

Des versions

Tous les exemples ci-dessus fonctionnent avec

- Express v4.x
- IIS 7.x / 8.x
- Socket.io v1.3.x ou supérieur

Examples

Commencer

IISNode permet aux applications Web Node.js d'être hébergées sur IIS 7/8, comme le ferait une application .NET. Bien sûr, vous pouvez héberger vous-même votre processus `node.exe` sous Windows, mais pourquoi faire alors que vous pouvez simplement exécuter votre application dans IIS.

IISNode prendra en charge la mise à l'échelle sur plusieurs cœurs, la gestion des processus de `node.exe` et le recyclage automatique de votre application IIS à chaque mise à jour de votre application, pour ne citer que quelques-uns de ses avantages .

Exigences

IISNode a quelques exigences avant de pouvoir héberger votre application Node.js dans IIS.

1. Node.js doit être installé sur l'hôte IIS, 32 bits ou 64 bits, soit pris en charge.
2. IISNode installé x86 ou x64 , cela devrait correspondre au bitness de votre hôte IIS.
3. Le module Microsoft URL-Rewrite pour IIS installé sur votre hôte IIS.
    - C'est la clé, sinon les demandes à votre application Node.js ne fonctionneront pas comme prévu.
4. Un `Web.config` dans le dossier racine de votre application Node.js.
5. Configuration IISNode via un fichier `iisnode.yml` ou un élément `<iisnode>` dans votre fichier `Web.config` .

Exemple de base Hello World utilisant Express

Pour que cet exemple fonctionne, vous devez créer une application IIS 7/8 sur votre hôte IIS et ajouter le répertoire contenant l'application Web Node.js en tant que répertoire physique. Assurez-vous que votre identité de pool d'applications / applications peut accéder à l'installation Node.js. Cet exemple utilise l'installation Node.js 64 bits.

Projet Strucure

C'est la structure de projet de base d'une application Web IISNode / Node.js. Il semble presque identique à toute application Web non IISNode, à l'exception de l'ajout de `Web.config` .

```
- /app_root
  - package.json
  - server.js
  - Web.config
```

server.js - Application Express

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

// We need to get the port that IISNode passes into us
// using the PORT environment variable, if it isn't set use a default value
const port = process.env.PORT || 3000;

// Setup a route at the index of our app
server.get('/', (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

Configuration & Web.config

`Web.config` est comme tout autre `Web.config` IIS, à l'exception des deux éléments suivants: URL `<rewrite><rules>` et IISNode `<handler>` . Ces deux éléments sont des enfants de l'élément `<system.webServer>` .

Configuration

Vous pouvez configurer IISNode en utilisant un fichier `iisnode.yml` ou en ajoutant l'élément `<iisnode>` tant qu'enfant de `<system.webServer>` dans votre `Web.config` . Ces deux configurations peuvent être utilisées conjointement, mais dans ce cas, `Web.config` devra spécifier le fichier `iisnode.yml` **ET** les conflits de configuration seront pris à partir du fichier `iisnode.yml` . Ce remplacement de configuration ne peut pas se produire dans l'autre sens.

Gestionnaire IISNode

Pour qu'IIS sache que `server.js` contient notre application Web Node.js, nous devons le lui indiquer explicitement. Nous pouvons le faire en ajoutant le `<handler>` IISNode à l'élément `<handler>` handlers `<handlers>` .

```
<handlers>
  <add name="iisnode" path="server.js" verb="*" modules="iisnode"/>
</handlers>
```

Règles de réécriture d'URL

La dernière partie de la configuration consiste à s'assurer que le trafic destiné à notre application Node.js entrant dans notre instance IIS est dirigé vers IISNode. Sans règles de réécriture d'URL, nous aurions besoin de visiter notre application en accédant à `http://<host>/server.js` et, pire encore, en essayant de demander une ressource fournie par `server.js` vous obtiendrez un `404` . C'est pourquoi la réécriture d'URL est nécessaire pour les applications Web IISNode.

```
<rewrite>
    <rules>
        <!-- First we consider whether the incoming URL matches a physical file in the /public
folder -->
        <rule name="StaticContent" patternSyntax="Wildcard">
            <action type="Rewrite" url="public/{R:0}" logRewrittenUrl="true"/>
            <conditions>
                <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true"/>
            </conditions>
            <match url="*.*"/>
        </rule>

        <!-- All other URLs are mapped to the Node.js application entry point -->
        <rule name="DynamicContent">
            <conditions>
                <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="True"/>
```

```
            </conditions>
            <action type="Rewrite" url="server.js"/>
        </rule>
    </rules>
</rewrite>
```

Ceci est un fichier `Web.config` fonctionnel pour cet exemple , une configuration pour une installation Node.js 64 bits.

