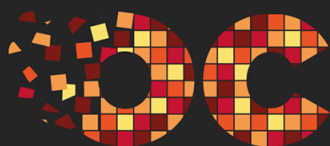


DES APPLICATIONS ULTRA-RAPIDES AVEC

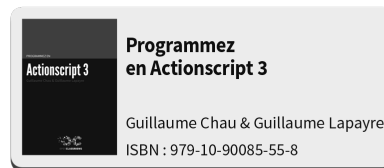
NODE.JS

Mathieu Nebra



OPENCLASSROOMS

DANS LA MÊME COLLECTION



Rejoignez la communauté OpenClassrooms :



www.openclassrooms.com



www.facebook.com/openclassrooms

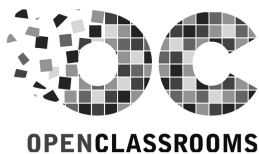


[@OpenClassrooms](https://twitter.com/OpenClassrooms)

DES APPLICATIONS ULTRA-RAPIDES AVEC

NODE.JS

Mathieu Nebra





Sauf mention contraire, le contenu de cet ouvrage est publié sous la licence :
Creative Commons BY-NC-SA 2.0

La copie de cet ouvrage est autorisée sous réserve du respect des conditions de la licence
Texte complet de la licence disponible sur : <http://creativecommons.org/licenses/by-nc-sa/2.0/fr/>

OpenClassrooms 2014 - ISBN : 979-10-90085-61-9

Mentions légales :
Conception couverture : Amalgam Impression
Illustrations chapitres : Alexandra Persil

Avant-propos

JavaScript est décidément un langage étonnant. Lancé par Brendan Eich en 1995 dans le navigateur Netscape pour permettre le développement côté client (c'est-à-dire l'exécution de code sur la machine du visiteur), il est aujourd'hui devenu un langage incontournable du Web.

Jusqu'ici, JavaScript pouvait être utilisé uniquement du côté client... et voilà que depuis peu il est aussi possible de l'utiliser pour développer côté serveur ! PHP, Java, Python, Ruby et les autres n'ont qu'à bien se tenir !

Tout le petit monde du développement web est en ébullition et n'a d'yeux que pour Node.js et les perspectives qu'il ouvre pour le développement en JavaScript côté serveur. Pourquoi ? Parce que la conception de Node.js est très différente de ce qu'on a l'habitude de voir. Tout est basé sur un système d'évènements qui lui permet d'être extrêmement efficace, ce qui lui confère sa réputation de rapidité... et aussi une certaine complexité parfois, il faut bien l'avouer !

Mon but dans ce cours est de vous faire découvrir l'utilisation de Node.js pas à pas avec plusieurs applications pratiques. Étant donné que Node.js est assez complexe à prendre en main, j'ai fait de mon mieux pour que votre apprentissage soit le plus progressif possible et que vous preniez autant de plaisir que moi avec cette nouvelle façon de programmer !

Qu'allez-vous apprendre en lisant ce livre ?

Ce livre a été conçu en partant du principe que vous connaissez déjà le langage JavaScript. En effet, Node.js est directement basé sur JavaScript et il ne m'est pas possible de vous enseigner les deux à la fois¹ ! Cependant, je ferai des rappels régulièrement pour les notions les plus délicates.

Voici les différentes parties qui jalonnent ce cours :

1. **Comprendre Node.js** : cette première partie sera l'occasion de faire des rappels sur JavaScript, son histoire et la naissance de Node.js. Nous installerons Node.js et nous créerons notre première application pas à pas ensemble ;

1. D'autant plus qu'un excellent livre "Dynamisez vos sites web avec JavaScript" existe déjà dans la même collection !

2. **Structurer son application Node.js** : ce sera le moment de rentrer plus en profondeur dans le fonctionnement d'une application Node.js. Nous parlerons d'évènements, de modules et du framework Express.js, puis réaliserons ensemble une todo list ;
3. **La communication temps réel avec socket.io** : socket.io est l'une des bibliothèques les plus célèbres de Node.js car elle permet une communication en temps réel dans les deux sens entre le client et le serveur. A la clé : nous serons capables de réaliser un véritable Chat temps réel en ligne !

Comment lire ce livre ?

Suivez l'ordre des chapitres

Lisez ce cours comme on lit un roman. Il a été conçu pour cela.

