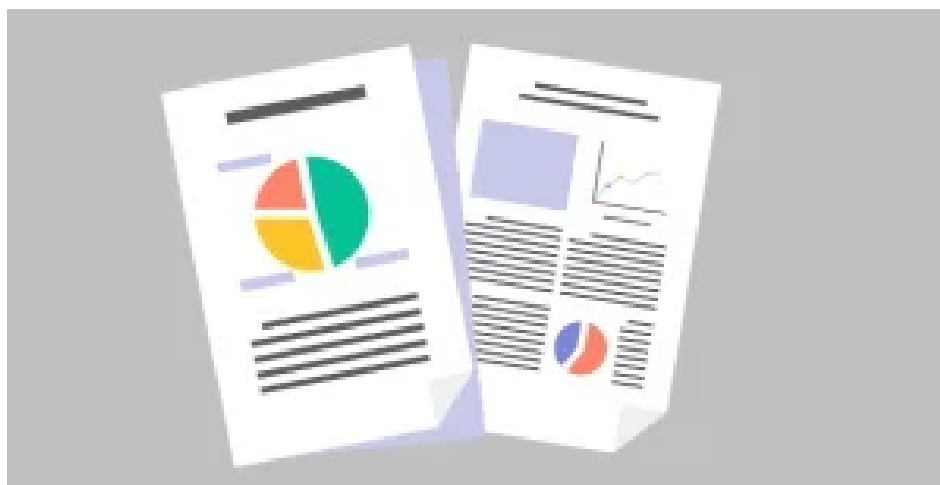


Support de formation complet avec exemples pour apprendre node js



3 étoiles sur 5 a partir de 24 votes.  

Votez ce document:



Chapitre 1: Démarrer avec

Remarques

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 sont écrites en JavaScript pur et peuvent être exécutées dans l'environnement sous Windows, Linux, etc.

Versions

Version	Date de sortie
	2017-07-20
	2017-07-19
	2017-07-11
	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

Articles similaires

[Comment s'améliorer en informatique avec plaisir ?](#)

[9 façons amusantes et créatives d'apprendre l'anglais](#)


[Tutoriel Excel : la fonction CHOISIR avec des exemples](#)

[Quelle langue choisir pour booster ma carrière : espagnol, chinois, une autre ?](#)


[Comment faire une convocation pour une formation ?](#)

[L'importance d'apprendre l'anglais pro pour les futurs ingénieurs](#)


Documents similaires

 [Support de cours complet avec exemples pour démarrer avec Angular JS](#)


Support de cours complet avec exemples pour démarrer avec...

 [Manuel pour apprendre à travailler avec MS Project](#)


Manuel pour apprendre à travailler avec MS Project

 [Tutoriel pour apprendre à Modifier une image avec GIMP](#)

Tutoriel pour apprendre à Modifier une image avec GIMP

 [Cours complet en maintenance PC](#)

Cours complet en maintenance PC

 [Formation pour apprendre à créer des applications graphiques en Python avec PyQt](#)

Formation pour apprendre à créer des applications graphiqu...

 [Cours CSS3 : Les effets de texte](#)

Cours CSS3 : Les effets de texte

v6.9.4 2017-01-05

[Contactez-nous](#)

v6.9.3 2017-01-05

[A propos de nous](#)

v6.9.2 2016-12-06

[On recrute](#)

v6.9.1 2016-10-19

[Rechercher dans le site](#)

v6.9.0 2016-10-18

[Politique de confidentialité](#)

v6.8.0 2016-10-12

[Droit d'auteur/Copyright](#)

[Plan du site](#)

v6.8.0 2016-10-12

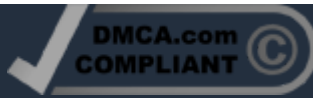
v6.7.0 2016-09-27

v6.6.0 2016-09-14

v6.5.0 2016-08-26

v6.4.0 2016-08-12

... ..



Examples

Bonjour HTTP server HTTP

Tout d'abord, installez 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 (un module inclus dans la source de , 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 ();
}).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' nous pouvons exécuter l'application en allant dans le répertoire où se trouve le fichier et en utilisant la commande suivante:

node

Le serveur créé est alors accessible avec l'URL `http://localhost:1337` ou 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

peut également être utilisé pour créer des utilitaires de ligne de commande. L'exemple cidessous 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 ``` array, which has the following structure:

[0] The path of the executable that started the process

[1] The path to this application

[2-n] the command line arguments

Example: ['/bin/node', '/path/to/yourscript', 'arg1', 'arg2',] src: #process_process_argv

```
*/
```

```
// Store the first argument as username.
```

```
var username = [2];
```

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

```
    // Extract the filename
```

```
    var appName = [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:
```

```
    // && ls -> won't execute ls
```

```
    // David && ls -> will execute ls (1);
```

```
}
```

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

Installation et exécution de

Pour commencer, installez 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 commande.

Exécution d'un programme de noeud

Pour exécuter un programme , exécutez simplement `node` ou `nodejs` , où 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), cet environnement propose généralement une variable d'environnement `PORT` que vous pouvez utiliser pour exécuter votre serveur. Changer le numéro de port en vous permet d'accéder à l'application.

Par exemple,

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

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( || 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
```

Le dépôt Github peut être trouvé ici:

Déboguer en mode natif

Vous pouvez également déboguer nativement en le démarrant comme suit:

```
node debug
```

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

```
debugger;
```

Pour plus d'informations, voir ici .

Dans 2, utilisez la commande suivante:

Dans 8, utilisez la commande suivante.

