

# Node.js : le livre du débutant



Par Manuel Kiessling (auteur)  - Didier Mouronval (traducteur) 

Date de publication : 8 mars 2012

Ce livre est le guide du débutant de Node.js, il vous apprendra comment réaliser une première application Web fonctionnelle à l'aide de cet environnement JavaScript côté serveur.

L'auteur présente son travail comme étant le guide qu'il aurait aimé pouvoir lire en débutant avec Node.js. Il a volontairement simplifié ses explications de façon à être compréhensible par le plus grand nombre et son but est manifestement atteint.

Selon Ryan Dahl, créateur de Node.js (et pour ne citer que lui) : « il s'agit d'une formidable introduction à Node.js ».

Ce livre est la traduction de **The Node Beginner Book**.

À propos.....	3
Statut du document.....	3
Public concerné.....	3
Structure du livre.....	3
JavaScript et Node.js.....	4
JavaScript et vous.....	4
Un mot d'avertissement.....	4
JavaScript côté serveur.....	5
"Hello World".....	5
Une application Web complète avec Node.js.....	5
Les cas d'utilisation.....	5
Les couches de l'application.....	6
Développer les différents éléments.....	6
Un serveur HTTP basique.....	6
Analyse du serveur HTTP.....	7
Les fonctions paramètres.....	7
Passer une fonction à la création du serveur HTTP.....	8
Fonctions de rappel liées aux événements.....	9
Comment le serveur gère les requêtes.....	10
Trouver une place pour notre serveur.....	10
Router nos requêtes.....	12
Exécution au royaume des verbes.....	14
Router vers un vrai gestionnaire de requêtes.....	14
Faire répondre les gestionnaires.....	17
La mauvaise méthode.....	17
Bloquant et non bloquant.....	18
Répondre aux requêtes avec des opérations non bloquantes.....	21
Proposer quelque chose d'utile.....	23
Gérer les requêtes POST.....	23
Gérer le transfert de fichiers.....	27
Conclusion et remerciements.....	32

## À propos

Le but de ce document est de vous permettre de développer des applications avec Node.js en vous apprenant au fur et à mesure toutes les notions avancées dont vous aurez besoin avec JavaScript. Cela va bien au-delà d'un simple tutoriel du type « Hello World ».

## Statut du document

Vous lisez la version finale du livre, seules des mises à jour corrigeant des erreurs ou apportant des précisions sur des évolutions futures de Node.js seront apportées.

Les exemples du livre ont été testés avec la version 0.6.11.

## Public concerné

Ce livre conviendra le mieux aux lecteurs ayant un profil similaire au mien : maîtrise d'au moins un langage orienté-objet, comme Ruby, Python, PHP ou Java, quelques connaissances en JavaScript et inexpérimentés avec Node.js.

Cibler des lecteurs connaissant déjà d'autres langages de programmation signifie que nous ne couvrirons pas les notions basiques, comme les types de données, les variables, les structures de contrôle, etc. Vous devez déjà connaître ces fondamentaux pour comprendre le livre.

Par ailleurs, comme les fonctions et les objets sont différents en JavaScript - versus la plupart des autres langages - ceux-ci seront expliqués en détail.

## Structure du livre

Quand vous aurez terminé ce livre, vous aurez réalisé une application Web complète, dans laquelle les utilisateurs pourront voir des pages Web et uploader des fichiers.

Bien sûr, cela n'a rien de vraiment révolutionnaire, mais nous irons un peu plus loin que la simple écriture d'un code suffisant pour mettre en place ces fonctionnalités, nous allons en fait créer une bibliothèque simple (mais complète malgré tout) pour séparer proprement les différents aspects de notre application. Vous allez rapidement comprendre ce que je veux dire.

Nous allons commencer par regarder en quoi le développement JavaScript est différent avec Node.js de celui dans le navigateur.

Ensuite, nous nous plierons à la tradition en développement avec la mise en place d'une application « Hello World », qui est la plus simple utilisation fonctionnelle de Node.js.

Enfin, nous aborderons l'application réelle que nous souhaitons mettre en place. Nous verrons les différentes parties que nous devrons implémenter pour la mettre en œuvre et commencerons à travailler sur chacune de ces parties, pas à pas.

Comme promis, nous verrons au fur et à mesure certaines techniques avancées en JavaScript, comment les utiliser et pourquoi il est préférable d'utiliser ces concepts plutôt que ceux que nous connaissons dans d'autres langages.

Le code source de l'application finie est disponible sur [Sources](#) **le dépôt Github de NodeBeginnerBook**, vous pouvez aussi [Source](#) **télécharger l'archive francisée**.

## JavaScript et Node.js

### JavaScript et vous

Avant de rentrer dans les détails techniques, prenons un moment pour parler de vous et de votre rapport avec JavaScript. Ce chapitre va vous servir à déterminer s'il vous est utile de continuer la lecture de ce livre.

Si vous êtes comme moi, vous avez commencé le développement Web il y a quelques années en écrivant des pages HTML. Vous avez commencé à utiliser cette chose étrange appelée JavaScript, mais juste pour ajouter un peu d'interactivité ici ou là.

Ce que vous vouliez était surtout apprendre à créer des pages Web complexes ; vous avez donc appris des langages comme PHP, Ruby, Java pour écrire du code serveur.

Malgré tout, vous avez gardé un œil sur JavaScript et avez découvert, avec l'apparition de jQuery, Prototype ou autres bibliothèques, que JavaScript pouvait réaliser des choses avancées, bien au-delà de simples `window.open()`.

Pour autant, tout cela restait de la technologie côté client et même s'il est plaisant de pouvoir compter sur jQuery pour agrémenter son site, au bout du compte, vous n'étiez au final qu'un *utilisateur* de JavaScript, pas un *développeur* JavaScript.

Puis vint Node.js : du JavaScript côté serveur ! Voilà qui est prometteur.

Vous avez donc décidé de vous intéresser aux nouveautés de ce vieux JavaScript. Mais si écrire des applications avec Node.js est ce qui vous importe, comprendre comment le faire correctement signifie comprendre JavaScript et donc l'apprendre ; et cette fois pour de vrai !

Le problème est bien là : JavaScript a eu deux, peut-être même trois vies (le DHTML du milieu des années 90 pour s'amuser ; le plus mature créateur d'interfaces côté client, aidé par jQuery et autres ; maintenant, le langage serveur). Il n'est pas aisé de trouver des informations pour apprendre correctement JavaScript d'une façon permettant d'écrire des applications Node.js donnant le sentiment de *développer* en JavaScript et non juste de *l'utiliser*.

Car c'est bien le point-clé : vous êtes déjà un développeur expérimenté et vous ne souhaitez pas apprendre une nouvelle technologie en bidouillant et en l'utilisant mal, vous voulez être sûr de l'aborder sous le bon angle.

Il existe, bien entendu, d'excellentes documentations ici ou là. Mais la documentation seule n'est pas toujours suffisante. Ce dont vous avez plus besoin, c'est d'être guidé.

Mon but est d'être votre guide.

### Un mot d'avertissement

Il existe d'excellents spécialistes de JavaScript. Je ne suis pas de ceux-là.

Je suis réellement celui présenté au chapitre précédent. Je connais certaines choses en programmation d'applications Web, mais je ne connais le « vrai » JavaScript que depuis peu, de même pour Node.js. Je n'ai découvert certains aspects avancés de JavaScript que récemment. Je ne suis pas expérimenté.

C'est pour cela que ce livre n'est pas sous-titré « De débutant à expert », ce serait plus « De débutant à débutant expérimenté ».

Si je parviens à atteindre le but que je me suis fixé, alors ce livre sera celui que j'aurais aimé avoir en découvrant Node.js.

## JavaScript côté serveur

JavaScript a connu son essor dans le navigateur. Cependant, il ne s'agit que d'un contexte, qui vous dit ce que vous pouvez faire avec le langage, mais pas ce que le langage lui-même peut faire. JavaScript est un langage complet ; il peut être utilisé dans différents contextes et peut réaliser les mêmes choses que tout autre langage.

Node.js agit dans un autre contexte : il vous permet d'utiliser JavaScript côté serveur, en dehors du navigateur.

Avant de pouvoir exécuter votre JavaScript sur le serveur, le code a besoin d'être interprété. C'est précisément ce que fait Node.js en utilisant le moteur V8 de Google, celui qui est utilisé par le navigateur Chrome.

D'autre part, Node.js intègre un nombre important de modules utiles, ainsi, vous n'avez pas à écrire tout le code à partir de zéro, comme par exemple l'impression de texte dans la console.

Ainsi, Node.js comprend deux éléments distincts : un contexte d'exécution et une bibliothèque.

La première chose à faire pour utiliser Node.js est de l'installer. Je ne vais pas répéter la procédure d'installation et vous invite donc à consulter [la page officielle d'installation](#). Une fois que ce sera fait, vous serez prêt pour la suite de la lecture.

## "Hello World"

Nous pouvons maintenant nous lancer dans le grand bain et créer notre première application Node.js, le traditionnel « Hello World ».

Ouvrez votre éditeur favori et créez un fichier `helloworld.js`. Nous souhaitons afficher le message « Hello World » sur la sortie standard. Voici le code permettant de le faire :

```
console.log("Hello World");
```

Sauvegardez le fichier et exécutez-le avec Node.js :

```
node helloworld.js
```

Ceci devrait afficher le texte « Hello World » dans la console.

Bon, ceci n'est pas très amusant à vrai dire n'est-ce pas ? Alors commençons à écrire du vrai code.

## Une application Web complète avec Node.js

### Les cas d'utilisation

Nous les voulons simples, mais réalistes.