Contrairement à beaucoup de livres techniques où il est courant de lire en diagonale et de sauter certains chapitres, il est ici très fortement recommandé de suivre l'ordre du cours, à moins que vous ne soyez déjà un peu expérimentés.

Pratiquez en même temps

Pratiquez régulièrement. N'attendez pas d'avoir fini de lire ce livre pour allumer votre ordinateur et faire vos propres essais.

Utilisez les codes web !

Afin de tirer parti d'OpenClassrooms dont ce livre est issu, celui-ci vous propose ce qu'on appelle des « codes web ». Ce sont des codes à six chiffres à saisir sur une page d'OpenClassrooms pour être automatiquement redirigé vers un site web sans avoir à en recopier l'adresse.

Pour utiliser les codes web, rendez-vous sur la page suivante :

<http://www.openclassrooms.com/codeweb>

Un formulaire vous invite à rentrer votre code web. Faites un premier essai avec le code ci-dessous :

▷

▶

Tester le code web

Code web : 123456

Ces codes web ont deux intérêts :

- ils vous redirigent vers les sites web présentés tout au long du cours, vous permettant ainsi d'obtenir les logiciels dans leur toute dernière version ;
- ils vous permettent d'afficher les codes sources inclus dans ce livre, ce qui vous évitera d'avoir à recopier certains programmes un peu longs.

Ce système de redirection nous permet de tenir à jour le livre que vous avez entre les mains sans que vous ayez besoin d'acheter systématiquement chaque nouvelle édition. Si un site web change d'adresse, nous modifierons la redirection mais le code web à utiliser restera le même. Si un site web disparaît, nous vous redirigerons vers une page d'OpenClassrooms expliquant ce qui s'est passé et vous proposant une alternative.

En clair, c'est un moyen de nous assurer de la pérennité de cet ouvrage sans que vous ayez à faire quoi que ce soit !

Table des matières

Avant-propos	i
Comment lire ce livre ?	ii
Suivez l'ordre des chapitres	ii
Pratiquez en même temps	ii
Utilisez les codes web !	ii
 I Comprendre Node.js	 1
1 Node.js : mais à quoi ça sert ?	3
Du JavaScript « à la papa » à Node.js	4
Node.js : le JavaScript côté serveur	5
Pourquoi Node.js est-il rapide ?	7
Le moteur V8	8
Le modèle non bloquant	8
 2 Installer Node.js	 13
Installation de Node.js sous Windows	14
Installation de Node.js sous Mac OS X	17
Installation de Node.js sous Linux	19
Tester Node.js avec un programme minimal	20
 3 Une première application avec Node.js	 23
Des serveurs web et des threads	24

Construire son serveur HTTP	25
Disséquons du code	26
Tester le serveur HTTP	27
Retourner du code HTML	28
Déterminer la page appelée et les paramètres	31
Quelle est la page demandée par le visiteur ?	31
Quels sont les paramètres ?	33
Schéma résumé	34
 II Structurer son application Node.js	 35
 4 Les évènements	 37
Écouter des évènements	38
Émettre des évènements	40
 5 Les modules Node.js et NPM	 43
Créer des modules	44
Utiliser NPM pour installer des modules	46
Trouver un module	47
Installer un module	48
L'installation locale et l'installation globale	49
Mettre à jour les modules	50
Déclarer et publier son module	50
Le fonctionnement des numéros de version	51
La gestion des versions des dépendances	52
Publier un module	52
 6 Le framework Express.js	 55
Les routes	56
Routes simples	56
Routes dynamiques	58
Les templates	59
Les bases d'EJS	60
Plusieurs paramètres et des boucles	60
Aller plus loin : Connect et les middlewares	61

Express, connect et les middlewares	62
Utiliser les fonctionnalités de Connect au sein d'Express	64
7 TP : la todo list	67
Besoin d'aide ?	68
Les modules et le package.json	68
Les routes	69
Bien chaîner les appels aux middlewares	70
Correction	71
La solution	71
Les explications	73
Télécharger le projet	74
Allez plus loin !	74
 III La communication temps réel avec socket.io	 75
8 socket.io : passez au temps réel !	77
Que fait socket.io ?	78
Émettre et recevoir des messages avec socket.io	80
Installer socket.io	80
Premier code : un client se connecte	80
Envoi et réception de messages	83
Communiquer avec plusieurs clients	87
Envoyer un message à tous les clients (broadcast)	87
Les variables de session	88
 9 TP : le super Chat	 95
Besoin d'aide ?	96
package.json	97
app.js	97
index.html	98
Correction	98
package.json	99
app.js	99
index.html	100