Ça y est, maintenant visitez votre site IIS et voyez votre application Node.js fonctionner.

Utiliser un répertoire virtuel IIS ou une application imbriquée via

L'utilisation d'un répertoire virtuel ou d'une application imbriquée dans IIS est un scénario courant et très probable que vous souhaiterez utiliser lors de l'utilisation d'IISNode.

IISNode ne fournit pas de prise en charge directe des répertoires virtuels ou des applications imbriquées via la configuration. Pour y parvenir, nous devons tirer parti d'une fonctionnalité d'IISNode qui ne fait pas partie de la configuration et qui est beaucoup moins connue. Tous les enfants de l'élément `<appSettings>` avec `Web.config` sont ajoutés à l'objet `process.env` tant que propriétés à l'aide de la clé appSetting.

Permet de créer un répertoire virtuel dans nos `<appSettings>`

```
<appSettings>
  <add key="virtualDirPath" value="/foo" />
</appSettings>
```

Dans notre application Node.js, nous pouvons accéder au paramètre `virtualDirPath`

```
console.log(process.env.virtualDirPath); // prints /foo
```

Maintenant que nous pouvons utiliser l'élément `<appSettings>` pour la configuration, profitons-en et utilisons-le dans notre code de serveur.

```
// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
  virtualDirPath = '/' + virtualDirPath;

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});
```

Nous pouvons également utiliser virtualDirPath avec nos ressources statiques

```
// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath,
'bower_components')));
```

Permet de mettre tout cela ensemble

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

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

// Access the virtualDirPath appSettings and give it a default value of '/'
// in the event that it doesn't exist or isn't set
var virtualDirPath = process.env.virtualDirPath || '/';

// We also want to make sure that our virtualDirPath
// always starts with a forward slash
if (!virtualDirPath.startsWith('/', 0))
  virtualDirPath = '/' + virtualDirPath;

// Public Directory
server.use(express.static(path.join(virtualDirPath, 'public')));
// Bower
server.use('/bower_components', express.static(path.join(virtualDirPath,
'bower_components')));

// Setup a route at the index of our app
server.get(virtualDirPath, (req, res) => {
    return res.status(200).send('Hello World');
});

server.listen(port, () => {
    console.log(`Listening on ${port}`);
});
```

Utiliser Socket.io avec IISNode

Pour que Socket.io fonctionne avec IISNode, les seules modifications nécessaires lorsque vous n'utilisez pas un répertoire virtuel / une application imbriquée se trouvent dans `Web.config` .

Étant donné que Socket.io envoie des requêtes commençant par `/socket.io` , IISNode doit communiquer à IIS pour que ces requêtes soient également gérées par IISNode et ne soient pas uniquement des requêtes de fichiers statiques ou un autre trafic. Cela nécessite un `<handler>` différent des applications IISNode standard.

```
<handlers>
    <add name="iisnode-socketio" path="server.js" verb="*" modules="iisnode" />
</handlers>
```

Outre les modifications apportées aux `<handlers>` nous devons également ajouter une règle de réécriture d'URL supplémentaire. La règle de réécriture envoie tout le trafic `/socket.io` à notre fichier serveur sur lequel le serveur Socket.io est exécuté.

```
<rule name="SocketIO" patternSyntax="ECMAScript">
    <match url="socket.io.+"/>
    <action type="Rewrite" url="server.js"/>
</rule>
```

Si vous utilisez IIS 8, vous devez désactiver le paramètre webSockets dans votre fichier `Web.config` en plus de l'ajout du gestionnaire ci-dessus et des règles de réécriture. Cela n'est pas nécessaire dans IIS 7 car il n'y a pas de prise en charge de webSocket.