- L'utilisateur doit pouvoir utiliser notre application depuis un navigateur.
- L'utilisateur doit voir, depuis `http://domain/start`, une page d'accueil affichant un formulaire de transfert de fichier.
- En soumettant le formulaire après avoir choisi une image, celle-ci doit être uploadée à l'adresse `http://domain/upload` qui l'affichera une fois le transfert terminé.

Voilà qui est amplement suffisant. Bien sûr, vous pourriez y parvenir en bidouillant des solutions trouvées sur Google, mais ce n'est pas ce que nous souhaitons faire.

D'autre part, nous ne voulons pas nous contenter d'écrire le code minimal pour atteindre notre objectif, quand bien même ce code pourrait être correct et élégant ; nous allons volontairement ajouter plus de couches d'abstraction que nécessaire afin de nous entraîner pour le développement d'applications Node.js plus complexes.

## Les couches de l'application

Détaillons les besoins de notre application afin de savoir quelles parties nous aurons besoin d'implémenter afin de compléter les cas d'utilisation.

- Nous voulons servir des pages Web, nous avons donc besoin d'un **serveur HTTP**.
- Notre serveur devra répondre différemment en fonction de l'URL demandée, nous avons besoin pour cela d'une sorte de **routeur** pour attribuer aux requêtes les bonnes réponses.
- Pour satisfaire les requêtes reçues sur le serveur et qui ont été dirigées grâce au routeur, il nous faudra un **gestionnaire de requêtes**.
- Le routeur va aussi devoir traiter les données reçues par POST et les transmettre au gestionnaire de requête de façon appropriée, pour cela, nous aurons besoin d'un **gestionnaire de données**.
- Nous ne souhaitons pas uniquement traiter les URL reçues, nous voulons aussi afficher un contenu lorsque ces URL sont appelées, ce qui signifie qu'il nous faudra un **modèle de vue** qui sera utilisé par le gestionnaire de requêtes pour renvoyer des éléments au navigateur.
- Enfin, l'utilisateur devra pouvoir transférer des images, c'est-à-dire qu'il faudra un **gestionnaire d'upload** pour gérer cela.

Prenons un instant pour envisager comment nous pourrions créer cette application en PHP. Ce n'est logiquement un secret pour personne, la configuration typique sera un serveur Apache avec `mod_php5` installé.

En filigrane, cela signifie que toute la partie « nous devons pouvoir servir des pages Web et recevoir des requêtes HTTP » n'est pas à gérer directement en PHP.

Avec Node.js, c'est quelque peu différent : nous n'implémentons pas uniquement notre application mais aussi le serveur HTTP. En fait, l'application Web et son serveur se confondent.

Cela peut paraître un gros travail, mais comme nous le verrons bientôt, avec Node.js, ce n'est pas le cas.

Mais commençons par le commencement et créons le premier élément de l'application : le serveur HTTP.

## Développer les différents éléments

### Un serveur HTTP basique

Lorsque je suis arrivé au point de départ de ma première application « réelle » avec Node.js, je me suis bien sûr demandé comment je devais la coder, mais aussi comment organiser mon code.

Faut-il tout mettre dans un seul fichier ? La plupart des tutoriels que j'ai trouvés expliquant comment créer un serveur Web avec Node.js regroupent toute la logique au même endroit. Mais comment être sûr que mon code restera lisible au fur et à mesure que j'ajouterai des fonctionnalités ?

En fait, il est assez simple de séparer les différents éléments de votre code en les mettant dans des modules.

Cela permet d'avoir un fichier principal propre, que vous exécutez avec Node.js, couplé à des modules séparés, appelés par le fichier principal ou entre eux.

Nous allons donc créer un fichier principal pour lancer notre application et un fichier de module contenant notre serveur Web.

Selon moi, le standard est d'appeler le fichier principal `index.js`. Il est aussi logique de mettre le module serveur dans un fichier `server.js`.

Commençons par le module serveur. Créez un fichier `server.js` à la racine du répertoire de votre projet, contenant le code suivant :

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

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

C'est tout ! Vous venez juste d'écrire un serveur HTTP fonctionnel. Pour le vérifier, nous allons le lancer et le tester. D'abord, exécutez votre script avec Node.js :

```
node server.js
```

Maintenant, ouvrez votre navigateur à l'adresse <http://localhost:8888/>. Ce qui devrait afficher une page avec le message « Hello World ».

Plutôt intéressant n'est-ce pas ? Nous allons nous attarder un peu sur ce que nous venons de réaliser et reprendrons plus tard la question de l'organisation du code.

## Analyse du serveur HTTP

Détaillons donc un peu notre code.

La première ligne indique que nous avons besoin du module `http` inclus dans Node.js et qu'il sera disponible à travers la variable `http`.

Nous appelons ensuite la fonction `createServer()` présente dans le module `http`. Cette fonction retourne un objet, dont une des méthodes, `listen()`, prend une valeur numérique en paramètre correspondant au port que doit écouter le serveur.

Dans un premier temps, nous allons oublier la définition de fonction passée en paramètre de `http.createServer()`.

Nous aurions pu écrire le code démarrant le serveur et le faisant écouter le port 8888 comme suit :

```
var http = require("http");
var server = http.createServer();

server.listen(8888);
```

Ce code ne fait rien d'autre que de démarrer un serveur HTTP écoutant le port 8888 (il ne renvoie aucun résultat aux requêtes reçues).

La partie vraiment intéressante (et qui pourra sembler bizarre à ceux habitués à des langages plus conventionnels comme PHP) est la présence d'une définition de fonction comme paramètre de la fonction `createServer()`.

D'ailleurs, la définition de fonction est bien le seul paramètre passé lors de l'appel à `createServer()`. En effet, en JavaScript, les fonctions peuvent être passées en paramètre comme n'importe quelle autre valeur.

## Les fonctions paramètres

Vous pouvez par exemple faire quelque chose comme ça :

```
function say(word) {
```

```
console.log(word);
}

function execute(someFunction, value) {
  someFunction(value);
}

execute(say, "Hello");
```

Lisez le code bien attentivement ! Ce que nous faisons ici est de passer la fonction `say` comme premier paramètre à la fonction `execute`. Pas la valeur retournée par `say` mais bien `say` elle-même !

De la sorte, `say` est affectée à la variable locale `someFunction` dans `execute` et `execute` peut appeler cette fonction avec la syntaxe `someFunction()` (avec les parenthèses).

Bien entendu, comme `say` attend un paramètre, `execute` peut passer ce paramètre lors de l'appel de `someFunction`.

Il est possible, comme nous venons de le faire, de passer une fonction en paramètre en utilisant son nom. Ceci dit, il n'est pas nécessaire d'utiliser cette indirection (d'abord définir la fonction puis passer son nom), nous pouvons simplement passer comme paramètre la définition de la fonction elle-même :

```
function execute(someFunction, value) {
  someFunction(value);
}

execute(function(word) { console.log(word) }, "Hello");
```

Nous définissons la fonction à transmettre à `execute` lors de l'appel à celle-ci, là où doit se trouver le premier paramètre.

De cette façon, nous n'avons même pas besoin de donner un nom à cette fonction, c'est pourquoi on l'appelle une *fonction anonyme*.

Ceci est un premier exemple de ce que j'appelle du JavaScript avancé, mais n'allons pas trop vite. Pour le moment, souvenez-vous juste qu'en JavaScript, une fonction peut être passée comme paramètre lors de l'appel d'une autre fonction. Cela peut se faire en définissant une fonction et en passant son nom en paramètre ou alors en définissant directement la fonction lors de l'appel de la fonction englobante.

## Passer une fonction à la création du serveur HTTP

Nous pouvons désormais revenir à notre serveur rudimentaire :

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

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

Désormais, ce que nous faisons ici est compréhensible : nous passons à `createServer` une fonction anonyme.

Nous pourrions d'ailleurs arriver au même résultat avec le code suivant :

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

function onRequest(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}
```



```
}  
  
http.createServer(onRequest).listen(8888);
```

C'est peut-être le bon moment de se demander : pourquoi procédons-nous de cette façon ?

## Fonctions de rappel liées aux événements

La réponse tient en deux points :

- 1 Elle n'est pas facile à donner (du moins pour moi) ;
- 2 Elle repose sur la nature du fonctionnement de Node.js qui est événementiel, ce qui explique qu'il soit si rapide.

Je vous invite à lire l'excellent article de Felix Geisendörfer,  [Understanding node.js](#) pour plus d'explications.

Ce qu'il faut retenir, c'est que Node.js est événementiel. Bon, je le confesse, je ne saurais trop expliquer exactement ce que cela signifie. Mais je vais essayer d'expliquer en quoi cela nous intéresse si nous voulons écrire des applications Web avec Node.js.

Lorsque nous appelons la méthode `http.createServer()`, notre but n'est bien sûr pas uniquement de se contenter d'avoir un serveur qui écoute un port donné, nous souhaitons aussi accomplir une action lorsqu'une requête arrive à ce serveur.

Le problème est que cela arrive de façon asynchrone : une requête peut arriver n'importe quand et notre serveur utilise un processus unique.

Lorsque vous écrivez des applications en PHP, vous n'avez pas à vous soucier de ce genre de considération. Lorsqu'une requête HTTP arrive, le serveur Web (habituellement Apache) crée un nouveau processus et le script PHP associé est exécuté de façon séquentielle, c'est-à-dire en interprétant le code dans l'ordre dans lequel il est écrit.

Si l'on se place au niveau du contrôle du flux, nous sommes au beau milieu de notre application Node.js lorsqu'une requête arrive sur le port 8888 et que nous devons la traiter. Comment gérer cela sans perdre la raison ?

C'est précisément à ce niveau que la notion de programmation événementielle de Node.js / JavaScript intervient, bien que nous ayons besoin d'apprendre de nouveaux concepts pour bien le maîtriser. Voyons donc comment ces concepts sont appliqués dans le code de notre serveur.

Nous avons créé le serveur, en passant une fonction comme paramètre de la méthode de création. À chaque fois que notre serveur reçoit une requête, cette fonction sera appelée.

Nous ne savons pas quand une requête arrivera, mais nous avons maintenant défini un emplacement où nous pouvons la gérer. Cet endroit est notre fonction, que nous l'ayons définie au préalable ou qu'elle soit anonyme, cela est sans importance.

Ce concept s'appelle *fonction de rappel* (*callback*). Nous passons une fonction en paramètre d'une méthode et cette fonction est utilisée pour être rappelée (*called back*) lorsqu'un événement relatif à la méthode est déclenché.

Personnellement, ça m'a pris un peu de temps à vraiment comprendre ce concept, si vous ne vous sentez pas encore à l'aise avec ce principe, n'hésitez pas à relire l'article de Felix.

Amusons-nous un peu avec ce nouveau concept. Est-on en mesure de prouver que notre code continue à s'exécuter après avoir créé le serveur même si aucune requête HTTP n'est reçue et donc que la fonction de rappel n'est pas appelée ? Essayons ceci :

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


function onRequest(request, response) {
  console.log("Requête reçue.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}

http.createServer(onRequest).listen(8888);
console.log("Démarrage du serveur.");
```

Notez l'utilisation de `console.log()` pour afficher un message dès que `onRequest` (notre fonction de rappel) est appelée et un autre juste après avoir démarré le serveur.