Télécharger le projet	103
Allez plus loin !	103

Première partie

Comprendre Node.js

Chapitre 1

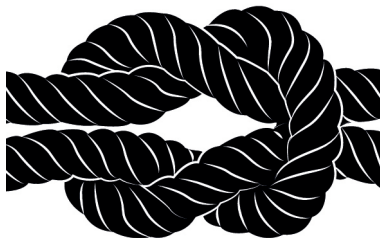
Node.js : mais à quoi ça sert ?

Difficulté : 

Et si on commençait par répondre à toutes ces questions qui vous taraudent :

- Pourquoi **Node.js** semble-t-il aussi apprécié des développeurs web ?
- Pourquoi avoir utilisé un langage comme **JavaScript** ? Je croyais que c'était juste pour faire des effets dans sa page web ?
- D'où vient cette rapidité supposée de **Node.js** ? À quoi ça peut me servir ?
- Est-ce que ce truc est mature ? Qui l'utilise (à part des geeks barbus !) ? Devrais-je l'utiliser moi aussi ?

Je vais répondre à toute vos interrogations, et même aux questions que vous ne vous posez pas, dans ce premier chapitre. Et si l'aventure vous tente, vous pourrez passer à la suite avec moi et installer **Node.js** sur votre machine !



Du JavaScript « à la papa » à Node.js

OK, l'expression « JavaScript à la papa » est un peu forte, mais il y a du vrai là-dedans : **JavaScript** a eu plusieurs vies. Il a eu mauvaise presse. On l'a longtemps considéré comme un « petit truc parfois bien pratique pour faire des effets sur sa page web ». Or, **JavaScript** est avant tout un *langage* au même titre que **C**, **Ruby**, **PHP** et bien d'autres.

Désormais, **JavaScript** est de retour et il prend sa revanche. Les développeurs sont en train de découvrir que ce langage qu'ils ont longtemps ignoré, parfois même méprisé, cache en fait bien son jeu.

Non, **JavaScript** n'est pas juste un petit langage utilitaire. Oui, **JavaScript** est un langage à part, qui s'utilise vraiment différemment de **Java**, du **C** et d'un tas d'autres langages. Oui, **JavaScript** peut être compliqué à utiliser, mais recèle une véritable puissance.

Alors que ce langage a été créé en 1995 (la préhistoire de l'informatique, rendez-vous compte!), il a depuis bien évolué. Je vous invite à (re)lire la petite histoire de Javascript, racontée par les compères Sébastien de la Marck et Johann Pardanaud dans leur cours intitulé "Dynamisez vos sites web avec Javascript!" que vous retrouverez sur le site d'Openclassrooms.

Je vous disais que **JavaScript** était un langage qui avait connu plusieurs vies. Pour être plus précis, je dirais même qu'il a connu trois vies (comme l'illustre la figure 1.1) :

1. Dans les années 90, on parlait de **DHTML** (Dynamic HTML). On utilisait en fait les toutes premières versions de **JavaScript** pour créer des petits effets dans ses pages web : afficher une image lors d'un clic sur un bouton par exemple. C'était l'époque de Netscape et d'Internet Explorer 5.5.
2. Dans les années 2000, on a commencé à utiliser le langage pour créer des interfaces côté client. C'est là que des bibliothèques comme **jQuery** ou **Mootools** sont apparues. Aujourd'hui, cet usage de **JavaScript** est très répandu et mature. On a pris l'habitude de manipuler le **DOM** (Document Object Model) pour affecter ses **balises HTML** en **JavaScript** et leur faire subir toutes sortes de traitements.
3. Puis, aux alentours de 2010, **JavaScript** est entré dans une nouvelle ère. Google a commencé à rendre le langage beaucoup plus rapide avec l'apparition du navigateur Google Chrome. Avec ce navigateur est né le moteur d'exécution V8 qui a considérablement permis d'accélérer l'exécution de code **JavaScript** (j'y reviendrai). Des outils comme **Node.js** sont ensuite apparus. Les bibliothèques dont le nom finit par .js se sont multipliées : **Backbone.js**, **Ember.js**, **Meteor.js**. **JavaScript** a l'air à nouveau « cool » et semble en même temps plus compliqué qu'il n'y paraît au premier abord.