```
<webSocket enabled="false" />
```

Lire Utilisation d'IISNode pour héberger les applications Web Node.js dans IIS en ligne: https://riptutorial.com/fr/node-js/topic/6003/utilisation-d-iisnode-pour-heberger-les-applications-web-node-js-dans-iis

Chapitre 109: Utiliser Browserfiy pour résoudre les erreurs "requises" avec les navigateurs

Examples

Exemple - fichier.js

Dans cet exemple, nous avons un fichier appelé **file.js.**

Supposons que vous devez analyser une URL à l'aide de JavaScript et du module de chaîne de requête NodeJS.

Pour ce faire, il vous suffit d'insérer la déclaration suivante dans votre fichier:

```
const querystring = require('querystring');
var ref = querystring.parse("foo=bar&abc=xyz&abc=123");
```

Que fait cet extrait?

Eh bien, tout d'abord, nous créons un module de chaîne de requête qui fournit des utilitaires pour analyser et formater les chaînes de requête URL. On peut y accéder en utilisant:

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

Ensuite, nous analysons une URL en utilisant la méthode .parse (). Il analyse une chaîne de requête URL (str) dans une collection de paires de clés et de valeurs.

Par exemple, la chaîne de requête `'foo=bar&abc=xyz&abc=123'` est analysée en:

```
{  foo: 'bar',  abc: ['xyz', '123']   }
```

Malheureusement, les navigateurs ne disposent pas de la méthode *require* , mais Node.js le fait.

Installer Browserfy

Avec Browserify vous pouvez écrire du code qui utilise *nécessitent* de la même manière que vous l' utiliser dans le nœud. Alors, comment résolvez-vous cela? C'est simple.

1. Premier noeud d’installation, livré avec npm. Alors fais:

   npm install -g **browserify**

2. Accédez au répertoire dans lequel se trouve votre fichier.js et installez notre module de *requête* avec npm:

   npm install **querystring**

Remarque: Si vous ne modifiez pas le répertoire spécifique, la commande échouera car elle ne trouve pas le fichier contenant le module.

3. Maintenant regroupez récursivement tous les modules requis en commençant par file.js dans un seul fichier appelé bundle.js (ou tout ce que vous voulez pour le nommer) avec la **commande browserify** :

   browserify file.js -o bundle.js

Browserify analyse l'arbre de syntaxe abstraite pour *que les* appels *require ()* parcourent l'ensemble du graphe de dépendance de votre

4. Enfin, déposez une seule balise dans votre HTML et vous avez terminé!

```
<script src="bundle.js"></script>
```

Ce qui se passe est que vous obtenez une combinaison de votre ancien fichier .js (**fichier.js** qui est) et de votre fichier **bundle.js** nouvellement créé. Ces deux fichiers sont fusionnés en un seul fichier.

Important

Veuillez garder à l'esprit que si vous souhaitez apporter des modifications à votre fichier.js et n'affectera pas le comportement de votre programme. **Vos modifications ne prendront effet que si vous éditez le bundle.js nouvellement créé**

Qu'est-ce que ça veut dire?

Cela signifie que si vous souhaitez modifier **file.js** pour quelque raison que ce soit, les modifications n'auront aucun effet. Il faut vraiment éditer **bundle.js** car c'est une fusion de **bundle.js** et de **file.js.**

Lire Utiliser Browserfiy pour résoudre les erreurs "requises" avec les navigateurs en ligne: https://riptutorial.com/fr/node-js/topic/7123/utiliser-browserfiy-pour-resoudre-les-erreurs--requises--avec-les-navigateurs

Chapitre 110: Utiliser des flux

Paramètres

Paramètre	Définition
Flux lisible	type de flux où les données peuvent être lues
Courant inscriptible	type de flux où les données peuvent être écrites
Flux duplex	type de flux à la fois lisible et inscriptible
Transformer le flux	type de flux duplex pouvant transformer les données en cours de lecture puis d'écriture

Examples

Lire des données depuis TextFile avec des flux

Les E / S dans le nœud sont asynchrones. Ainsi, interagir avec le disque et le réseau implique de transmettre des rappels aux fonctions. Vous pourriez être tenté d'écrire du code qui sert un fichier à partir du disque comme ceci:

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
    fs.readFile(__dirname + '/data.txt', function (err, data) {
        res.end(data);
    });
});
server.listen(8000);
```

Ce code fonctionne mais il est encombrant et stocke en mémoire tout le fichier data.txt pour chaque requête avant de renvoyer le résultat aux clients. Si le fichier data.txt est très volumineux, votre programme pourrait commencer à consommer beaucoup de mémoire car il est utilisé simultanément par de nombreux utilisateurs, en particulier pour les utilisateurs dont les connexions sont lentes.

L'expérience utilisateur est également médiocre car les utilisateurs devront attendre que tout le fichier soit mis en mémoire tampon sur votre serveur avant de pouvoir recevoir du contenu.