Quand nous lançons ce code (`node server.js`, comme d'habitude), il va aussitôt afficher « Démarrage du serveur. » dans la console. Dès que nous appelons notre serveur (en ouvrant une page à l'adresse `http://localhost:8888`), le message « Requête reçue. » s'affiche à son tour.

Voilà notre JavaScript événementiel asynchrone et ses fonctions de rappel en action !

 Le message « Requête reçue. » va probablement s'afficher deux fois dans la console, cela est dû au fait que la plupart des navigateurs tentent de récupérer le favicon lorsqu'une page est ouverte en demandant le fichier `http://localhost:8888/favicon.ico`.

## Comment le serveur gère les requêtes

Regardons rapidement le reste du code de notre serveur, à savoir le contenu de la fonction de rappel `onRequest()`.

Lorsque l'événement intervient et que la fonction de rappel est lancée, deux paramètres lui sont passés : `request` et `response`.

Ce sont des objets et vous pouvez utiliser leurs différentes méthodes pour gérer de façon détaillée la requête reçue ainsi que la réponse à renvoyer (c'est-à-dire retourner par le réseau un résultat au navigateur ayant effectué la requête).

Notre code se contente actuellement de renvoyer des en-têtes HTTP `status` et `content-type` avec `response.writeHead()` et la méthode `response.write()` renvoie quant à elle le message « Hello World » comme corps de la réponse.

Enfin, nous appelons `response.end()` pour terminer la réponse.

À ce point, nous ne nous soucions pas de la nature de la requête, c'est pourquoi nous n'utilisons pas l'objet `request`.

## Trouver une place pour notre serveur

Je vous avais promis de revenir sur la façon d'organiser notre application. Nous avons le code de notre rudimentaire serveur HTTP dans le fichier `server.js`. Je vous avais indiqué qu'il était courant d'utiliser un fichier `index.js` pour amorcer et démarrer l'application en initialisant les autres modules nécessaires à l'application (comme le serveur HTTP dans `server.js`).

Voyons comment transformer `server.js` en véritable module Node.js que l'on pourra utiliser dans notre futur fichier `index.js`.

Vous l'avez probablement déjà remarqué, nous avons déjà utilisé des modules dans notre code :

```
var http = require("http");  
...  
http.createServer(...);
```

Quelque part dans Node.js, il existe un module appelé `http`. Nous pouvons l'utiliser dans notre propre code en l'important et en affectant le résultat de l'importation à une variable locale.

Cela fait de cette variable locale un objet possédant toutes les méthodes publiques mises à disposition par le module `http`.

Il est courant de prendre le nom du module comme nom de la variable locale, mais vous êtes libre de choisir le nom qu'il vous plaît :

```
var foo = require("http");  
...  
foo.createServer(...);
```

Bien, nous savons maintenant comment utiliser un module Node.js dans notre code. Mais comment créer notre propre module et comment l'utiliser ?

Pour cela, transformons notre script `server.js` en véritable module.

En fait, nous n'aurons pas à modifier grand-chose. Transformer un code en module revient à *exporter* les fonctionnalités de ce code que nous voulons rendre disponibles pour les scripts qui utiliseront ce module.

Pour l'instant, la fonctionnalité que notre serveur HTTP a besoin d'exporter est simple : les scripts qui utiliseront notre module s'en serviront juste pour démarrer le serveur.

Pour rendre cela possible, nous allons mettre le code du serveur dans une fonction appelée `start` et nous allons exporter cette fonction :

```
var http = require("http");  
  
function start() {  
  function onRequest(request, response) {  
    console.log("Request received.");  
    response.writeHead(200, {"Content-Type": "text/plain"});  
    response.write("Hello World");  
    response.end();  
  }  
  http.createServer(onRequest).listen(8888);  
  console.log("Démarrage du serveur.");  
}  
  
exports.start = start;
```

De cette façon, nous allons pouvoir créer notre fichier principal `index.js` et y démarrer notre serveur HTTP, bien que le code du serveur soit toujours dans le fichier `server.js`.

Créez un fichier `index.js` contenant le code suivant :

```
var server = require("./server");  
server.start();
```

Comme vous pouvez le voir, nous pouvons utiliser notre module serveur comme n'importe quel module interne, en important le fichier correspondant et en l'assignant à une variable. Les fonctions exportées par le module sont désormais accessibles dans le code.

Et voilà, nous pouvons maintenant lancer notre application avec le script principal et le fonctionnement est exactement le même :

```
node index.js
```

Parfait, nous pouvons maintenant mettre les différentes parties de notre application dans des fichiers distincts et les relier entre eux en les transformant en modules.

Cependant, nous n'avons pour le moment que le tout début de notre application en place : recevoir des requêtes HTTP. Nous avons besoin d'en faire quelque chose et de renvoyer des résultats différents en fonction de l'URL demandée par le navigateur à notre serveur.

Pour une application basique, nous pourrions faire cela directement dans la fonction de rappel `onRequest()`. Mais comme je l'ai déjà dit, autant apporter un peu plus de complexité afin de rendre notre exemple plus intéressant et complet.

Faire que des requêtes HTTP différentes impliquent des parties différentes du code s'appelle les *router* ; eh bien allons-y, créons un module `router`.

## Router nos requêtes

Nous avons besoin d'alimenter notre routeur avec les URL demandées ainsi que d'éventuels paramètres GET ou POST. Le routeur doit ensuite, en fonction de ces éléments, de décider quelles parties de code doivent être exécutées (le code à exécuter est la troisième partie de notre application : une collection de gestionnaires de requêtes pour faire le travail effectif lorsqu'une requête est reçue).

Nous devons donc analyser les requêtes HTTP pour en extraire l'URL demandée et les éventuels paramètres GET ou POST. L'on pourrait discuter de savoir si cela est du ressort du routeur ou du serveur (ou même d'un module à part), mais contentons-nous d'accepter qu'il s'agit d'une tâche dévolue au serveur HTTP pour l'instant.

Toutes les informations dont nous avons besoin sont disponibles à partir de l'objet `request` passé en premier paramètre de notre fonction de rappel `onRequest()`. Cependant, pour utiliser ces informations, nous avons besoin d'autres modules Node.js appelés `url` et `querystring`.

Le module `url` met à disposition des méthodes permettant d'extraire différents éléments d'une URL (par exemple, le *chemin* et la *chaîne de requête*). De son côté, `querystring` permet de récupérer les valeurs associées aux paramètres reçus dans la chaîne de requête :

```
url.parse(string).query
```

```
|
```

```
url.parse(string).pathname |
```

```
| |
```

```
| |
```

```
-----
```

```
http://localhost:8888/start?foo=bar&hello=world
```

```
--- -----
```

```
| |
```

```
| |
```

```
querystring(string) ["foo"] |
```

```
|  
querystring(string) ["hello"]
```

Bien sûr, nous pouvons aussi (comme nous le verrons ultérieurement), utiliser `querystring` pour récupérer les paramètres POST dans le corps de la requête.

Ajoutons à notre fonction `onRequest()` le mécanisme nécessaire pour connaître le chemin de l'URL demandée par le navigateur :

```
var http = require("http");  
var url = require("url");  
  
function start() {  
  function onRequest(request, response) {  
    var pathname = url.parse(request.url).pathname;  
    console.log("Requête reçue pour le chemin " + pathname + ".");  
    response.writeHead(200, {"Content-Type": "text/plain"});  
    response.write("Hello World");  
    response.end();  
  }  
  http.createServer(onRequest).listen(8888);  
  console.log("Démarrage du serveur.");  
}  
  
exports.start = start;
```

Parfait. Notre application sait maintenant différencier les requêtes à partir de l'URL demandée. Cela permet de rediriger les requêtes vers notre gestionnaire en fonction de son URL en utilisant notre routeur (qui est à écrire).

Dans le cadre de notre application, cela signifie que nous pourrions différencier le traitement pour les URL `/start` et `/upload`. Nous verrons bientôt comment mettre tout cela en place.

Voilà, il est maintenant temps d'écrire notre routeur. Créez un fichier appelé `router.js` avec le contenu suivant :

```
function route(pathname) {  
  console.log("Début du traitement de l'URL " + pathname + ".");  
}  
  
exports.route = route;
```

Bien entendu, ce code ne fait fondamentalement rien, mais c'est suffisant pour l'instant. Voyons déjà comment connecter ce routeur avec notre serveur avant d'ajouter plus de traitements dans le routeur.