Soyons clairs : **jQuery** n'est pas mort et ça ne veut pas dire qu'il faut cesser de l'utiliser (par contre **DHTML** et **Netscape** sont bel et bien morts eux). Les nouveaux outils **JavaScript** comme **Node.js** font des choses très différentes de **jQuery** et consorts. Les deux peuvent tout à fait se compléter.

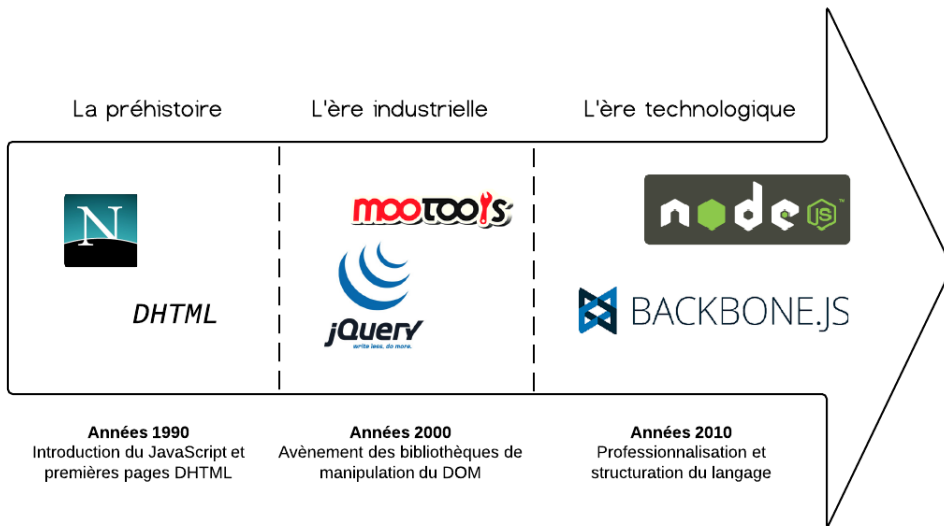


FIGURE 1.1 – Les trois vies de JavaScript



Mais ça apporte quoi concrètement **Node.js** ?

Node.js nous permet d'utiliser le langage **JavaScript** sur le serveur... Il nous permet donc de faire du **JavaScript** *en dehors du navigateur* ! **Node.js** bénéficie de la puissance de **JavaScript** pour proposer une toute nouvelle façon de développer des sites web dynamiques.

Je vous propose justement de rentrer maintenant dans le détail du fonctionnement de **Node.js**.

Node.js : le JavaScript côté serveur

Jusqu'ici, **JavaScript** avait toujours été utilisé du côté du client, c'est-à-dire du côté du visiteur qui navigue sur notre site. Le navigateur web du visiteur (Firefox, Chrome, IE...) exécute le code **JavaScript** et effectue des actions sur la page web. (Voir figure 1.2)



Qu'est-ce qui change avec l'arrivée de **Node.js** ?

On peut toujours utiliser du **JavaScript** côté client pour manipuler la **page HTML**.

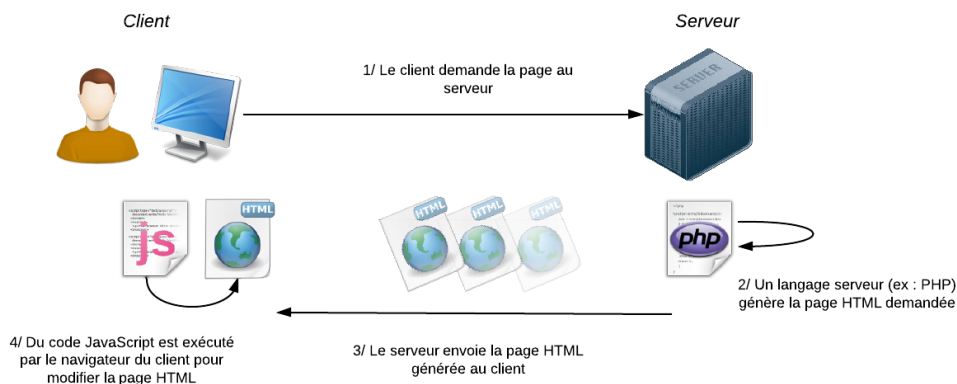


FIGURE 1.2 – Le schéma classique : PHP sur le serveur, JavaScript chez le client

Ça, ça ne change pas.