Heureusement, les deux arguments (req, res) sont des flux, ce qui signifie que nous pouvons écrire cela beaucoup mieux en utilisant fs.createReadStream () au lieu de fs.readFile ():

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
    var stream = fs.createReadStream(__dirname + '/data.txt');
    stream.pipe(res);
});
server.listen(8000);
```

Ici, .pipe () prend en charge l'écoute des événements 'data' et 'end' du fichier fs.createReadStream (). Ce code n'est pas seulement plus propre, mais maintenant, le fichier data.txt sera écrit sur les clients, un morceau à la fois, dès qu'ils sont reçus du disque.

Cours d'eau

Les flux lisibles peuvent être "canalisés" ou connectés à des flux inscriptibles. Cela rend le flux de données du flux source vers le flux de destination sans trop d'effort.

```
var fs = require('fs')

var readable = fs.createReadStream('file1.txt')
var writable = fs.createWriteStream('file2.txt')

readable.pipe(writable) // returns writable
```

Lorsque des flux inscriptibles sont également des flux lisibles, c'est-à-dire lorsqu'ils sont *des* flux *duplex* , vous pouvez continuer à les transmettre à d'autres flux inscriptibles.

```
var zlib = require('zlib')

fs.createReadStream('style.css')
  .pipe(zlib.createGzip()) // The returned object, zlib.Gzip, is a duplex stream.
  .pipe(fs.createWriteStream('style.css.gz')
```

Les flux lisibles peuvent également être acheminés vers plusieurs flux.

```
var readable = fs.createReadStream('source.css')
readable.pipe(zlib.createGzip()).pipe(fs.createWriteStream('output.css.gz'))
readable.pipe(fs.createWriteStream('output.css')
```

Notez que vous devez accéder aux flux de sortie de manière synchrone (en même temps) avant tout flux de données. Sinon, des données incomplètes pourraient être transmises.

Notez également que les objets de flux peuvent émettre `error` événements d' `error` ; Assurez-vous de gérer ces événements de manière responsable sur *chaque* flux, selon vos besoins:

```
var readable = fs.createReadStream('file3.txt')
var writable = fs.createWriteStream('file4.txt')
readable.pipe(writable)
readable.on('error', console.error)
writable.on('error', console.error)
```

Création de votre propre flux lisible / inscriptible

Nous verrons les objets de flux renvoyés par des modules tels que fs, etc., mais si nous voulons créer notre propre objet pouvant être diffusé.

Pour créer un objet Stream, nous devons utiliser le module de flux fourni par NodeJs

```
    var fs = require("fs");
    var stream = require("stream").Writable;

    /*
     *  Implementing the write function in writable stream class.
     *  This is the function which will be used when other stream is piped into this
     *  writable stream.
     */
    stream.prototype._write = function(chunk, data){
        console.log(data);
    }

    var customStream = new stream();

    fs.createReadStream("am1.js").pipe(customStream);
```

Cela nous donnera notre propre flux accessible en écriture. nous pouvons implémenter n'importe quoi dans la fonction *_write* . La méthode ci-dessus fonctionne dans la version NodeJs 4.xx mais dans NodeJs 6.x **ES6** introduit des classes, donc la syntaxe a changé. Voici le code pour la version 6.x de NodeJs

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

    class MyWritable extends Writable {
      constructor(options) {
        super(options);
      }

      _write(chunk, encoding, callback) {
        console.log(chunk);
      }
    }
```

Pourquoi Streams?

Examinons les deux exemples suivants pour lire le contenu d'un fichier:

Le premier, qui utilise une méthode asynchrone pour lire un fichier et fournit une fonction de rappel qui est appelée une fois que le fichier est entièrement lu dans la mémoire:

```
fs.readFile(`${__dirname}/utils.js`, (err, data) => {
  if (err) {
    handleError(err);
  } else {
    console.log(data.toString());
  }
})
```

Et le second, qui utilise des `streams` pour lire le contenu du fichier, pièce par pièce:

```
var fileStream = fs.createReadStream(`${__dirname}/file`);
var fileContent = '';
fileStream.on('data', data => {
  fileContent += data.toString();
})

fileStream.on('end', () => {
  console.log(fileContent);
})

fileStream.on('error', err => {
  handleError(err)
})
```

Il convient de mentionner que les deux exemples font **exactement la même chose** . Quelle est la différence alors?

- Le premier est plus court et semble plus élégant
- La seconde vous permet de faire un traitement sur le fichier **en** cours de lecture (!)