Notre serveur HTTP a besoin de connaître et d'utiliser le routeur. Nous pourrions créer cette dépendance au niveau du serveur, mais comme c'est ce que l'on sait déjà faire avec d'autres langages, nous allons procéder plus finement pour injecter la dépendance (vous voudrez peut-être lire l'excellent article de Martin Fowler [🇬🇧 Inversion of Control Containers and the Dependency Injection pattern](#) avant d'aller plus loin).

Tout d'abord, complétons la fonction `start()` de notre serveur afin de pouvoir lui passer en paramètre la fonction `route` que nous utiliserons :

```
var http = require("http");  
var url = require("url");  
  
function start(route) {  
  function onRequest(request, response) {  
    var pathname = url.parse(request.url).pathname;  
    console.log("Requête reçue pour le chemin " + pathname + ".");  
    route(pathname);  
    response.writeHead(200, {"Content-Type": "text/plain"});  
    response.write("Hello World");  
  }  
  http.createServer(onRequest).listen(8888);  
  console.log("Démarrage du serveur.");  
}
```

```
response.end();
}
http.createServer(onRequest).listen(8888);
console.log("Démarrage du serveur.");
}

exports.start = start;
```

Complétons de même `index.js` en injectant la fonction `route` du routeur dans le serveur :

```
var server = require("./server");
var router = require("./router");

server.start(router.route);
```

Ici encore, nous passons une fonction en paramètre, mais ceci n'est plus vraiment nouveau.

Si nous lançons notre application maintenant (`node index.js`, comme toujours) et que vous appelez une URL, vous pouvez constater avec les sorties de notre application que le serveur HTTP utilise notre routeur et lui passe le chemin appelé dans l'URL :

```
bash$ node index.js
Requête reçue pour le chemin /foo.
Début du traitement de l'URL /foo.
```

J'ai volontairement omis certaines sorties inutiles (par exemple la requête pour le `/favicon.ico`).

## Exécution au royaume des verbes

Je vais encore une fois faire une petite digression et reparler de programmation fonctionnelle.

Passer des fonctions à d'autres fonctions n'est pas uniquement une considération technique. En gardant un œil sur le développement d'applications, ça devient presque philosophique. Comprenez bien ce que nous faisons : dans notre fichier `index`, nous pouvons passer l'objet `router` à notre serveur et le serveur peut appeler la méthode `route()` de cet objet `router`.

De cette façon, nous passons *quelque chose* et le serveur utilise cette *chose* pour *effectuer* une action.

Mais surtout, le serveur n'a pas besoin de cette chose, il a juste besoin de *faire* quelque chose et pour cela, vous n'avez aucunement besoin de choses, juste d'*actions*. Vous n'avez pas besoin de *noms*, uniquement de *verbes*.

Comprendre ce changement de mentalité, qui est au centre de cette idée, m'a permis de vraiment comprendre la programmation fonctionnelle.

En fait, j'ai vraiment tout compris en lisant le chef-d'œuvre de Steve Yegge 🇬🇧 **Execution in the Kingdom of Nouns**. Allez le lire dès maintenant. C'est réellement l'un des meilleurs articles que j'ai été amené à lire concernant la programmation.

## Router vers un vrai gestionnaire de requêtes

Revenons-en à nos moutons. Notre serveur et notre routeur sont désormais les meilleurs amis du monde et peuvent communiquer entre eux comme nous le souhaitions.

Ce n'est bien sûr pas suffisant. « Router » signifie gérer différemment les requêtes vers des URL différentes. Nous voulons en particulier avoir le traitement pour les requêtes vers `/start` gérées dans une autre fonction que celles demandant `/upload`.

Pour l'instant, le routage s'arrête au niveau du routeur, or le routeur n'est pas l'endroit adéquat pour faire quelque chose avec la requête, ce ne serait pas adapté lorsque notre application deviendra plus complexe.

Appelons les fonctions vers lesquelles les requêtes seront routées *gestionnaires de requêtes*. D'ailleurs, nous pouvons nous y attaquer, puisque tant qu'elles ne sont pas en place, il n'y a pas grand-chose à faire au niveau du routeur.

Nouvelle partie de l'application, nouveau module, rien de nouveau sous le soleil. Créons donc un module appelé `requestHandlers`, ajoutons une méthode `placeholder()` pour toutes les requêtes et exportons ceci comme méthode du module :

```
function start() {  
  console.log("Le gestionnaire 'start' est appelé.");  
}  
  
function upload() {  
  console.log("Le gestionnaire 'upload' est appelé.");  
}  
  
exports.start = start;  
exports.upload = upload;
```

Ceci nous permet de relier nos gestionnaires au routeur en donnant à ce dernier un endroit où router les requêtes.

À ce point, nous avons un choix à faire. Devons-nous coder en dur l'utilisation du module `requestHandlers` au niveau du routeur ou devons-nous injecter une nouvelle dépendance ? Si l'injection de dépendance ne devrait pas, comme tout modèle, être utilisé juste par principe, dans notre cas, il est cohérent de séparer le routeur du gestionnaire de requête, afin de rendre le routeur vraiment réutilisable.

Cela signifie que nous devons utiliser le serveur pour passer des gestionnaires de requêtes au routeur, mais cela semble une solution encore pire, c'est pourquoi nous devrions préférer les passer au serveur depuis le fichier principal puis les passer du serveur au routeur.

Comment faire pour les transmettre ? Pour l'instant, nous avons simplement deux gestionnaires, mais dans une application réelle, le nombre augmentera significativement et variera, or nous ne souhaitons pas bidouiller l'appel aux gestionnaires dès qu'une nouvelle URL et le gestionnaire associé est ajouté. Surtout, multiplier les instructions `if request == x then call handler y` au niveau du routeur serait assez affreux.

Un nombre variable de valeurs, chacune reliée à une chaîne (l'URL appelée) ? Cela ressemble à un tableau associatif qui serait tout à fait adapté.

Ceci dit, cette solution est assez frustrante, parce qu'en JavaScript, les tableaux associatifs n'existent pas... Ou alors ? En fait, ce sont précisément des objets que nous voulons utiliser, qui peuvent correspondre à des tableaux associatifs !

Il existe une bonne introduction à cela à l'adresse <http://msdn.microsoft.com/fr-fr/magazine/cc163419.aspx>. En voici un extrait :

« Dans C++ ou C#, lorsque nous parlons d'objets, nous faisons référence aux instances de classes ou « structs ». Les objets disposent de différentes propriétés et méthodes, selon le modèle (autrement dit, les classes) qui a servi à en créer une instance. Ce n'est pas le cas pour les objets JavaScript. Dans JavaScript, les objets sont simplement des collections de paires nom/valeur. Un objet JavaScript est semblable à un dictionnaire avec des chaînes-clés. »

Si les objets JavaScript sont juste des couples nom / valeur, comment peuvent-ils posséder des méthodes ? En fait, en JavaScript, les fonctions sont des valeurs comme les autres !

Revenons un peu au code. Nous avons décidé de transmettre la liste des gestionnaires à utiliser sous forme d'objet et pour pouvoir avoir un couplage faible, nous transmettons cet objet par `route()`.



Commençons par mettre cela en forme dans le fichier `index.js` :

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");
var handle = {};

handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;

server.start(router.route, handle);
```

Bien que `handle` soit une collection de gestionnaires de requêtes, je propose de lui donner un nom de verbe, car il en résultera une action claire au niveau du routeur, comme nous le verrons bientôt.

Comme vous pouvez le voir, il est assez simple d'utiliser le même gestionnaire pour des URL différentes. En précisant le chemin `"/"` associé à `requestHandlers.start`, nous pouvons indiquer de façon élégante que non seulement la requête `/start` mais aussi la requête `/` doivent être traitées par le gestionnaire `start`.

Après avoir défini notre objet, nous le transmettons au serveur comme un paramètre supplémentaire. Adaptions `server.js` pour pouvoir s'en servir :

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Requête reçue pour le chemin " + pathname + ".");
    route(handle, pathname);
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }
  http.createServer(onRequest).listen(8888);
  console.log("Démarrage du serveur.");
}

exports.start = start;
```

Nous avons ajouté le paramètre `handle` à la fonction `start()` et transmis l'objet `handle` à la fonction de rappel `route()` comme premier paramètre.

Changeons donc la fonction `route()` en rapport dans `router.js`:

```
function route(handle, pathname) {
  console.log("Début du traitement de l'URL " + pathname + ".");
  if (typeof handle[pathname] === 'function') {
    handle[pathname]();
  } else {
    console.log("Aucun gestionnaire associé à " + pathname);
  }
}

exports.route = route;
```

Ce que nous faisons ici est de vérifier s'il existe un gestionnaire lié à l'URL appelée. Si c'est le cas, nous appelons la fonction adaptée. Comme nous pouvons accéder aux fonctions de notre objet comme s'il s'agissait d'un tableau associatif, nous pouvons utiliser la syntaxe `handle[pathname]()`, assez élégante.



Voilà, c'est tout ce dont nous avons besoin pour lier le serveur, le routeur et les gestionnaires. En démarrant l'application et en appelant l'adresse <http://localhost:8888/start> depuis le navigateur, nous voyons que c'est le bon gestionnaire qui a été appelé :

```
Démarrage du serveur.  
Requête reçue pour le chemin /start.  
Début du traitement de l'URL /start.  
Le gestionnaire 'start' est appelé.
```

Si vous appelez <http://localhost:8888/> dans le navigateur, vous pouvez constater que cette requête fait aussi appel au gestionnaire `start`:

```
Requête reçue pour le chemin /.  
Début du traitement de l'URL /.  
Le gestionnaire 'start' est appelé.
```

## Faire répondre les gestionnaires

Magnifique ! Mais si les gestionnaires pouvaient renvoyer une réponse au navigateur, ce serait encore mieux non ?