Par contre, **Node.js** offre un environnement *côté serveur* qui nous permet aussi d'utiliser le langage **JavaScript** pour générer des pages web. En gros, il vient en remplacement de langages serveur comme **PHP**, **Java EE**, etc. (Voir la figure 1.3)

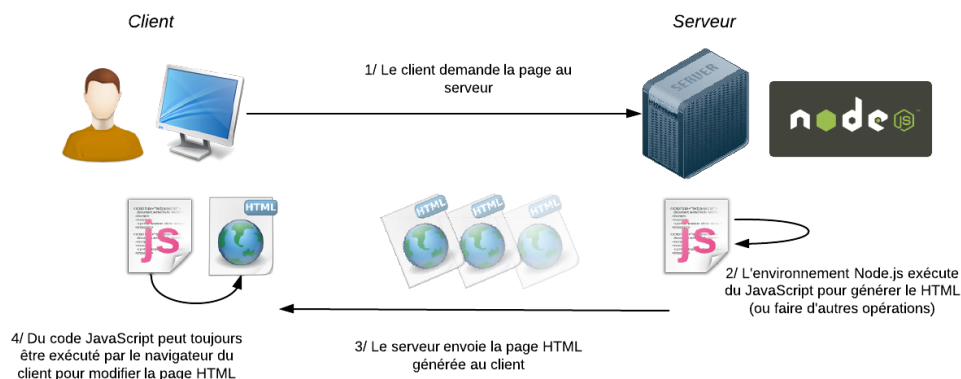


FIGURE 1.3 – Avec Node.js, on peut aussi utiliser du JavaScript sur le serveur !



Pourquoi **Node.js** semble-t-il si différent ? Parce qu'il utilise **JavaScript** ?

Oui, et parce que **JavaScript** est un langage basé sur les **événements**, donc **Node.js** est lui-même basé sur les **événements**. Du coup, c'est toute la façon d'écrire des applications web qui change ! C'est de là que **Node.js** tire toute sa puissance et sa rapidité.

Avec **Node.js**, vous pouvez créer des applications rapides comme :

- Un serveur de Chat.
- Un système d'upload très rapide.
- Et de façon générale, n'importe quelle application qui doit répondre à de nombreuses requêtes rapidement et efficacement, en temps réel.



Node.js n'est pas un **framework**. **Node.js** est un environnement *très bas niveau*. Il se rapproche donc en quelque sorte plus du **C** que de **PHP**, **Ruby on Rails** ou **Django**. Voilà pourquoi il n'est pas vraiment conseillé aux débutants. Notez cependant qu'il existe des frameworks web comme **Express** qui sont basés sur **Node.js**. Ils nous permettent d'éviter les tâches répétitives qui nous sont imposées par la nature bas niveau de **Node.js**, mais ils restent quand même plus complexes à utiliser que des langages comme **PHP**.

JavaScript n'est pas vraiment un langage Orienté Objet, il est donc très loin de **Java**, **Ruby** ou **Python**. Écrire une application avec **Node.js** demande une gymnastique d'esprit complètement différente ! C'est un peu déroutant au début pour tout vous avouer, mais quand on commence à maîtriser cet outil, on se sent un peu comme si on venait d'avoir de nouveaux super-pouvoirs qu'on ne soupçonnait pas. Voilà pourquoi tant de geeks barbus sont excités à propos de **Node.js** ! (Voir figure 1.4)



FIGURE 1.4 – Avant, j'étais un barbu. . .

Pourquoi Node.js est-il rapide ?

Si **Node.js** est rapide, cela tient principalement à deux choses : **le moteur V8** et son **fonctionnement non bloquant**.

Le moteur V8

Node.js utilise le **moteur d'exécution ultrarapide V8** de Google Chrome. Ce **moteur V8** avait beaucoup fait parler de lui à la sortie de Google Chrome, car c'est **un outil open source créé par Google qui analyse et exécute du code JavaScript très rapidement**. (Voir figure 1.5)



FIGURE 1.5 – Le logo du moteur JavaScript V8 de Google

Jusqu'à la sortie de Chrome, la plupart des navigateurs lisaient le **code JavaScript** de façon peu efficace : le code était lu et interprété au fur et à mesure. Le navigateur mettait beaucoup de temps à lire le **JavaScript** et à le transformer en code machine compréhensible pour le processeur.