Lorsque les fichiers que vous traitez sont petits, il n'y a pas d'effet réel lors de l'utilisation de `streams` , mais que se passe-t-il lorsque le fichier est volumineux? (si grand qu'il faut 10 secondes pour le lire en mémoire)

Sans `streams` vous attendez, ne faites absolument rien (à moins que votre processus ne fasse autre chose), jusqu'à ce que les 10 secondes passent et que le fichier soit **entièrement lu** , et alors seulement vous pouvez commencer à traiter le fichier.

Avec les `streams` , vous obtenez le contenu du fichier pièce par pièce, **exactement quand ils sont disponibles** - et cela vous permet de traiter le fichier **pendant** sa lecture.

L'exemple ci-dessus n'illustre pas la manière dont les `streams` peuvent être utilisés pour des tâches qui ne peuvent pas être effectuées lors du rappel, alors examinons un autre exemple:

Je voudrais télécharger un fichier `gzip` , le décompresser et enregistrer son contenu sur le disque. Étant donné l' `url` du fichier, c'est ce qui doit être fait:

- Télécharger le fichier
- Décompressez le fichier
- Enregistrez-le sur le disque

Voici un [petit fichier] [1], stocké dans mon stockage `S3` . Le code suivant effectue les opérations ci-dessus dans le mode de rappel.

```
var startTime = Date.now()
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}, (err, data) => {
  // here, the whole file was downloaded

  zlib.gunzip(data.Body, (err, data) => {
    // here, the whole file was unzipped
```

```
    fs.writeFile(`${__dirname}/tweets.json`, data, err => {
      if (err) console.error(err)

      // here, the whole file was written to disk
      var endTime = Date.now()
      console.log(`${endTime - startTime} milliseconds`) // 1339 milliseconds
    })
  })
})

// 1339 milliseconds
```

Voici à quoi cela ressemble en utilisant des `streams` :

```
s3.getObject({Bucket: 'some-bucket', Key: 'tweets.gz'}).createReadStream()
  .pipe(zlib.createGunzip())
  .pipe(fs.createWriteStream(`${__dirname}/tweets.json`));

// 1204 milliseconds
```

Oui, ce n'est pas plus rapide lorsque vous traitez de petits fichiers - le fichier testé pèse `80KB` . Tester cela sur un fichier plus gros, de `71MB` gzippé (`71MB` Mo `382MB`), montre que la version des `streams` est beaucoup plus rapide

- Il a fallu 20925 millisecondes pour télécharger `71MB` , décompressez-le, puis écrivez `382MB` sur le disque - en **utilisant le mode de rappel** .
- En comparaison, il a fallu 13434 millisecondes pour faire la même chose en utilisant la version de `streams` (35% plus rapide, pour un fichier pas si gros).

Lire Utiliser des flux en ligne: https://riptutorial.com/fr/node-js/topic/2974/utiliser-des-flux

Chapitre 111: Utiliser WebSocket avec Node.JS

Examples

Installation de WebSocket

Il existe plusieurs moyens d'installer WebSocket dans votre projet. Voici quelques exemples:

```
npm install --save ws
```

ou dans votre package.json en utilisant:

```
"dependencies": {
  "ws": "*"
},
```

Ajouter WebSocket à votre fichier

Pour ajouter des ws à votre fichier, utilisez simplement:

```
var ws = require('ws');
```

Utilisation de WebSocket et de WebSocket Server

Pour ouvrir un nouveau WebSocket, ajoutez simplement quelque chose comme:

```
var WebSocket = require("ws");
var ws = new WebSocket("ws://host:8080/OptionalPathName);
// Continue on with your code...
```

Ou pour ouvrir un serveur, utilisez:

```
var WebSocketServer = require("ws").Server;
var ws = new WebSocketServer({port: 8080, path: "OptionalPathName"});
```

Un exemple simple de serveur WebSocket

```
var WebSocketServer = require('ws').Server
, wss = new WebSocketServer({ port: 8080 }); // If you want to add a path as well, use path:
"PathName"

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(message) {
    console.log('received: %s', message);
  });
```

```
  ws.send('something');
});
```

Lire Utiliser WebSocket avec Node.JS en ligne: https://riptutorial.com/fr/node-js/topic/6106/utiliser-websocket-avec-node-js