Souvenez-vous, le message « Hello World » affiché par le navigateur lorsqu'il demande une page est toujours affiché depuis la fonction `onRequest()` du fichier `server.js`.

Après tout, « traiter des requêtes » signifie au final « répondre aux requêtes ». Nous devons donc permettre à nos gestionnaires de communiquer avec le navigateur, tout comme le fait la fonction `onRequest()`.

## La mauvaise méthode

La première approche que nous (les développeurs habitués à PHP ou Ruby) pourrions suivre est assez décevante : elle semble fonctionner correctement et être appropriée, mais tout d'un coup, alors que l'on ne s'y attend pas, elle se met à se comporter de façon inattendue.

Ce que j'entends par « première approche » est d'utiliser l'instruction `return` dans les gestionnaires pour renvoyer le texte à afficher à la fonction `onRequest()` qui elle transmettra cette réponse au navigateur.

Commençons par faire cela et regardons en quoi cette solution est mauvaise.

Tout d'abord, adaptons nos gestionnaires pour qu'ils retournent le texte à afficher dans le navigateur. Le fichier `requestHandlers.js` ressemble alors à ceci :

```
function start() {  
  console.log("Le gestionnaire 'start' est appelé.");  
  return "Bonjour Start";  
}  
  
function upload() {  
  console.log("Le gestionnaire 'upload' est appelé.");  
  return "Bonjour Upload";  
}  
  
exports.start = start;  
exports.upload = upload;
```

Bien. De façon similaire, le routeur doit renvoyer au serveur le texte reçu du gestionnaire. Pour cela, modifions `router.js` comme ceci :

```
function route(handle, pathname) {  
  console.log("Début du traitement de l'URL " + pathname + ".");
```

```
if (typeof handle[pathname] === 'function') {
    return handle[pathname]();
} else {
    console.log("Aucun gestionnaire associé à " + pathname);
    return "404 Non trouvé";
}

exports.route = route;
```

Comme vous pouvez le voir, nous retournons aussi un message si la requête n'a pas pu être routée.

Enfin, nous avons besoin de réorganiser notre serveur pour qu'il réponde au navigateur avec le texte retourné par le gestionnaire via le routeur. Le fichier `server.js` devient :

```
var http = require("http");
var url = require("url");

function start(route, handle) {
    function onRequest(request, response) {
        var pathname = url.parse(request.url).pathname;
        console.log("Requête reçue pour le chemin " + pathname + ".");
        response.writeHead(200, {"Content-Type": "text/plain"});
        var content = route(handle, pathname);
        response.write(content);
        response.end();
    }
    http.createServer(onRequest).listen(8888);
    console.log("Démarrage du serveur.");
}

exports.start = start;
```

Si nous démarrons l'application modifiée, tout semble se comporter correctement. Si nous appelons <http://localhost:8888/start>, le message « Bonjour Start » est bien affiché, de même si nous appelons <http://localhost:8888/upload>, nous obtenons « Bonjour Upload » et <http://localhost:8888/foo> affiche « 404 Non trouvé ».

Alors où est donc le problème ? La réponse courte : les soucis interviendront si l'un des gestionnaires doit effectuer des opérations non bloquantes.

Bon, c'est un peu léger comme explication. Penchons-nous plutôt sur la réponse longue.

## Bloquant et non bloquant

Comme nous l'avons dit, les problèmes surviennent lorsque les gestionnaires de requêtes incluent des opérations non bloquantes. Mais parlons d'abord des opérations bloquantes.

Plutôt que d'essayer d'expliquer ce que sont les opérations bloquantes et non bloquantes, voyons avec un exemple ce qu'il se passe si nous ajoutons des opérations bloquantes à notre gestionnaire de requêtes.

Pour cela, nous allons modifier le gestionnaire `start` et le faire attendre dix secondes avant de renvoyer son message « Bonjour Start ». Comme il n'existe pas en JavaScript d'équivalent au `sleep()` de PHP, nous allons utiliser une petite astuce.

Modifions `requestHandlers.js` comme ceci :

```
function start() {
    console.log("Le gestionnaire 'start' est appelé.");
    function sleep(milliseconds) {
        var startTime = new Date().getTime();
        while (new Date().getTime() < startTime + milliseconds);
    }
```

```
}  
sleep(10000);  
return "Bonjour Start";  
}  
  
function upload() {  
  console.log("Le gestionnaire 'upload' est appelé.");  
  return "Bonjour Upload";  
}  
  
exports.start = start;  
exports.upload = upload;
```

Pour clarifier les choses, expliquons ce que nous avons modifié. Lorsque la fonction `start()` est appelée, Node.js attend dix secondes puis retourne le message « Bonjour Start ». En revanche, la fonction `upload()` renvoie son résultat immédiatement.

Bien sûr, au lieu d'attendre dix secondes, il faut imaginer un traitement bloquant pour `start()`, comme des opérations très longues à effectuer.

Voyons ce que cela change.

Comme toujours, il faut d'abord redémarrer le serveur. Cette fois, vous allez devoir effectuer une manipulation un peu complexe pour pouvoir voir ce qu'il se passe. D'abord, ouvrez deux fenêtres de votre navigateur (ou deux onglets). Dans la première, entrez l'adresse <http://localhost:8888/start>, mais ne validez pas encore l'URL.

Dans la seconde fenêtre, entrez cette fois l'adresse <http://localhost:8888/upload> et là encore, ne validez pas tout de suite.

Maintenant, validez la première adresse ("`/start`") puis rapidement, passez à la seconde ("`/upload`") et validez aussi.

Voici ce que vous devriez constater : l'URL `/start` met dix secondes à afficher son résultat, comme attendu, mais l'URL `/upload` s'affiche aussi au bout de dix secondes bien qu'il n'y ait pas d'appel à `sleep()` dans ce gestionnaire !

Pourquoi ? Simplement parce que `start()` contient une opération bloquante et le blocage s'applique à toutes les autres opérations.

Le problème est là, car « avec Node.js, tout tourne en parallèle, excepté votre code ».

Cela signifie que Node.js est capable de gérer plusieurs actions concurrentes, mais il ne le fait pas en les séparant dans des processus distincts ; Node.js est *monothread*. Au lieu de cela, Node.js met en place une boucle d'événements que nous, développeurs, pouvons utiliser. Nous devrions éviter les opérations bloquantes au profit d'opérations non bloquantes.

Pour faire cela, nous devons utiliser les fonctions de rappel en passant comme paramètres de fonctions d'autres fonctions susceptibles de mettre du temps à accomplir leur tâche (comme attendre dix secondes, faire des appels à des bases de données ou effectuer des calculs complexes).

C'est une façon de dire : « Tiens, `fonctionPotentiellementLongueAExecuter()`, fais telles actions, mais moi, le processus unique de Node.js, je ne vais pas attendre que tu aies fini, je vais continuer à exécuter le code qui te suit, donc prend `fonctionDeRappel()` et appelle-la lorsque ton travail sera accompli. Merci. »

Si vous souhaitez avoir plus de détails là-dessus, je vous conseille cet article sur le blog de Mixu : [🇬🇧 Understanding the node.js event loop](#).

Nous allons maintenant voir pourquoi la façon dont nous avons codé les réponses renvoyées par le gestionnaire de requêtes ne permet pas l'utilisation correcte d'opérations non bloquantes.

Encore une fois, nous allons partir d'un exemple en modifiant le code de l'application.

Nous allons, encore une fois, modifier le gestionnaire `start()`. Nous modifions donc `requestHandlers.js` comme ceci :

```
var exec = require("child_process").exec;

function start() {
  console.log("Le gestionnaire 'start' est appelé.");
  var content = "vide";
  exec("ls -lah", function (error, stdout, stderr) {
    content = stdout;
  });
  return content;
}

function upload() {
  console.log("Le gestionnaire 'upload' est appelé.");
  return "Bonjour Upload";
}

exports.start = start;
exports.upload = upload;
```

Comme vous pouvez le voir, nous avons introduit un nouveau module Node.js, `child_process`. Nous l'utilisons car il permet de mettre en place une opération non bloquante très simple (mais particulièrement utile) : `exec()`.

La méthode `exec()` permet d'exécuter une commande `shell` depuis Node.js. Dans notre exemple, nous l'utilisons pour obtenir la liste de tous les fichiers du répertoire ("`ls -lah`") ce qui nous permet d'afficher cette liste dans le navigateur si l'utilisateur demande l'URL `/start`.

Ce que fait notre code est assez simple : on initialise une nouvelle variable `content` avec la valeur "vide" puis on exécute l'instruction "`ls -lah`" dont le résultat est affecté à `content` et l'on retourne cette variable.

Comme d'habitude, démarrons l'application et appelons l'adresse `http://localhost:8888/start`.

Cette adresse affiche une magnifique page avec la chaîne "vide". Que se passe-t-il donc ?

Comme vous l'aurez probablement déjà deviné, `exec()` effectue son travail de façon non bloquante, ce qui permet d'effectuer des actions système gourmandes (comme par exemple, copier de gros fichiers) sans pour autant bloquer le reste de l'application comme notre fonction `sleep()` le faisait.

Si vous voulez vérifier cela, remplacez "`ls -lah`" par une opération un peu plus lourde comme "`find /`".

Mais nous ne pouvons pas nous estimer satisfaits de cette élégante solution non bloquante si le navigateur n'affiche pas le bon résultat.

Nous allons donc réparer cela et tant qu'on y est, en profiter pour comprendre pourquoi la solution actuelle fonctionne mal.

Le problème est qu'`exec()`, pour pouvoir fonctionner de façon non bloquante, utilise une fonction de rappel.