```
node --inspect-brk
```

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 (le par exemple). Ensuite, installez le module `express` en utilisant `npm install --save express` depuis la ligne de commande. *Reportezvous à 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
('/', function(request, response) { ('Hello, World!');

});

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

});
```

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

```
node
```

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 (=== '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); ('Hello, World!');

}

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

```

    response.writeHead(404); (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); ('Hello, World!');
},
  '/foo': function foo (request, response) { response.writeHead(200); ('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 ( in routes) { return routes[](request, response);
}
  response.writeHead(404); (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 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 -out`

Maintenant que nous avons rempli nos documents, il est temps de prétendre que nous sommes une autorité de signature géniale. `openssl x509 -req -in -signkey -out`

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(""), cert: fs.readFileSync("")
};
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 ('data', function(data) {
    ('Received: %s [it is %d bytes long]', data.toString().replace(/(\n)/gm,""), data.length);
  });
  // Let us know when the transmission is over ('end', function() {
    ('EOT (End Of Transmission)');
  });
});
// Start listening on a specific port and address server.listen(PORT, HOST, function() {
  ("I'm listening at %s, on port %s", HOST, PORT);
});
// When an error occurs, show it. ('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(""), cert: fs.readFileSync(""), rejectUnauthorized: false
};
var client = tls.connect(PORT, HOST, options, function() {
  // Check if the authorization worked if (client.authorized) { ("Connection authorized by a Certificate
  Authority.");
  } else {
    ("Connection not authorized: " + client.authorizationError)
  }
  // Send a friendly message
  client.write("I am the client sending you a message.");
});
("data", function(data) {
  ('Received: %s [it is %d bytes long]', data.toString().replace(/(\n)/gm,""), data.length);
  // Close the connection after receiving the message ();
});
('close', function() {
  ("Connection closed");
});
// When an error occurs, show it. ('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, 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

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 (fichier 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 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 Uint8Array de manière plus optimisée et adaptée aux cas d'utilisation de .

c / c ++ _ addons

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

child_process

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

Une seule instance de s'exécute dans un seul thread. Pour tirer parti des systèmes multicore, l'utilisateur voudra parfois lancer un cluster de processus 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

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.

Une grande partie de l'API de base de 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.

Les interfaces HTTP dans sont conçues pour prendre en charge de nombreuses fonctionnalités du protocole qui étaient traditionnellement difficiles à utiliser. En particulier, des messages volumineux,

eventuellement 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 , ceci est implémenté en tant que module séparé.

module

a un système de chargement de module simple. Dans , 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 est obsolète .

chaîne de requête

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

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.

Un flux est une interface abstraite pour travailler avec des données en continu dans . 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 . 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 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 . **tls_ (ssl)**

Le module tls fournit une implémentation des protocoles TLS (Transport Layer Security) et 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 .

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 . 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 . *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 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. Aller dans ce répertoire:

2. allez dans ce repertoire:

```
cd conf
```

3. récupérer ce fichier à utiliser comme raccourci de configuration: wget

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

```
openssl req -new -x509 -days 9999 -config -keyout -out
```

5. Maintenant que nous avons notre autorité de certification dans et , générons une clé privée pour le serveur: openssl genrsa -out 4096

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

```
wget
```

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

```
openssl req -new -config -key -out
```

8. signer la demande:

```
openssl x509 -req -extfile -days 999 -passin "pass:password" -in -CA -CAkey -CAcreateserial -out
```

Étape 2: installez votre certificat en tant que certificat racine

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

```
sudo cp
```

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 contenant votre code de serveur actuel.

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

```
var https = require('https'); var fs = require('fs');  
var httpsOptions = { key: fs.readFileSync(""), cert: fs.readFileSync("")  
};  
var app = function (req, res) { res.writeHead(200); ("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(""), cert: fs.readFileSync("")  
};  
var app = function (req, res) { res.writeHead(200); ("hello world\n");  
}
```



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

1. allez dans le répertoire où se trouve votre :

```
cd /path/to
```

2. lancez :

```
node
```

Lire Démarrer avec en ligne:

Chapitre 2: Analyse des arguments de ligne de commande

Exemples

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 ", "Input file")
  .option("-o, --out-file ", "Output file")
  .action(doConvert);
options.parse();
if (!length) ();
function doConvert(options){
  //do something with options.inFile and options.outFile
};
```

Interrupteurs booléens

```
const options = require("commander");
options
  .option("-v, --verbose")
  .parse();
if (options.verbose){ ("Let's make some noise!");
}
```

Lire Analyse des arguments de ligne de commande en ligne:

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.

Exemples

Utiliser FS pour lire dans un fichier CSV

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

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

```
'use strict'
```

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

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

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

Lire analyseur csv dans le noeud js en ligne:

Chapitre 4: API CRUD simple basée sur REST

Exemples

API REST pour CRUD dans Express 3+

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

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

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

```
//GET all items
```

```
('/item', function (req, res) { (itemStore);
```

```
});
```

```
//GET the item with specified id
```

```
('/item/:id', function (req, res) { (itemStore[]);
```

```
});
```

```
//POST new item
```

```
('/item', function (req, res) { (); ();
```

```
});
```

```
//PUT edited item in-place of item with specified id ('/item/:id', function (req, res) { itemStore[] = ();
```

```
});
```

```
//DELETE item with specified id
```

```
server.delete('/item/:id', function (req, res) { itemStore.splice(, 1) ();
```

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

Lire API CRUD simple basée sur REST en ligne:

Chapitre 5: Applications Web avec Express

Introduction

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

Le site officiel d'Express est <https://expressjs.com/>. La source peut être trouvée [ici](https://github.com/expressjs/express).