Le **moteur V8** de Google Chrome, qui est réutilisé ici par **Node.js**, fonctionne complètement différemment. Très optimisé, il fait ce qu'on appelle de la **compilation JIT (Just In Time)**. Il transforme le **code JavaScript** très rapidement en code machine et l'optimise même grâce à des procédés complexes : **code inlining, copy elision** et j'en passe.

Vous n'avez pas besoin de connaître le fonctionnement de **V8** pour utiliser **Node.js**. Retenez juste qu'il permet de rendre l'exécution de **code JavaScript** ultrarapide (et que les développeurs chez Google qui l'ont conçu sont de bons vrais barbus :-°).

Le modèle non bloquant

Comme **JavaScript** est un langage conçu autour de la notion d'évènement, **Node.js** a pu mettre en place une architecture de code entièrement *non bloquante*. Mais au fait, connaissez-vous la différence entre un **code bloquant** et un **code non bloquant** ? Hmmm, un peu d'explication ne peut pas faire de mal !

Modèle bloquant vs modèle non bloquant

Imaginez un programme dont le rôle est de télécharger un fichier puis de l'afficher. Voici comment on écrirait le code dans un **modèle bloquant** :

```
1 | Télécharger un fichier
2 | Afficher le fichier
3 | Faire autre chose
```

Les actions sont effectuées dans l'ordre. Il faut lire les lignes de haut en bas :

1. Le programme va télécharger un fichier sur Internet.
2. Le programme affiche le fichier à l'utilisateur.

3. Ensuite, le programme peut faire d'autres choses (effectuer d'autres actions).

Maintenant, on peut écrire le même code sur un **modèle non bloquant** :

```
1 | Télécharger un fichier
2 |   Dès que c'est terminé, afficher le fichier
3 | Faire autre chose
```

Le programme n'exécute plus les lignes dans l'ordre où elles sont écrites. Il fait ceci :

1. Le programme lance le téléchargement d'un fichier sur Internet.
2. Le programme fait d'autres choses (le programme suit son cours).
3. Dès que le téléchargement est terminé, le programme effectue les actions qu'on lui avait demandées : il affiche le fichier.

Schématiquement, l'exécution du programme peut donc se représenter comme la figure 1.6.

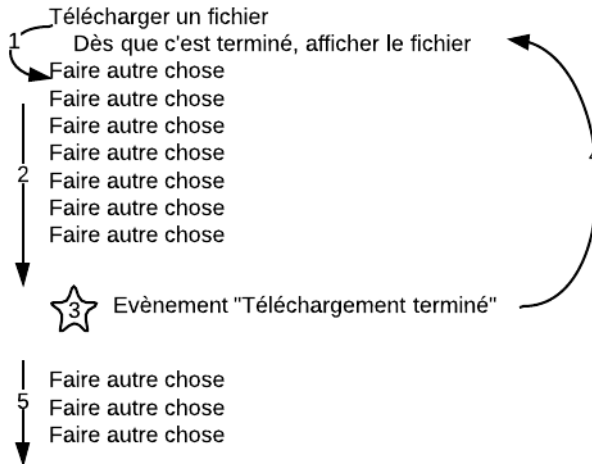


FIGURE 1.6 – Le modèle non bloquant en programmation

C'est justement comme ça que fonctionne **Node.js**. Dès que l'évènement « Fichier téléchargé » apparaît, une fonction appelée **fonction de callback** est appelée et effectue des actions (ici, la fonction de callback affiche le fichier).

Le modèle non bloquant avec Node.js

Et avec du vrai code ça donne quoi ? Voici un exemple de programme **Node.js** qui télécharge un fichier sur Internet et affiche « Fichier téléchargé ! » quand il a terminé :

```
1 | request('http://www.site.com/fichier.zip', function (error,
   |   response, body) {
2 |   console.log("Fichier téléchargé !");
```

```
3 | });  
4 | console.log("Je fais d'autres choses en attendant...");
```

La requête de téléchargement est lancée en premier. Ensuite, le programme fait d'autres choses (ici, il affiche un message dans la console, mais il pourrait faire n'importe quoi d'autre). Dès que le téléchargement est terminé, le programme va à la ligne 2 et affiche « Fichier téléchargé ! ».