Dans notre exemple, c'est une fonction anonyme qui est passée en deuxième paramètre lors de l'appel de la fonction `exec()`:

```
function (error, stdout, stderr) {
  content = stdout;
}
```


C'est ici que réside la base de notre problème : notre propre code est exécuté de façon synchrone, ce qui signifie que tout de suite après avoir appelé la fonction `exec()`, Node.js continue et exécute l'instruction `return content`; or à ce moment, `content` vaut toujours "vide" puisque `exec()` fonctionne de façon asynchrone, sa fonction de rappel n'a pas encore renvoyé son résultat.

L'instruction `"ls -lah"` n'est pas très coûteuse et est donc assez rapide (à moins bien sûr que vous ayez des millions de fichiers dans le répertoire). C'est ce qui explique que la fonction de rappel est appelée assez rapidement, mais cela reste néanmoins asynchrone.

Envisager une commande plus coûteuse rend tout cela plus évident. Sur ma machine, la commande `"find /"` prend plus d'une minute à s'exécuter, mais si je remplace `"ls -lah"` par `"find /"` dans le gestionnaire de requêtes, je reçois toujours immédiatement la réponse lorsque je demande l'URL `/start`. Il est clair que `exec()` fait quelque chose en tâche de fond, mais Node.js continue lui à interpréter le code de l'application et l'on peut supposer que la fonction de rappel passée à `exec()` sera appelée lorsque `"find /"` aura terminé son exécution.

Mais alors comment pourrions-nous obtenir ce que nous souhaitons ? C'est-à-dire afficher à l'utilisateur la liste des fichiers du répertoire courant.

Bien, après avoir vu *comment ne pas le faire*, nous allons voir comment permettre à notre gestionnaire de répondre correctement.

 Sur les systèmes Windows, vous devrez remplacer les commandes `"ls -lah"` et `"find /"` par des commandes DOS (par exemple `"dir"`) pour tester ces exemples.

## Répondre aux requêtes avec des opérations non bloquantes

Je viens de dire « correctement ». C'est une affirmation assez périlleuse. La plupart du temps, il n'y a pas une seule solution correcte.

Pour autant, une solution possible, comme toujours avec Node.js, est d'utiliser une fonction de rappel. Regardons ça de plus près.

Pour l'instant, notre application est capable d'envoyer le contenu à afficher à l'utilisateur depuis les gestionnaires de requêtes au serveur HTTP en le retournant au travers des différentes couches de l'application (gestionnaires → routeur → serveur).

Notre nouvelle approche va consister à ne plus transporter le contenu vers le serveur, mais de rendre le serveur disponible au contenu. Pour être plus précis, nous allons injecter l'objet `response` (depuis la fonction de rappel `onRequest()` de notre serveur) dans notre gestionnaire en passant par le routeur. Chaque gestionnaire pourra alors utiliser les méthodes de cet objet pour répondre directement aux requêtes.

Mais assez d'explications, voici la recette pas à pas pour modifier notre application.

Commençons par `server.js` :

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Requête reçue pour le chemin " + pathname + ".");
    route(handle, pathname, response);
  }
  http.createServer(onRequest).listen(8888);
  console.log("Démarrage du serveur.");
}
```

```
exports.start = start;
```

Au lieu d'attendre une valeur de retour de la fonction `route()`, nous allons lui passer un troisième paramètre : l'objet `response`. D'autre part, nous avons supprimé tous les appels aux méthodes de `response` puisque maintenant, c'est `route` qui va les gérer.

Ensuite, `router.js` :

```
function route(handle, pathname, response) {
  console.log("Début du traitement de l'URL " + pathname + ".");
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response);
  } else {
    console.log("Aucun gestionnaire associé à " + pathname);
    response.writeHead(404, {"Content-Type": "text/plain"});
    response.write("404 Non trouvé");
    response.end();
  }
}

exports.route = route;
```

Le principe est le même : au lieu d'attendre une valeur de retour de nos gestionnaires, nous leur passons l'objet `response`.

Si aucun gestionnaire ne peut être associé à la requête, alors nous prenons soin de renvoyer les bons en-têtes "404" et contenu.

Enfin, nous modifions `requestHandlers.js` :

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Le gestionnaire 'start' est appelé.");
  exec("ls -lah", function (error, stdout, stderr) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write(stdout);
    response.end();
  });
}

function upload(response) {
  console.log("Le gestionnaire 'upload' est appelé.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Bonjour Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Les fonctions du gestionnaire doivent recevoir le paramètre `response` et l'utiliser afin de renvoyer directement la réponse souhaitée.

Le gestionnaire `start` répond depuis la fonction de rappel anonyme passée à `exec()`, le gestionnaire `upload`, quant à lui, se contente toujours de répondre « Bonjour Upload », mais maintenant en utilisant l'objet `response`.

Si nous redémarrons l'application (`node index.js`), cela devrait fonctionner comme attendu.

Pour se persuader qu'une opération lourde effectuée par `/start` ne bloquera plus les requêtes vers `/upload` et ne les empêchera pas de répondre immédiatement, modifiez le fichier `requestHandlers.js` comme ceci :

```
var exec = require("child_process").exec;

function start(response) {
  console.log("Le gestionnaire 'start' est appelé.");
  exec("find /",
    { timeout: 10000, maxBuffer: 20000*1024 },
    function (error, stdout, stderr) {
      response.writeHead(200, {"Content-Type": "text/plain"});
      response.write(stdout);
      response.end();
    });
}

function upload(response) {
  console.log("Le gestionnaire 'upload' est appelé.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Bonjour Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Ceci va entraîner que les requêtes vers <http://localhost:8888/start> prendront au moins dix secondes pour répondre, alors que celles vers <http://localhost:8888/upload> répondront immédiatement, même si `/start` est toujours en cours de traitement.

## Proposer quelque chose d'utile

Jusqu'à présent, ce que nous avons fait est fonctionnel et élégant, mais nous n'avons rien créé d'utile pour les utilisateurs de notre site.

Nos serveur, routeur et gestionnaire de requêtes sont en place, désormais, nous pouvons ajouter du contenu à notre site de façon à ce que les utilisateurs puissent l'utiliser comme nous l'avons envisagé, c'est-à-dire en ayant la possibilité de sélectionner un fichier, l'envoyer au serveur et enfin le visualiser dans la page. Pour plus de simplicité, nous n'allons autoriser que le transfert d'images dans l'application.

Nous allons bien sûr procéder étape par étape, mais avec toutes les techniques JavaScript déjà vues, nous allons pouvoir accélérer un petit peu. Enfin, je sais que malgré tout, j'aime bien m'écouter parler !

Ici, étape par étape signifie deux étapes principales. D'abord, nous allons voir comment gérer les requêtes POST arrivantes (mais pas encore les chargements de fichier) puis nous utiliserons un module externe de Node.js pour traiter le transfert de fichier. J'ai choisi cette approche pour deux raisons.

D'abord, la gestion de base des requêtes POST est relativement simple avec Node.js, mais cela nous enseigne suffisamment de choses pour que ça vaille le coup de l'aborder.

Ensuite, le transfert de fichier n'est au contraire pas simple du tout avec Node.js. Du coup, c'est en dehors du cadre de ce tutoriel, mais l'utilisation d'un module externe est elle suffisamment pertinente pour être dans l'esprit de ce tutoriel pour débutant.

## Gérer les requêtes POST

Restons très basiques : nous allons juste afficher un `textarea` qui sera rempli puis envoyé au serveur dans une requête POST. Une fois la requête reçue et traitée, nous afficherons le contenu du `textarea`.

Le code HTML associé au `textarea` doit être envoyé par le gestionnaire `/start`. Ajoutons-le dès maintenant dans le fichier `requestHandlers.js` :

```
function start(response) {
```



```
console.log("Le gestionnaire 'start' est appelé.");
var body = '<html>'+
  '<head>'+
  '<meta http-equiv="Content-Type" content="text/html; '+
  'charset=UTF-8" />'+
  '</head>'+
  '<body>'+
  '<form action="/upload" method="post">'+
  '<textarea name="text" rows="20" cols="60"></textarea>'+
  '<input type="submit" value="Envoyer" />'+
  '</form>'+
  '</body>'+
  '</html>';
response.writeHead(200, {"Content-Type": "text/html"});
response.write(body);
response.end();
}

function upload(response) {
  console.log("Le gestionnaire 'upload' est appelé.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Bonjour Upload");
  response.end();
}

exports.start = start;
exports.upload = upload;
```

Si cette page n'est pas lauréate aux trophées du Web, je ne vois pas ce qu'il manque ! Vous devriez voir le formulaire en appelant <http://localhost:8888/start> dans votre navigateur. Si ce n'est pas le cas, c'est probablement que vous avez oublié de redémarrer l'application.

J'en entends certains commencer à râler : « Quelle horreur de mettre le contenu dans le gestionnaire de requêtes ! » C'est vrai, mais j'ai fait le choix de ne pas développer cette nouvelle abstraction (séparer la vue du contrôleur) dans ce tutoriel parce que selon moi, cela n'apporte rien de spécifique à la programmation JavaScript ou Node.js.

Au lieu de cela, penchons-nous sur quelque chose de plus intéressant et utilisons l'espace vide pour afficher le résultat du traitement de notre requête POST reçue à la soumission du formulaire.

Maintenant que nous sommes devenus débutants experts, nous ne sommes plus étonnés que le traitement des données POST se fera de façon non bloquante en utilisant une fonction de rappel asynchrone.

Cela n'a rien d'aberrant puisque les données passées dans une requête POST (et leur traitement) peuvent être assez volumineuses, rien n'interdit à l'utilisateur d'écrire de véritables romans ! Traiter cette masse potentielle de données directement serait une opération bloquante.