Syntaxe

- (chemin [, middleware], callback [, callback])
- (chemin [, middleware], callback [, callback])
- (chemin [, middleware], callback [, callback])
- app ['delete'] (chemin [, middleware], callback [, callback])
- (chemin [, middleware], callback [, callback])
- (rappel)

Paramètres

Paramètre	Détails
path	Spécifie la portion de chemin ou l'URL que le rappel donné va gérer. Une ou plusieurs fonctions qui seront appelées avant le rappel.
middleware	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 <i>rappel</i>	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.

ExemplesCommencer

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 et ajoutez le code suivant qui crée un nouveau serveur Express et lui ajoute un point de terminaison (/ping) avec la méthode :

```
const express = require('express');
const app = express();
('/ping', (request, response) => { ('pong');
});
app.listen(8080, 'localhost');
```

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

> node

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 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:

```
('/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
```

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

```
// POST
```

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

```
// PUT
```

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

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

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

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

ou

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

ou

```
('*', 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 ('/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:

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

Et puis dans le fichier contenant vos itinéraires:

```
const other = require(""); ('/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

```
('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');
```

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

Maintenant, supposons une requête comme celle-ci

maintenant, supposons une requête comme celle-ci

```
PUT /settings/32135
```

```
{  
  "name": "Peter"  
}
```

Vous pouvez accéder au nom affiché comme ceci

```
// "Peter"
```

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

Application express modulaire

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

Module:

```
//  
const express = require('express');  
module.exports = function(options = {}) { // Router factory  const router = express.Router();  
  ('/greet', (req, res, next) => { (options.greeting);  
  });  
  return router; };
```

Application:

```
//  
const express = require('express'); const greetMiddleware = require("");  
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://: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:

```
//  
const express = require('express');  
module.exports = function(options = {}) { // Router factory  const router = express.Router();  
  // Get controller  const {service} = options;  
  ('/greet', (req, res, next) => { ( service.createGreeting( || 'Stranger')  
  );
```

```

    ..
  });

  return router; };

Application:

//

const express = require('express'); const greetMiddleware = require("");

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://:8080/api/v1/service1/greet?name=World> le résultat sera Hello, World et vous accéderez à <http://: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

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

//Creates a Root Route ('/',function(req, res){ res.render('index'); //renders the 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) { ('Server is running at port', PORT);

} else {

  (.JSON.stringify(err));

```

```

    res.render('index', {
    });
});

```

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

<%= title %>

```

<% for(var i=0; i<supplies.length; i++) { %>

```

```

    <%= supplies[i] %>

```

```

<% } %>

```

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 devons 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:

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).

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(); (cors()); // for all routes

```

```

var port = || 8080;

```

```

('/', function(req, res) { var info = {
  'string value': 'StackOverflow'

```



```

    string_value: 'StackOverflow',
    'number_value': 8476
  }
  (info);
  // or
  /* (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() { (' 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 et 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 |—
              |—
              |— public
              |—
              |—

```

Voici comment configurer Express pour servir des fichiers statiques:

```

const express = require('express'); const app = express();
(express.static('public'));

```

Remarque: une fois le dossier configuré, , 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:

```

('/static', express.static('public'));

```

rendra les ressources disponibles sous le préfixe /static/ .

Plusieurs dossiers

Il est possible de définir plusieurs dossiers en même temps:

```
(express.static('public')); (express.static('images')); (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 `expressreverse` :

```
npm install express-reverse
```

Branchez-le dans votre projet:

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

Ensuite, utilisez-le comme:

```
('test', '/hello', function(req, res) { ('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) => { ('test', '/hello', function(req, res) { ('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 /`

```
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:

```
// catch 404 and forward to error handler (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 (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.

```
('/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());
});
});
('/api/foo', function(req, res, next) {
  // Only /api/foo will run this  doSomethingWithMember(req.member);
});
('/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, `('/foo', function(err, req, res, next))` mais généralement, un seul gestionnaire d'erreur qui affiche une page d'erreur suffit.

```
('/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. (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.

```
('/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. = data; next();
```

```
});
```

```
}, function(req, res, next) {
```

```
    req.member.parsedData = parseMemberData(); next();
```

```
});
```

```
('/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

```
('/path/:id(\\d+)', function (req, res, next) { // please note: "next" is passed if ( == 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("");
```

```
    } 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.status(200).end('OK');
});
// Be sure: order of have matter
// Error handler
(function(err, req, res, next) { if (smth-check, e.g. != 'POST') return next(err); // go-to Error handler 2.
    (, err.message);
    if () // if req via ajax then send json else render error-page (err); else res.render("", {error:
err.message});
});
// Error handler 2
(function(err, req, res, next) {
    // do smth here e.g. check that error is MyError if (err instanceof MyError) { (err.message, err.arg1,
err.arg2); }

    ()
});
```

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. (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';
```

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

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

```
(function (req, res, next) { function afterResponse() { res.removeListener('finish', afterResponse);
res.removeListener('close', afterResponse);
    // options after response
```

```
// actions after response
}

('finish', afterResponse); ('close', afterResponse);

// action before request

// eventually calling `next()` next(); });

(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

Gestion des requêtes POST

Tout comme vous gérez les demandes d'obtention dans Express avec la méthode `req.get()`, vous pouvez utiliser la méthode `req.body` 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. ({});
(bodyParser.urlencoded({ extended: true }));

('/post-data-here', function(req, res, next){
  // 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(); (cookieParser());

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

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

```
    return ('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.

```
(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 (function(req, res, next){ = 'testuser'; next(); // it will pass the
control to next matching route
});

('/', function(req, res){ var user = ; (user); // testuser return (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
('/:names/:name', function(req, res, next){ if ( == 'john'){ return ('Valid Name');
} else{
  next(new Error('Not valid name')); //pass to error handler
}
});

//error handler
(function(err, req, res, next){ (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:

```
(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 = || 3000;
```

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