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Node.js	4444, Abdelaziz Mokhnache, Abhishek Jain, Adam, Aeolingamenfel, Alessandro Trinca Tornidor, Aljoscha Meyer, Amila Sampath, Ankit Gomkale, Ankur Anand, arcs, Aule, B Thuy, baranskistad, Bundit J., Chandra Sekhar, Chezzwizz, Christopher Ronning, Community, Craig Ayre, David Gatti, Djizeus, Florian Hämmerle, Franck Dernoncourt, ganesshkumar, George Aidonidis, Harangue, hexacyanide, Iain Reid, Inanc Gumus, Jason, Jasper, Jeremy Banks, John Slegers, JohnnyCoder, Joshua Kleveter, KolesnichenkoDS, krishgopinath, Léo Martin, Majid, Marek Skiba, Matt Bush, Meinkraft, Michael Irigoyen, Mikhail, Milan Laslop, ndugger, Nick, olegzhermal, Peter Mortensen, RamenChef, Reborn, Rishikesh Chandra, Shabin Hashim, Shiven, Sibeesh Venu, sigfried, SteveLacy, Susanne Oberhauser, thefourtheye, theunexpected1, Tomás Cañibano, user2314737, Volodymyr Sichka, xam, zurfyx
2	Analyse des arguments de ligne de commande	yrtimiD
3	analyseur csv dans le noeud js	aisflat439
4	API CRUD simple basée sur REST	Iceman
5	Applications Web avec Express	Aikon Mogwai, Alex Logan, alexi2, Andres C. Viesca, Aph, Asaf Manassen, Batsu, bekce, brianmearns, Community, Craig Ayre, Daniel Verem, devnull69, Everettss, Florian Hämmerle, H. Pauwelyn, Inanc Gumus, jemiloii, Kid Binary, kunerd, Marek Skiba, Mikhail, Mohit Gangrade, Mukesh Sharma, Naeem Shaikh, Niklas, Nivesh, noob, Ojen, Pasha Rumkin, Paul, Rafal Wiliński, Shabin Hashim, SteveLacy, tandrewnichols, Taylor Ackley, themole, tverdohleb, Vsevolod Goloviznin, xims, Yerko Palma
6	Arrêt gracieux	RamenChef, Sathish
7	Async / En attente	Cami Rodriguez, Cody G., cyanbeam, Dave, David Xu, Dom Vinyard, m_callens, Manuel, nomanbinhussein, Toni Villena

8	async.js	David Knipe, devnull69, DrakaSAN, F. Kauder, jerry, lsampaio, Shriganesh Kolhe, Sky, walid
9	Authentification Windows sous node.js	CJ Harries
10	Base de données (MongoDB avec Mongoose)	zurfyx
11	Bibliothèque Mongoose	Alex Logan, manuerumx, Mikhail, Naeem Shaikh, Qiong Wu, Simplans, Will
12	Bluebird Promises	David Xu
13	Bon style de codage	Ajitej Kaushik, RamenChef
14	Cadres de modèles	Aikon Mogwai
15	Cas d'utilisation de Node.js	vintproykt
16	Chargement automatique des modifications	ch4nd4n, Dean Rather, Jonas S, Joshua Kleveter, Nivesh, Sanketh Katta, zurfyx
17	CLI	Ze Rubeus
18	Comment les modules sont chargés	RamenChef, umesh
19	Communication Arduino avec nodeJs	sBanda
20	Communication client-serveur	Zoltán Schmidt
21	Conception d'API reposante: meilleures pratiques	fresh5447, nilakantha singh deo
22	Connectez-vous à Mongodb	FabianCook, Nainesh Raval, Shriganesh Kolhe
23	Création d'API avec Node.js	Mukesh Sharma
24	Création d'une	Dave

	bibliothèque Node.js prenant en charge les promesses et les rappels d'erreur en premier	
25	Débogage à distance dans Node.JS	Rick, VooVoo
26	Débogage de l'application Node.js	4444, Alister Norris, Ankur Anand, H. Pauwelyn, Matthew Shanley
27	Défis de performance	Antenka, SteveLacy
28	Déploiement d'applications Node.js en production	Apidcloud, Brett Jackson, Community, Cristian Boariu, duncanhall, Florian Hämmerle, guleria, haykam, KlwntSingh, Mad Scientist, MatthieuLemoine, Mukesh Sharma, raghu, sjmarshy, tverdohleb, tyehia
29	Déploiement de l'application Node.js sans temps d'arrêt.	gentlejo
30	Désinstallation de Node.js	John Vincent Jardin, RamenChef, snuggles08, Trevor Clarke
31	ECMAScript 2015 (ES6) avec Node.js	David Xu, Florian Hämmerle, Osama Bari
32	Émetteurs d'événements	DrakaSAN, Duly Kinsky, Florian Hämmerle, jamescostian, MindlessRanger, Mothman
33	Environnement	Chris, Freddie Coleman, KlwntSingh, Louis Barranqueiro, Mikhail, sBanda
34	Envoi d'un flux de fichiers au client	Beshoy Hanna
35	Envoyer une notification Web	Houssem Yahiaoui
36	Eventloop	Kelum Senanayake
37	Évitez le rappel de l'enfer	tyehia
38	Exécuter des fichiers ou des commandes	guleria, hexacyanide, iSkore