Pour rendre l'ensemble du traitement non bloquant, Node.js récupère et fournit à notre code les données POST par paquets, les fonctions de rappels réagissant à différents événements. Ces événements sont `data` (lorsqu'un nouveau paquet est disponible) et `end` (lorsque toutes les données ont été reçues).

Nous devons indiquer à Node.js quelles fonctions appeler lorsque ces événements sont déclenchés. Ceci est réalisé en ajoutant des *écouteurs* à l'objet `request` passés à la fonction de rappel `onRequest` chaque fois qu'une requête HTTP est reçue.

Voilà à quoi cela peut ressembler :

```
request.addListener("data", function(chunk) {
  // traitement lorsqu'un paquet est reçu
});

request.addListener("end", function() {
  // traitement lorsque toutes les données sont arrivées
});
```



La question qui se pose est de savoir où placer cette logique. Actuellement, nous n'avons accès à l'objet `request` qu'au niveau du serveur puisque nous ne le passons pas au routeur (et donc pas au gestionnaire de requêtes non plus) comme nous l'avons fait avec l'objet `response`.

À mon sens, c'est une tâche dévolue au serveur de fournir au reste de l'application les données de la requête utiles au reste du traitement. C'est pour cela que je préconise que les données POST soient traitées au niveau du serveur et de passer le résultat au routeur et au gestionnaire de requêtes qui pourront décider ce qu'il faut en faire.

L'idée est donc de placer les fonctions de rappel liées à `data` et `end` au niveau du serveur. Nous utilisons `data` pour réunir les paquets de données et `end` pour transmettre les données collectées au gestionnaire de requêtes, en passant par le routeur.

Voici comment, en commençant par `server.js` :

```
var http = require("http");
var url = require("url");

function start(route, handle) {
  function onRequest(request, response) {
    var postData = "";
    var pathname = url.parse(request.url).pathname;
    console.log("Requête reçue pour le chemin " + pathname + ".");
    request.setEncoding("utf8");
    request.addListener("data", function(postDataChunk) {
      postData += postDataChunk;
      console.log("Paquet POST reçu '" + postDataChunk + "'.");
    });
    request.addListener("end", function() {
      route(handle, pathname, response, postData);
    });
  }
  http.createServer(onRequest).listen(8888);
  console.log("Démarrage du serveur.");
}

exports.start = start;
```

Nous faisons principalement trois choses. D'abord, nous précisons que nous attendons des données au format utf-8, ensuite, nous avons ajouté un écouteur pour l'événement `data` qui remplit au fur et à mesure la variable `postData` avec les paquets reçus, enfin, nous avons déplacé l'appel au routeur au niveau de la fonction de rappel de l'écouteur de l'événement `end` pour être sûrs qu'il ne sera appelé que lorsque toutes les données auront été reçues. Bien entendu, nous en profitons aussi pour transmettre les données POST au routeur puisque nous en aurons besoin dans le gestionnaire de requêtes.

Ajouter une inscription dans la console n'est pas nécessairement opportun (nous recevons peut-être des mégaoctets de données), mais cela est utile pour voir ce qu'il se passe.

Nous pouvons en profiter pour regarder un peu le fonctionnement. Testez le `textarea` avec alternativement des textes courts et longs et vous constaterez que plus le texte est long, plus il y a d'appels à la fonction de rappel de `data`.

Allez, rendons l'application encore plus impressionnante. Sur la page `/upload`, nous allons maintenant afficher le texte reçu. Pour cela, nous devons transmettre `postData` au gestionnaire de requêtes depuis `router.js` :

```
function route(handle, pathname, response, postData) {
  console.log("Début du traitement de l'URL " + pathname + ".");
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response, postData);
  } else {
    console.log("Aucun gestionnaire associé à " + pathname);
    response.writeHead(404, {"Content-Type": "text/plain"});
    response.write("404 Non trouvé");
    response.end();
  }
}
```

```
}  
}  
  
exports.route = route;
```

Puis, dans `requestHandlers.js`, nous ajoutons les données transmises dans la réponse associée à `upload` :

```
function start(response, postData) {  
  console.log("Le gestionnaire 'start' est appelé.");  
  var body = '<html>'+  
    '<head>'+  
    '<meta http-equiv="Content-Type" content="text/html; '+  
    'charset=UTF-8" />'+  
    '</head>'+  
    '<body>'+  
    '<form action="/upload" method="post">'+  
    '<textarea name="text" rows="20" cols="60"></textarea>'+  
    '<input type="submit" value="Envoyer" />'+  
    '</form>'+  
    '</body>'+  
    '</html>';  
  response.writeHead(200, {"Content-Type": "text/html"});  
  response.write(body);  
  response.end();  
}  
  
function upload(response, postData) {  
  console.log("Le gestionnaire 'upload' est appelé.");  
  response.writeHead(200, {"Content-Type": "text/plain"});  
  response.write("Vous avez envoyé : " + postData);  
  response.end();  
}  
  
exports.start = start;  
exports.upload = upload;
```

C'est fait, nous savons maintenant recevoir des données POST et les utiliser dans nos gestionnaires de requêtes.

Un dernier détail reste à régler. Ce que nous envoyons au routeur puis au gestionnaire de requêtes, c'est l'intégralité des données POST reçues. Or ce qui nous intéresse est en réalité uniquement le contenu renseigné dans chaque champ de saisie (donc sans les noms des différents champs).

Nous avons déjà évoqué le module `querystring`, qui va nous aider à corriger cela :

```
var querystring = require("querystring");  
  
function start(response, postData) {  
  console.log("Le gestionnaire 'start' est appelé.");  
  var body = '<html>'+  
    '<head>'+  
    '<meta http-equiv="Content-Type" content="text/html; '+  
    'charset=UTF-8" />'+  
    '</head>'+  
    '<body>'+  
    '<form action="/upload" method="post">'+  
    '<textarea name="text" rows="20" cols="60"></textarea>'+  
    '<input type="submit" value="Envoyer" />'+  
    '</form>'+  
    '</body>'+  
    '</html>';  
  response.writeHead(200, {"Content-Type": "text/html"});  
  response.write(response.write(body));  
  response.end();  
}  
  
function upload(response, postData) {  
  console.log("Le gestionnaire 'upload' est appelé.");  
  response.writeHead(200, {"Content-Type": "text/plain"});
```

```
response.write("Vous avez envoyé : "+ querystring.parse(postData).text);
response.end();
}

exports.start = start;
exports.upload = upload;
```

Voilà, pour un guide du débutant, c'est amplement suffisant concernant le traitement de données POST.

## Gérer le transfert de fichiers

Abordons maintenant l'aspect final de notre étude de cas. Notre objectif est de permettre à l'utilisateur d'uploader une image puis d'afficher cette image dans le navigateur.

Dans les années 90, cela aurait pu être un modèle économique pour une introduction en bourse ! Aujourd'hui, c'est uniquement le moyen de découvrir comment installer et utiliser un module externe Node.js dans son code.

Le module externe que nous allons utiliser est `node-formidable` de Felix Geisendörfer. Il permet de se passer de tous les détails pénibles liés à la réception de fichiers. Au final, recevoir des fichiers revient presque au même que recevoir des données POST, mais dans ce cas de figure, les détails deviennent presque démoniaques et se servir d'une solution clés en main prend tout son sens.

Pour pouvoir se servir du code de Felix, le module doit d'abord être installé. Node.js incorpore un gestionnaire de paquetages appelé NPM. Il nous permet d'installer des modules externes de façon très simple. Depuis Node.js, il suffit juste d'appeler

```
npm install formidable
```

en ligne de commande. Si vous obtenez le message suivant :

```
npm info build Success: formidable@1.0.2
npm ok
```

C'est que tout est bon (attention, sous Windows, le message peut être différent).

Le module `formidable` est maintenant disponible pour notre code, tout ce que nous avons à faire est de l'importer, comme pour tout autre module interne :

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

Le module `formidable` rend tous les éléments de formulaire reçus depuis une requête HTTP de type POST facilement accessibles depuis Node.js. Tout ce que nous avons à faire est de créer un objet `IncomingForm` qui est une représentation du formulaire reçu et qui peut être parcouru depuis l'objet `request` de notre serveur HTTP, autant pour les champs du formulaire que pour les fichiers.

Le code d'exemple présenté sur la page du projet `node-formidable` montre comment tous les éléments se mettent en place :

### Exemple d'utilisation de node-formidable


```
var formidable = require('formidable'),
    http = require('http'),
    sys = require('sys'); // ou 'util', voir note

http.createServer(function(req, res) {
  if (req.url == '/upload' && req.method.toLowerCase() == 'post') {
    // parse a file upload
    var form = new formidable.IncomingForm();
```

### Exemple d'utilisation de node-formidable

```
form.parse(req, function(err, fields, files) {
  res.writeHead(200, {'content-type': 'text/plain'});
  res.write('received upload:\n\n');
  res.end(sys.inspect({fields: fields, files: files}));
});
return;
}

// show a file upload form
res.writeHead(200, {'content-type': 'text/html'});
res.end(
  '<form action="/upload" enctype="multipart/form-data" '+
  'method="post">'+
  '<input type="text" name="title"><br>'+
  '<input type="file" name="upload" multiple="multiple"><br>'+
  '<input type="submit" value="Upload">'+
  '</form>'
);
}).listen(8888);
```

 Pour les versions récentes de Node.js, le module `sys` s'appelle désormais `util`. Vous aurez peut-être besoin de corriger le code dans ce sens.