```
('/',(req,res)=>('HelloWorld!')); ('/wiki',(req,res)=>('This is wiki page.')); ((req,res)=>('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:

Chapitre 6: Arrêt gracieux

Exemples

Arrêt gracieux - SIGTERM

En utilisant **server.close ()** et **()** , 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'}); ('Hello World\n');  
}, 4000);
```

```
}).listen(9090, function (err) {
```

```
('listening http://localhost:9090/'); ('pid is ' + );
```

```
});
```

```
('SIGTERM', function () { server.close(function () { (0);
```

```
});
```

```
});
```


Lire Arrêt gracieux en ligne:

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 8 ou 7 en utilisant le drapeau `--harmonyasync-await` .

Exemples

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:

```
('/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, ); if (flags.length === 0) return res.status(404).send({
        message: 'flag not found' });
      return ({ 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) => { (" (res) => { callback(.temperature)
  })
}

const getAirPollution = (callback) => { (" (res) => { callback(.pollution)
  });
}

getTemperature(function(temp) { getAirPollution(function(pollution) { (`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) => { (" (res) => {
```

```

        resolve(.temperature)
    })
})
}

const getAirPollution = () => { return new Promise((resolve, reject) => {    (" (res) => {
resolve(.pollution)

    })
    })
}

getTemperature()
.then(temp => (`the temp is ${temp}`))
.then(() => getAirPollution())
.then(pollution => (`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:

Chapitre 8:

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);`

Exemples

Parallèle: multi-tâches

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);
}
(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);
}
(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', 'resultOfMediumTime2');
}, 500);
}
},
function(err, results) { if (err) { return console.error(err);
}
(results);
});

```

Résultat :

```

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

```

Série: mono-tâche indépendante

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);
    }
    (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);
    }
    (results);
});
```

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

Cascade: **mono-tâche dépendante**

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 ( === "Aamu"){ friendsResult = [{ name : 'Alice'
  }, {
  name: 'Bob'
  }];
  }
  callback(null, friendsResult);
  }, 500);
}

async.waterfall([ getUserRequest, getUserFriendsRequest
],
function(err, results) { if (err) { return console.error(err);
}
(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 , 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(pour gérer efficacement le tableau de données)** Lorsque nous voulons gérer un tableau de données, il est préférable d'utiliser . 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

toutes les données et que nous voulions 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']; (arrayOfData, function(eachUserName, callback) {
    // Perform operation on each user.
    ('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.
    ('unable to create user');
    } else {
    ('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) { ('First Execute ..'); callback (null,
'userPersonalData');}, function (callback) { ('Second Execute ..'); callback (null, 'userDependentData');}],
fonction (err, result) { (result);});
```

//Sortie:

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

Lire en ligne:

Chapitre 9: Authentification Windows sous

Remarques

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

Exemples

Utiliser activedirectory

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

Installation

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

Usage

```
// Initialize
```

```
var ActiveDirectory = require('activedirectory'); var config = { url: " ", baseDN: 'dc=domain,dc=com'
};
```

```
var ad = new ActiveDirectory(config); var username = " "; var password = 'password';
```

```
// Authenticate
```

```
ad.authenticate(username, password, function(err, auth) { if (err) { ('ERROR: '+JSON.stringify(err));
return;
```

```
  } if (auth) { ('Authenticated!');
```

```
  } else { ('Authentication failed!');
```

```
  }
```

```
});
```

Lire Authentification Windows sous en ligne:

Chapitre 10: Base de données (MongoDB avec Mongoose)

Exemples

Connexion Mongoose

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

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

(ECMA 6)

```
import mongoose from 'mongoose';
```

```
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');      const      db      =
mongoose.connection;
```

```
('error', (console, 'DB connection error!'));
```

(ECMA 5.1)

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

```
mongoose.connect('mongodb://localhost:27017/stackoverflow-example');      var      db      =
```

```
mongoose.connection; ('error', (console, 'DB connection error!'));
```

Modèle

Définissez votre (vos) modèle (s): app / models / (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 / (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',
});

((err) => { if (err) throw err;
('User saved!');
});
```

ECMA5.1:

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

(function (err) { if (err) throw err;
('User saved!');
});
```

Lire des données

ECMA6:

```
User.findOne({ name: 'stack' }, (err, user) => { if (err) throw err;
if (!user) { ('No user was found');
} else {
('User was found');
}
});
```

ECMA5.1:

```
User.findOne({ name: 'stack'
}, function (err, user) { if (err) throw err;
  if (!user) { ('No user was found');
  } else {
    ('User was found');
  }
});
```

Lire Base de données (MongoDB avec Mongoose) en ligne:

Chapitre 11: Bibliothèque Mongoose

Exemples

Connectez-vous à MongoDB en utilisant Mongoose

Tout d'abord, installez Mongoose avec:

```
npm install mongoose
```

Ensuite, ajoutez-le à 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 utilisant node

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; ('error', (console, 'connection error:')); ('open', function() {
  // we're connected!
});
```

Enregistrer les données sur MongoDB en utilisant les routes Mongoose et

Installer

D'abord, installez les paquets nécessaires avec:

npm install express cors mongoose

Code

Ensuite, ajoutez des dépendances à votre fichier , créez le schéma de base de données et le nom de la collection, créez un serveur 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 = || 8080; app.listen(port, function() { (' listening on port ' + port);
```

```
});
```

Ajoutez maintenant les routes que nous utiliserons pour écrire les données:

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

```
  var savedata = new Model({
```

```
    'request': query,
```

```
    'time': Math.floor() / 1000) // Time of save the data in unix timestamp format
```

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

```
  if(result) {
```

```
    (result)
```

```
  }
```

```
  })
```

```
})
```

Ici, la variable de query sera le paramètre 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) {
```

```
    // ...
    (result)
  }
}) //
```