	avec des processus enfants	
39	Exécution de node.js en tant que service	Buzut
40	Exiger()	Philip Cornelius Glover
41	Exportation et consommation de modules	Aminadav, Craig Ayre, cyanbeam, devnull69, DrakaSAN, Fenton, Florian Hämmerle, hexacyanide, Jason, jdrydn, Loufylouf, Louis Barranqueiro, m02ph3u5, Marek Skiba, MrWhiteNerdy, MSB, Pedro Otero, Shabin Hashim, tkone, uzaif
42	Exportation et importation d'un module dans node.js	AndrewLeonardi, Bharat, commonSenseCode, James Billingham, Oliver, sharif.io, Shog9
43	forgeron	RamenChef, vsjn3290ckjnaoij2jikndckjb
44	Frameworks de tests unitaires	David Xu, Florian Hämmerle, skiilaa
45	Frameworks NodeJS	dthree
46	Garder une application de noeud en cours d'exécution	Alex Logan, Bearington, cyanbeam, Himani Agrawal, Mikhail, mscdex, optimus, pietrovismara, RamenChef, Sameer Srivastava, somebody, Taylor Swanson
47	Gestion de la requête POST dans Node.js	Manas Jayanth
48	Gestion des exceptions	KlwntSingh, Nivesh, riyadhalnur, sBanda, sjmarshy, topheman
49	Gestionnaire de paquets de fils	Andrew Brooke, skiilaa
50	grognement	Naeem Shaikh, Waterscroll
51	Guide du débutant NodeJS	Niroshan Ranapathi
52	Histoire de Nodejs	Kelum Senanayake
53	http	Ahmed Metwally
54	Injection de dépendance	Niroshan Ranapathi

55	Installer Node.js	Alister Norris, Aminadav, Anh Cao, asherbar, Batsu, Buzut, Chance Snow, Chezzwizz, Dmitriy Borisov, Florian Hämmerle, GilZ, guleria, hexacyanide, HungryCoder, Inanc Gumus, Jacek Labuda, John Vincent Jardin, Josh, KahWee Teng, Maciej Rostański, mmhyamin, Naing Lin Aung, NuSkooler, Shabin Hashim, Siddharth Srivastva, Sveratum, tandrewnichols, user2314737, user6939352, V1P3R, victorkohl
56	Intégration de Cassandra	Vsevolod Goloviznin
57	Intégration des passeports	Ankit Rana, Community, Léo Martin, M. A. Cordeiro, Rupali Pemare, shikhar bansal
58	Intégration Mongodb	cyanbeam, FabianCook, midnightsyntax
59	Intégration MongoDB pour Node.js / Express.js	William Carron
60	Intégration MSSQL	damitj07
61	Intégration MySQL	Aminadav, Andrés Encarnación, Florian Hämmerle, Ivan Schwarz, jdrydn, JohnnyCoder, Kapil Vats, KlwntSingh, Marek Skiba, Rafael Gadotti Bachovas, RamenChef, Simplans, Sorangwala Abbasali, surjikal
62	Intégration PostgreSQL	Niroshan Ranapathi
63	Interagir avec la console	ScientiaEtVeritas
64	Koa Framework v2	David Xu
65	Livrer du code HTML ou tout autre type de fichier	Himani Agrawal, RamenChef, user2314737
66	Localisation du noeud JS	Osama Bari
67	Lodash	M1kstur
68	Loopback - Connecteur basé sur REST	Roopesh
69	Module de cluster	Benjamin, Florian Hämmerle, Kid Binary, MayorMonty, Mukesh Sharma, riyadhalnur, Vsevolod Goloviznin