Si nous mettons ce code dans un fichier et que nous l'exécutons, nous pouvons soumettre un formulaire simple comprenant un champ `file`. Nous pouvons voir comment l'objet `files`, passé à la fonction de rappel définie, est organisé :

```
received upload:
{ fields: { title: 'Hello World' },
  files:
    { upload:
      { size: 1558,
        path: '/tmp/1c747974a27a6292743669e91f29350b',
        name: 'us-flag.png',
        type: 'image/png',
        lastModifiedDate: Tue, 21 Jun 2011 07:02:41 GMT,
        _writeStream: [Object],
        length: [Getter],
        filename: [Getter],
        mime: [Getter] } } }
```

Pour atteindre notre objectif, nous avons besoin d'inclure le traitement par formidable des éléments de formulaire reçus dans le code de notre application. D'autre part, nous devrons aussi trouver comment renvoyer l'image récupérée (qui est sauvegardée dans le répertoire `/tmp`) au navigateur.

Nous allons donc utiliser notre serveur pour lire le contenu du fichier. Sans surprise, il existe un module pour ça : `fs`.

Ajoutons un nouveau gestionnaire pour l'URL `/show`. Celui-ci va afficher le contenu du fichier `/tmp/test.png` par programmation. Bien entendu, il est préférable de mettre un fichier PNG à cet emplacement au préalable.

Modifions le fichier `requestHandlers` comme ceci :

```
var querystring = require("querystring"),
    fs = require("fs");

function start(response, postData) {
  console.log("Le gestionnaire 'start' est appelé.");
  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" method="post">'+
```

```
'<textarea name="text" rows="20" cols="60"></textarea>' +
'<input type="submit" value="Envoyer" />' +
'</form>' +
'</body>' +
'</html>';
response.writeHead(200, {"Content-Type": "text/html"});
response.write(body);
response.end();
}

function upload(response, postData) {
  console.log("Le questionnaire 'upload' est appelé.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Vous avez envoyé : " + querystring.parse(postData).text);
  response.end();
}

function show(response, postData) {
  console.log("Le questionnaire 'show' est appelé.");
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
    if(error) {
      response.writeHead(500, {"Content-Type": "text/plain"});
      response.write(error + "\n");
      response.end();
    } else {
      response.writeHead(200, {"Content-Type": "image/png"});
      response.write(file, "binary");
      response.end();
    }
  });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

Nous devons aussi prévoir le traitement de l'URL `/show` dans `index.js` :

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");
var handle = {}

handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;
handle["/show"] = requestHandlers.show;

server.start(router.route, handle);
```

En redémarrant le serveur et en appelant l'URL <http://localhost:8888/show>, le navigateur affiche bien l'image `/tmp/test.png`.

Parfait. Tout ce qu'il nous reste désormais à faire est :

- ajouter un champ de formulaire de type fichier sur la page `/start` ;
- intégrer `node-formidable` au questionnaire `upload` pour sauvegarder le fichier reçu à l'emplacement `/tmp/test.png` ;
- renvoyer l'image récupérée dans le contenu HTML de `/upload`.

La première étape est relativement simple. Nous ajoutons un attribut `multipart/form-data` au formulaire, nous supprimons le `textarea` que nous remplaçons par un champ de type `file` et nous changeons le libellé du bouton `submit` en « Transférer le fichier ». Cela se passe dans le fichier `requestHandlers.js` :

```
var querystring = require("querystring"),
    fs = require("fs");
```

```
function start(response, postData) {
  console.log("Le gestionnaire 'start' est appelé.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="file" name="upload">'+
    '<input type="submit" value="Transférer le fichier" />'+
    '</form>'+
    '</body>'+
    '</html>';
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(body);
  response.end();
}

function upload(response, postData) {
  console.log("Le gestionnaire 'upload' est appelé.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Vous avez envoyé : " + querystring.parse(postData).text);
  response.end();
}

function show(response, postData) {
  console.log("Le gestionnaire 'show' est appelé.");
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
    if(error) {
      response.writeHead(500, {"Content-Type": "text/plain"});
      response.write(error + "\n");
      response.end();
    } else {
      response.writeHead(200, {"Content-Type": "image/png"});
      response.write(file, "binary");
      response.end();
    }
  });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

L'étape suivante est un peu plus compliquée. Un premier problème se pose : nous voulons gérer la récupération de l'image dans le gestionnaire `upload`, nous devons donc passer l'objet `request` à la méthode `form.parse` de `node-formidable`.

Or tout ce que nous avons est l'objet `response` et le tableau `postData`. Nous allons donc devoir passer l'objet `request` du serveur au routeur puis au gestionnaire de requêtes. Ce n'est peut-être pas la solution la plus élégante, mais elle répond convenablement à nos besoins.

Tant que nous y sommes, nous allons aussi supprimer toute la logique liée à `postData` du serveur et du gestionnaire de requêtes. En effet, nous n'en avons pas besoin pour le traitement du fichier reçu, surtout, il pose un problème supplémentaire : nous avons déjà utilisé les données issues des événements `data` de l'objet `request` au niveau du serveur, ce qui signifie que `form.parse`, qui a aussi besoin de ces données, ne pourra plus rien recevoir des événements `data` car Node.js ne bufferise pas les données.

Commençons par `server.js`. Nous supprimons le traitement de `postData` et l'instruction `request.setEncoding` (qui est désormais gérée par `node-formidable`) et nous ajoutons la transmission de l'objet `request` au routeur :

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

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

function start(route, handle) {
  function onRequest(request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Requête reçue pour le chemin " + pathname + ".");
    route(handle, pathname, response, request);
  }

  http.createServer(onRequest).listen(8888);
  console.log("Démarrage du serveur.");
}

exports.start = start;
```

Modifions maintenant router.js. Nous n'avons plus besoin de transmettre postData, à la place, nous envoyons request :

```
function route(handle, pathname, response, request) {
  console.log("Début du traitement de l'URL " + pathname + ".");
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response, request);
  } else {
    console.log("Aucun gestionnaire associé à " + pathname);
    response.writeHead(404, {"Content-Type": "text/html"});
    response.write("404 Non trouvé");
    response.end();
  }
}

exports.route = route;
```

Maintenant, l'objet `request` est disponible pour le gestionnaire `upload`. C'est `node-formidable` qui se charge de sauvegarder le fichier reçu dans le répertoire local `/tmp`, mais nous devons nous charger nous-même de le renommer en `test.png`. Bien sûr, nous simplifions à l'extrême en considérant que seuls des fichiers PNG seront transmis.

Il y a quelques pièges à éviter lors du renommage. Sous Windows, Node.js n'est pas capable d'écraser un fichier existant, dans ce cas (si le renommage soulève une erreur), nous devons d'abord supprimer le fichier déjà présent avant de renommer le nouveau.

Nous pouvons maintenant rassembler tous ces éléments dans le fichier `requestHandlers` :

```
var querystring = require("querystring"),
    fs = require("fs"),
    formidable = require("formidable");

function start(response) {
  console.log("Le gestionnaire 'start' est appelé.");

  var body = '<html>'+
    '<head>'+
    '<meta http-equiv="Content-Type" '+
    'content="text/html; charset=UTF-8" />'+
    '</head>'+
    '<body>'+
    '<form action="/upload" enctype="multipart/form-data" '+
    'method="post">'+
    '<input type="file" name="upload" multiple="multiple">'+
    '<input type="submit" value="Transférer le fichier" />'+
    '</form>'+
    '</body>'+
    '</html>';

  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(body);
  response.end();
}
```

```
function upload(response, request) {
  console.log("Le gestionnaire 'upload' est appelé.");

  var form = new formidable.IncomingForm();
  console.log("Récupération des éléments reçus");
  form.parse(request, function(error, fields, files) {
    console.log("Traitement terminé");

    /* En cas d'erreur sous Windows :
       tentative d'écrasement d'un fichier existant */
    fs.rename(files.upload.path, "/tmp/test.png", function(err) {
      if (err) {
        fs.unlink("/tmp/test.png");
        fs.rename(files.upload.path, "/tmp/test.png");
      }
    });
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("Image reçue :<br/>");
    response.write("<img src='/show' />");
    response.end();
  });
}

function show(response) {
  console.log("Le gestionnaire 'show' est appelé.");
  fs.readFile("/tmp/test.png", "binary", function(error, file) {
    if(error) {
      response.writeHead(500, {"Content-Type": "text/plain"});
      response.write(error + "\n");
      response.end();
    } else {
      response.writeHead(200, {"Content-Type": "image/png"});
      response.write(file, "binary");
      response.end();
    }
  });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```



Voilà, nous en avons terminé. Redémarrez le serveur pour avoir notre étude de cas complète. Vous pouvez maintenant sélectionner une image (au format PNG) sur votre disque, l'envoyer au serveur et l'afficher dans une page Web.

## Conclusion et remerciements

Félicitations, la mission est réussie ! Nous avons écrit une application Web certes simple mais complète à l'aide de Node.js.


Évidemment, il y a beaucoup de choses que nous n'avons pas abordées : interroger une base de données, écrire des tests unitaires, créer un module externe récupérable avec npm ou même des choses plus simples comme gérer les paramètres GET.


Mais c'est le lot de tous les guides de débutants, ils ne peuvent pas aborder de façon détaillée tous les aspects possibles.

La bonne nouvelle, c'est que la communauté autour de Node.js est extrêmement active (imaginez un enfant hyperactif sous caféine, mais dans le bon sens) ce qui signifie qu'il existe de nombreuses ressources ici ou là et beaucoup d'endroits où poser vos questions et obtenir des réponses. Deux bons points d'entrée pour obtenir des informations sont  **Node.js community wiki** et  **the NodeCloud directory**.



Vous pouvez [Source](#) **télécharger l'archive** du code de l'application complète.

Ce guide est la traduction de  **The Node Beginner Book**. Ne tenons à remercier Manuel Kiessling pour son aimable autorisation.

Vous pouvez soutenir son travail et en apprendre plus sur Node.js en achetant  **le pack Node.js** comprenant la version imprimée de ce guide et *Hands-on Node.js*.

Nous tenons à remercier particulièrement **ClaudeLELOUP** pour sa relecture attentive de cet article.