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

Usage

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

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

Où 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

Installer

D'abord, installez les paquets nécessaires avec:

```
npm install express cors mongoose
```

Code

Ensuite, ajoutez des dépendances à , créez le schéma de base de données et le nom de la collection, créez un serveur 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');
```

```
mongoose.connect('mongodb://localhost:27017/dbName',
var port = || 8080; app.listen(port, function() { (' listening on port ' + port);
});
```

Ajoutez maintenant les routes que nous utiliserons pour interroger les données:

```
( '/find/:query', cors(), function(req, res) { var query = req.params.query;
({
  'request': query
}, function(err, result) { if (err) throw err; if (result) { (result)
} else {
(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 avec node :

Usage

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

http://localhost:8080/find/

Où 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,

Routes et \$ text Operator

Installer

D'abord, installez les paquets nécessaires avec:

```
npm install express cors mongoose
```

Code

Ensuite, ajoutez des dépendances à , créez le schéma de base de données et le nom de la collection, créez un serveur 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 = || 8080; app.listen(port, function() { (' listening on port ' + port);
});

```

Ajoutez maintenant les routes que nous utiliserons pour interroger les données:

```

('/find/:query', cors(), function(req, res) { var query = req.params.query;
({
  'request': query
}, function(err, result) { if (err) throw err; if (result) { (result)
} else {
(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 :

```

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

```



```
const find = async ({ request, text }) => {
```

Et remplacer:

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

Avec:

```
{  
  $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/>

Où 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:
}
});
```

```
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 `ensureIndex` 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`.

```
('autoIndex', false);
```

Fonctions utiles de Mongoose

Mongoose contient des fonctions intégrées basées sur `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 , créez le schéma de base de données et le nom de la collection, créez un serveur 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 = || 8080; app.listen(port, function() { (' listening on port ' + port);
});
(function(err, req, res, next) { console.error(err.stack); res.status(500).send('Something broke!');
});
(function(req, res, next) { res.status(404).send('Sorry cant find that!');
});

```

Ajoutez maintenant les routes que nous utiliserons pour interroger les données:

```

('/find/:query', cors(), function(req, res, next) { var query = req.params.query;
({
  'request': query
})
.exec() //remember to add exec, queries have a .then attribute but aren't promises
.then(function(result) { if (result) { (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 avec node :

Usage

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

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

Où 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:

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("").then(contents => { (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()
```

Exemple de réduire:

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

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("").then()
```

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()
```

Like Bluebird Promises on [line](#):

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.

Exemples

Programme de base pour l'inscription

Dans cet exemple, il sera expliqué de diviser le code 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. : - 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
(config.API_PATH, userRoutes());

//Start the server app.listen(); ('Server started at - '+ + ":" );

: - 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 : || 8080,
DB : {
//MongoDB configuration
HOST : 'localhost',
```

```

    PORT : '27017',
    DATABASE : 'db'
  },
  /*
  * Get DB Connection String for connecting to MongoDB database
  */
  getDBString : function(){ return 'mongodb://' + '+' : '+' + '/' + .DATABASE;
  },
  /*
  * Get the http URL
  */
  getHTTPEndpoint : function(){ return 'http://' + '+' : '+' ;
  }
  module.exports = config;
  : - 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()
  User = mongoose.model('User'); else
  User = mongoose.model('User', UserSchema);
  //Export the User Model module exports = User;

```



```
//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 = .repassword; var password = .password; var userEmail  
= .email;
```

```
  //Check if the email address already exists
```

```
  ({'email': userEmail}, function(err, usr){ if(usr.length > 0){ //Email Exists
```

```
  ('Email already exists'); return;
```

```
  } else
```

```
  {
```

```
    //New Email
```

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

```
  }
```

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

```
    //Create User var user = new User(); = ; user.email = .email; user.password = passwordHash; =  
Date.parse() || ""; user.gender = .gender;
```

```
    //Validate the User user.validate(function(err){ if(err){ (err); return;
```

```
    }else{
```

```
    //Finally save the User (function(err){ if(err)
```

```
    {
```

```
    (err); return;
```

```
    }
```

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

```
  });
```

```
  }
```

```
  });
```

```
  }
```

```
  });
```

```
  }
```

```
}
```

```
module.exports = UserController;
```

: - 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 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:

Chapitre 14: Cadres de modèles

Exemples

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 - ;

Installer - npm i nunjucks

Utilisation de base avec ci-dessous.

```
var express = require('express'); var nunjucks = require('nunjucks');
var app = express(); (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) { ('myFilter', obj, arg1, arg2);
  // Do smth with obj return obj;
});
env.addGlobal('myFunc', function(obj, arg1) { ('myFunc', obj, arg1);
  // Do smth with obj return obj;
});
('/', function(req, res){ res.render("", {title: 'Main page'});
});
('/foo' function(req, res){ res.locals smthVar = 'This is Sparta!'; res.render(" {title: 'Foo page'});
```

```

    (/foo, function(req, res){ res.render(smthVar | This is sparta. , res.render( , {title: 'foo page'}),
  });

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

```

Nunjucks example

```

{% block content %}

{{title}}

{% endblock %}

{% extends "" %}

{# This is comment #}

{% block content %}

```

{{title}}

```

{# apply custom function and next build-in and custom filters #}

{{ myFunc(smthVar) | lower | myFilter(5, 'abc') }}

{% endblock %}

```

Lire Cadres de modèles en ligne:

Chapitre 15: Cas d'utilisation de

Exemples

Serveur HTTP

```

const http = require('http');

('Starting server '); var config = { port: 80, contentType: 'application/json; charset=utf-8'
};