70	Multithreading	arcs
71	N-API	Parham Alvani
72	Node.js (express.js) avec angular.js Exemple de code	sigfried
73	Node.js Architecture & Inner Workings	Ivan Hristov
74	Node.js avec CORS	Buzut
75	Node.JS avec ES6	Inanc Gumus, xam, ymz, zurfyx
76	Node.js avec Oracle	oliolioli
77	Node.js code pour STDIN et STDOUT sans utiliser de bibliothèque	Syam Pradeep
78	Node.js Conception fondamentale	Ankur Anand, pietrovismara
79	Node.JS et MongoDB.	midnightsyntax, RamenChef, Satyam S
80	Node.js Gestion des erreurs	Karlen
81	Node.js v6 Nouvelles fonctionnalités et améliorations	creyD, DominicValenciana, KlwntSingh
82	NodeJS avec Redis	evalsocket
83	NodeJs Routing	parlad neupane
84	Nœud serveur sans framework	Hasan A Yousef, Taylor Ackley
85	Notifications push	Mario Rozic
86	npm	Abhishek Jain, AJS, Amreesh Tyagi, Ankur Anand, Asaf Manassen, Ates Goral, ccnokes, CD.., Cristian Cavalli, David G., DrakaSAN, Eric Fortin, Everettss, Explosion Pills, Florian Hämmerle, George Bailey, hexacyanide, HungryCoder, Ionică Bizău, James Taylor, João Andrade, John Slegers, Jojodmo, Josh, Kid Binary, Loufylouf, m02ph3u5, Matt, Matthew Harwood

		, Mehdi El Fadil, Mikhail, Mindsers, Nick, notgiorgi, num8er, oscarm, Pete TNT, Philipp Flenker, Pieter Herroelen, Pyloid, QoP, Quill, Rafal Wiliński, RamenChef, Ratan Kumar, RationalDev, rdegges, refaelos, Rizowski, Shiven, Skanda, Sorangwala Abbasali, still_learning, subbu, the12, tlo, Un3qual, uzaif, VladNeacsu, Vsevolod Goloviznin, Wasabi Fan, Yerko Palma
87	nvm - Node Version Manager	cyanbeam, guleria, John Vincent Jardin, Luis González, pranspach, Shog9, Tushar Gupta
88	OAuth 2.0	tyehia
89	package.json	Ankur Anand, Asaf Manassen, Chance Snow, efeder, Eric Smekens, Florian Hämmerle, Jaylem Chaudhari, Kornel, lauriys , mezzode, OzW, RamenChef, Robbie, Shabin Hashim, Simplans, SteveLacy, Sven 31415, Tomás Cañibano, user6939352, V1P3R, victorkohl
90	passport.js	Red
91	Performances Node.js	Florian Hämmerle, Inanc Gumus
92	Pirater	signal
93	Pool de connexions Mysql	KlwntSingh
94	Prise en main du profilage de nœuds	damitj07
95	Programmation asynchrone	Ala Eddine JEBALI, cyanbeam, Florian Hämmerle, H. Pauwelyn, John, Marek Skiba, Native Coder, omgimanerd, slowdeath007
96	Programmation synchrone vs asynchrone dans nodejs	Craig Ayre, Veger
97	Rappel à la promesse	Clement JACOB, Michael Buen, Sanketh Katta
98	Readline	4444, Craig Ayre, Florian Hämmerle, peteb
99	Routage des requêtes ajax avec Express.JS	RamenChef, SynapseTech

100	Sécurisation des applications Node.js	akinjide, devnull69, Florian Hämmerle, John Slegers, Mukesh Sharma, Pauly Garcia, Peter G, pranspach, RamenChef, Simplans
101	Sequelize.js	Fikra, Niroshan Ranapathi, xam
102	Socket.io communication	Forivin, N.J.Dawson
103	Sockets TCP	B Thuy
104	Structure du projet	damitj07
105	Structure Route-Controller-Service pour ExpressJS	nomanbinhussein
106	Système de fichiers I / O	4444, Accepted Answer, Aeolingamenfel, Christophe Marois, Craig Ayre, DrakaSAN, Duly Kinsky, Florian Hämmerle, gnerkus, Harshal Bhamare, hexacyanide, jakerella, Julien CROUZET, Louis Barranqueiro, midnightsyntax, Mikhail, peteb, Shiven, still_learning, Tim Jones, Tropic, Vsevolod Goloviznin, Zanon
107	Téléchargement de fichiers	Aikon Mogwai, Iceman, Mikhail, walid
108	Utilisation d'IISNode pour héberger les applications Web Node.js dans IIS	peteb
109	Utiliser Browserfiy pour résoudre les erreurs "requises" avec les navigateurs	Big Dude
110	Utiliser des flux	cyanbeam, Duly Kinsky, efeder, johni, KlwntSingh, Max, Ze Rubeus
111	Utiliser WebSocket avec Node.JS	Rowan Harley