// JSON-API server on port 80

var server = http.createServer(); server.listen(); ('error', (err) => { if ( == 'EADDRINUSE')
console.error('Port '+ + ' is already in use'); else console.error(err.message);
});

('request', (request, res) => { var remoteAddress = request.headers['x-forwarded-for'] ||
request.connection.remoteAddress; // Client address (remoteAddress + ' ' + request.method + ' ');

var out = {};

// Here you can change output according to `` = ; res.writeHead(200, {
'Content-Type': config.contentType
});

(JSON.stringify(out));

\\

```

```

    }},
    ('listening', () => {
        c.info('Server is available: http://localhost:' + );
    });

```

Console avec invite de commande

```

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

rl.pause(); ('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 = (' '); try { (eval(arg)); } catch
(e) { (e); }
    },
    exit: function(arg) { ()
    }
};

('line', (str) => {  rl.pause();  var arg = ().match(/([^\"]+)|("[^"\\]|\\.)+"/g); // Applying regular
expression for removing all spaces except for what between double quotes:
; 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 ('Command "' + commandName + '" doesn\'t exist');
    }

    rl.prompt();
});

/*
    END OF
    Commands recognition
*/

```

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

Lire Cas d'utilisation de en ligne:

Chapitre 16: Chargement automatique des modifications

Exemples

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 (ou nodemon entry)

Cela remplace l'utilisation habituelle de node (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 :

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

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

Browsersync

Vue d'ensemble

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

Installation

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

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

```
$ npm install browsersync --save-dev
```

```
$ 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 "" --files "**/*.js"
```

Remplacez 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 et .

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

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

Ensuite, vous allez ajouter cette ligne à votre :

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

Browsersync fonctionne comme un module CommonJS , donc pas besoin d'un plugin . 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:

Lire Chargement automatique des modifications en ligne:

Chapitre 17: CLI

Syntaxe

```
• noeud [options] [options v8] [ -e "script" ] [arguments]
```

noeud [options] [script v0] [fichier 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`

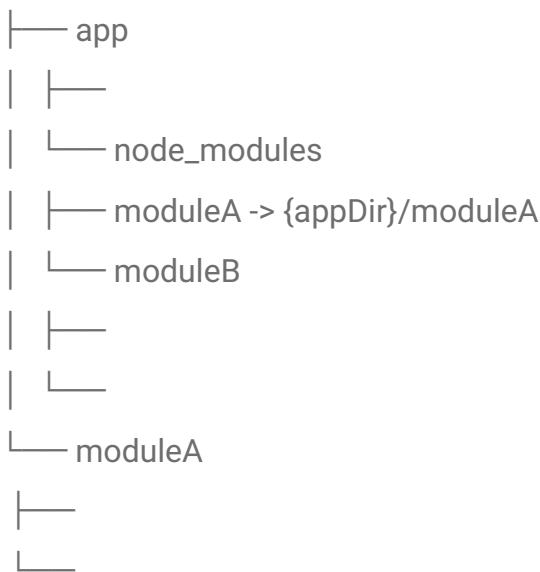
Ajouté à: v6.0.0 Remplit automatiquement toutes les nouvelles instances de Buffer et SlowBuffer.

--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 charge un module à partir d'un chemin d'accès symboliquement lié à un autre emplacement sur disque, 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}



L'indicateur de ligne de commande --preserve-symlinks indique à 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 (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 pour être construit avec un support cryptographique. (Par défaut))

`--enable-fips`

Ajouté dans: v6.0.0 Activer la crypto compatible FIPS au démarrage. (Nécessite 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:

Chapitre 18: Comment les modules sont chargés

Exemples

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 travaillez en ligne de commande, NPM recherche téléchargement

dans `/usr/local/lib/node_modules` . Si vous tapez ce qui suit dans le shell, NPM recherchera, téléchargera et installera 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 noeud 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 :

```
var myModule = require("");
```

Si Node ne parvient pas à trouver le fichier, il examinera le dossier parent nommé

. 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 noeud 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 et Node recherchera, dans ce cas, un fichier sous le chemin d'accès `package.json` .

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:

Chapitre 19: Communication Arduino avec nodeJs

Introduction

Façon de montrer comment peut communiquer avec Arduino Uno.

Exemples

Node.js communication avec Arduino via serialport

Node js code

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

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

Exemple d':

```
const express = require('express'); const app = express(); var SerialPort = require("serialport");
var port = 3000;
var arduinoCOMPort = "COM3";
var arduinoSerialPort = new SerialPort(arduinoCOMPort, { baudrate: 9600
});
('open',function() { ('Serial Port ' + arduinoCOMPort + ' is opened.')}
});
('/', function (req, res) {
  return ('Working');
})
('/:action', function (req, res) {
  var action = req.params.action || req.param('action');
  if(action == 'led'){ arduinoSerialPort.write("w"); return ('Led light is on!');
  }
  if(action == 'off') { arduinoSerialPort.write("t");
  return ("Led light is off!");
  }
  return ('Action: ' + action);
});
app.listen(port, function () { ('Example app listening on port http://0.0.0.0:' + port + '!');
});
```

Exemple de serveur express:

node

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); // Begin listening on port 9600 for serial
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);
```

```

    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) ();
        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:

Chapitre 20: Communication client-serveur

Exemples

/ w Express, jQuery et 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=")

```

```

//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/domainetest',

//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( !=
predeclared){ document.getElementById("modify").innerHTML = "JSON changed!\n" + jsonstr;
};
}
});
});
});
});
//"

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/domainetest' ('/', function(req, res, next) {
});

//same for POST requests - notice, how the AJAX request above was defined as 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 " var sent_data = ; = "ttony33";
(sent_data);
});
module.exports = router;
// basé sur un élément personnel utilisé:
Lire Communication client-serveur en ligne:

```

Chapitre 21: Conception d'API reposante: meilleures pratiques

Exemples

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) => {
    ((err, r) => { if(err){
      (err)
    } else { (r)
    }
  })
  .post((req, res) => { const request = new Request({ type: , info:
  });
    = ._id; ("ABOUT TO SAVE REQUEST", request); ((err, r) => { if (err) { ({ message: 'there was an error
saving your r' });
    } else { (r);
    }
  });
});

```

Meilleure façon:

```

Router.route('/')
  .get((req, res) => {
    ((err, r) => { if(err){ (err)
    } else {

```

```

    return next(err)
  }
})
})
.post((req, res) => { const request = new Request({ type: , info:
});
= ._id; ("ABOUT TO SAVE REQUEST", request); ((err, r) => { if (err) {
  return next(err)
} else { (r);
}
});
});

```

Lire Conception d'API reposante: meilleures pratiques en ligne:

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

Syntaxe

- MongoClient.connect ('mongodb: //127.0.0.1: 27017 / crud', fonction (err, db) { // faites quelque chose ici});

Exemples

Exemple simple pour connecter mongoDB à partir de

```

MongoClient.connect('mongodb://localhost:27017/myNewDB',function (err,db) { if(err) ("Unable to
connect DB. Error: " + err) else ('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 de base

```

var MongoClient = require('mongodb').MongoClient;
//connection with mongoDB
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {

```

```
//check the connection if(err){ ("connection failed.");  
}else{  
  ("successfully connected to mongoDB.");  
  };
```

Lire Connectez-vous à Mongoddb en ligne:

Chapitre 23: Création d'API avec

Exemples

GET api en utilisant Express

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: ""  
  }];  
// GET /api/users  
( '/api/users', function(req, res){ return (users); //return response as JSON  
});  
app.listen('3000', function(){ ('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 .

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: ""  
  }];  
(bodyParser.urlencoded({ extended: false }));  
  
(());  
// GET /api/users  
( '/api/users', function(req, res){ return (users);  
});  
/* POST /api/users
```



```

{
  "user": {
    "id": 3,
    "name": "Test User",
    "age" : 20,
    "email": ""
  }
}
*/

('/api/users', function (req, res) { var user = ; (user);
  return ('User has been added successfully');
});

app.listen('3000', function(){ ('Server listening on port 3000');
});

```

Lire Création d'API avec en ligne:

Chapitre 24: Création d'une bibliothèque 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`.

Exemples

Exemple de module et programme correspondant utilisant Bluebird

```

'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

```

```

// 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);
},
};

'use strict';
const math = require('./math');

// classic callbacks
math.callbackSum(1, 3, function(err, result) { if (err) ('Test 1: ' + err); else ('Test 1: the answer is ' +
result);
});

math.callbackSum(1, 'd', function(err, result) { if (err) ('Test 2: ' + err); else ('Test 2: the answer is ' +
result);
});

// promises
math.promiseSum(2, 5).then(function(result) {
('Test 3: the answer is ' + result);
})
.catch(function(err) { ('Test 3: ' + err);
});

math.promiseSum(1)
.then(function(result) {
('Test 4: the answer is ' + result);
})
.catch(function(err) { ('Test 4: ' + err);
});

// promise/callback method used like a promise
(8, 2)
.then(function(result) {
('Test 5: the answer is ' + result);
});

```

```

    ('Test 5: the answer is ' + result),
  ))
).catch(function(err) {
  ('Test 5: ' + err);
});
// promise/callback method used with callbacks
(7, 11, function(err, result) { if (err) ('Test 6: ' + err); else ('Test 6: the answer is ' + result);
});
// promise/callback method used like a promise with async/await syntax
(async () => {
  try { let x = await (6, 3); ('Test 7a: ' + x);
    let y = await (4, 's'); ('Test 7b: ' + y);
  } catch(err) {
    (err.message);
  }
})();

```

Lire Création d'une bibliothèque prenant en charge les promesses et les rappels d'erreur en premier en ligne:

Chapitre 25: Débogage à distance dans

Exemples

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=` .

Lorsque votre processus nœud démarre, vous devriez voir le message

Debugger listening on 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

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

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 en ligne:

Chapitre 26: Débogage de l'application

Exemples

Core debugger et inspecteur de noeud

Utiliser le débogueur de base

fournit un utilitaire de débogage non graphique. Pour démarrer la génération dans le débogueur, démarrez l'application avec cette commande:

```
node debug
```

Considérez l'application simple suivante contenue dans le

```
'use strict';
```

```
function addTwoNumber(a, b){
```

```
// function returns the sum of the two numbers debugger return a + b;
```

```
}
```

```
var result = addTwoNumber(5, 9); (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
```

Une fois que les commandes ci-dessus s'exécutent, vous verrez la sortie suivante. Pour quitter l'interface du débogueur, tapez ()

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 eval pour exécuter le code de manière interactive. Le mode eval a le même contexte que la ligne

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 nœud intégré

v6.3.0

Vous pouvez exécuter l'inspecteur v8 intégré au nœud! Le plug-in d' n'est plus nécessaire. Passez simplement le drapeau de l'inspecteur et vous recevrez une URL pour l'inspecteur

```
node --inspect
```

Utilisation de l'inspecteur de nœud

Installez l'inspecteur de nœud:

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

Exécutez votre application avec la commande node-debug:

```
node-debug
```

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 nœud 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 en ligne:

Chapitre 27: Défis de performance

Exemples

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 nœud 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

3. Le client envoie continuellement des checkes à un serveur pour les mises à jour de statut de l'ID de tâche donné.

Structure du projet:

```
project
|
|
|
|
|——js
|
|
|
|
|——srv
|
|——models
|
|——tasks
```

:

```
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');
(bodyParser.urlencoded({ extended: true })); ();
(express.static(__dirname + '/../'));
('/', function(request, response){ response.render("");
});
//route for the request itself
('/: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 = ();
    //process the messages coming from the task processor ('message', function(msg){ task.status
    = msg.status; (); }.bind(this));
    //remove previously opened node instance when we finished ('close', function(msg){ ();
  });
  //send some params to our separate task var params = { message: 'Hello from main thread'
};
```

```

    (params); response.status(200).json(task);
  });
});
//route to check is the request is finished the calculations ('/is-ready', function(request, response){
  Task
  .findById() .exec(function(err, task){
    response.status(200).json(task);
  });
});
mongoose.connect('mongodb://localhost/test'); http.listen('1234');
:
var mongoose = require('mongoose');
var taskSchema = mongoose.Schema({ status: { type: String
  }
});
mongoose.model('Task', taskSchema);
module.exports = mongoose.model('Task');
:
('message', function(msg){ init = function(){ processData(msg.message);
  }.bind(this());
  function processData(message){
    //send status update to the main app
    ({ status: 'We have started processing your data.' });
    //long calculations .. setTimeout(function(){ ({ status: 'Done!' });
    //notify node, that we are done with this task process.disconnect();
  }, 5000);
  }
});
('uncaughtException',function(err){ ("Error happened: " + err.message + "\n" + err.stack + ".\n");
("Gracefully finish the routine.");
});
:

```

Example of processing long-running node requests.

Run

Log:

:

```
$(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);
});
});
```

:

```
{
  "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:

Chapitre 28: Déploiement d'applications en production

Exemples

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(.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 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
```

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`

méthode 4: Utiliser `cross-env` package `cross-env`

Ce 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 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 -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`

Déploiement à l'aide de `PM2`

`PM2` est un gestionnaire de processus de production pour les applications, 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 utilisant `PM2`.

```
pm2 start --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.

Forever

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 surveille forever votre processus et le redémarre s'il se bloque.

Installer forever globalement.

\$ npm install -g forever

Exécuter l'application:

\$ forever start

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:

Utiliser différentes propriétés / configurations pour différents environnements tels que dev, qa, staging etc

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.

•

```
{
  "PORT": 3000,
  "DB": {
    "host": "localhost",
    "user": "bob",
    "password": "12345"
  }
}
```

•

```
{
  "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.

```
.forEach(function (val) { var arg = val.split("="); if (arg.length > 0) { if (arg[0] === 'env') { var env =
require('./' + arg[1] + '.json'); = env;
}
}
});
```

Nous donnons des arguments à l'application comme suit

node env=dev

si nous utilisons gestionnaire de processus comme *pour toujours* aussi simple que

forever start env=dev

Profiter des clusters

... .. -

Une seule instance de s'exécute dans un seul thread. Pour tirer parti des systèmes multicore, l'utilisateur voudra parfois lancer un cluster de processus 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.
    ('your server is working on ' + numCPUs + ' cores');
    for (var i = 0; i < numCPUs; i++) { ()
    }
    ('disconnect', function(worker) { console.error('disconnect!');
    //clearTimeout(timeout); ()
    });
    } else {
    require("");
    }
}
```

Lire Déploiement d'applications en production en ligne:

Chapitre 29: Déploiement de l'application sans temps d'arrêt.

Examples

Déploiement à l'aide de PM2 sans temps d'arrêt.

```
{
  "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 ('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 SIGKILL final.

```
const http = require('http'); const express = require('express');
const app = express(); const server = http.Server(app); const port = 80;
server.listen(port, function() { ('ready');
});
('SIGINT', function() { server.close(function() { (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 ('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 sans temps d'arrêt. en ligne:

Chapitre 30: Désinstallation de

Exemples

Désinstallez complètement sur Mac OSX

Dans Terminal sur votre système d'exploitation Mac, entrez les 2 commandes suivantes:

```
lsbom -f -l -s -pf | 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 sous Windows

Pour désinstaller sous Windows, utilisez Ajout / Suppression de programmes comme celui-ci:

1. Ouvrez Add or Remove Programs dans le menu Démarrer.
2. Recherchez **Windows 10**:
3. Cliquez sur .
4. Cliquez sur Désinstaller.
5. Cliquez sur le nouveau bouton Désinstaller.

Windows 7-8.1:

3. Cliquez sur le bouton Désinstaller sous .

Lire Désinstallation de en ligne:

Chapitre 31: ECMAScript 2015 (ES6) avec

Exemples

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  
}
```

(x) // 1 (y) // ReferenceError: y is not defined (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 = (function (n) { return n * n  
})(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 = ((n) => { return n * n  
})(squares)
```

Exécuter dans RunKit

Cependant, cela peut être écrit encore plus concis. Si le corps de la fonction se compose d'une seule

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 = (n => n * n) (squares)
```

Exécuter dans RunKit **déstructurer**

```
let [x,y, nums] = [0, 1, 2, 3, 4, 5, 6]; (x, y, nums);  
let {a, b, props} = {a:1, b:2, c:3, d:{e:4}} (a, b, props);  
let dog = {name: 'fido', age: 3}; let {name:n, age} = dog; (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(" (err, res, payload)=>{ payload = JSON.parse(payload); let {LastPrice} = payload;  
(LastPrice);  
});
```

Classe ES6

```
class Mammel { constructor(legs){ = legs;  
  } eat(){ ('eating ');  
  }  
  static count(){ ('static count ');  
  }  
}  
class Dog extends Mammel{ constructor(name, legs){ super(legs); = name;  
  } sleep(){ (); ('sleeping');  
  }  
}  
let d = new Dog('fido', 4);  
d.sleep();  
d.eat(); ('d', d);  
.... ..
```