Mais comment ça marche ? Je vois une fonction en paramètre de la fonction `request` ! C'est une fonction dans une fonction, au secours ces barbus sont fous !

Pas de panique ! Ce que vous voyez là est une **fonction de callback**. En **JavaScript**, on peut tout à fait envoyer une fonction en paramètre d'une autre fonction. Cela signifie ici : « Exécute cette fonction quand le téléchargement est terminé. »

Ici, la fonction n'a pas de nom. On dit que c'est une **fonction anonyme**. Mais on pourrait décomposer ce code comme ceci, le résultat serait *identique* :

```
1 | // Résultat identique au code précédent  
2 |  
3 | var callback = function (error, response, body) {  
4 |     console.log("Fichier téléchargé !");  
5 | };  
6 |  
7 | request('http://www.site.com/fichier.zip', callback);  
8 | console.log("Je fais d'autres choses en attendant...");
```

La **fonction de callback** est enregistrée dans une variable. Comme toutes les fonctions, elle n'est pas exécutée tant qu'on ne l'a pas appelée. Ensuite, on envoie cette **fonction de callback** en paramètre de la fonction `request()` pour dire : « Dès que la requête de téléchargement est terminée, appelle cette fonction de callback. »

En pratique, les développeurs **JavaScript** mettent régulièrement des **fonctions anonymes** directement à l'intérieur d'autres fonctions en paramètre, comme dans mon premier code. C'est un peu étrange au début, mais on s'y fait vite !



Je ne vois pas pourquoi ça rendrait le programme plus rapide. J'ai l'impression que ça le rend surtout plus compliqué !

Je vous avais dit que **Node.js** n'était pas simple, mais le jeu en vaut la chandelle ! Vous allez comprendre pourquoi. Imaginez qu'on demande le téléchargement de deux fichiers à **Node.js** :

```
1 | var callback = function (error, response, body) {  
2 |     console.log("Fichier téléchargé !");  
3 | };  
4 |  
5 | request('http://www.site.com/fichier.zip', callback);
```

```
6 | request('http://www.site.com/autrefichier.zip', callback);
```

Si le modèle avait été *bloquant*, le programme aurait :

1. Lancé le téléchargement du fichier 1, et attendu qu'il se termine.
2. Puis lancé le téléchargement du fichier 2, et attendu qu'il se termine.

Or, avec **Node.js**, les deux téléchargements sont lancés *en même temps* ! Le programme n'attend pas la fin du premier téléchargement pour passer à l'instruction suivante.

Du coup, le téléchargement des deux fichiers va beaucoup plus vite puisque le programme fait les deux à la fois (voir figure 1.7) :

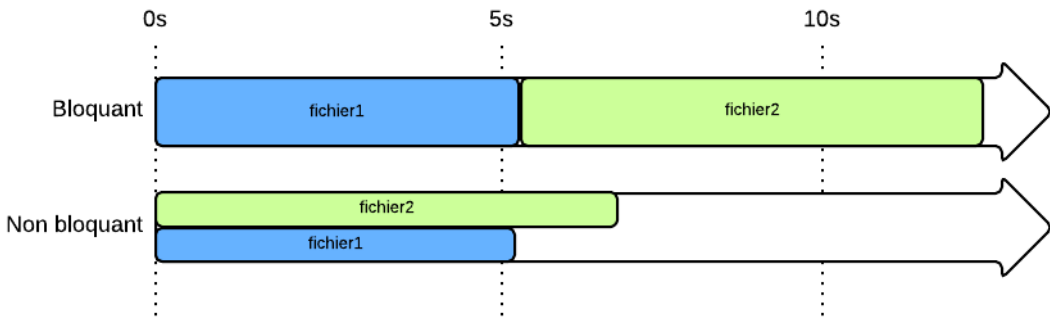


FIGURE 1.7 – En modèle non bloquant (comme Node.js), les 2 fichiers sont téléchargés en même temps et l'ensemble finit plus vite

Dans les applications web, il est courant d'avoir des opérations longues et bloquantes comme :

- Les appels aux bases de données.
- Les appels à des services web (ex : l'API de Twitter).

Node.js nous évite de perdre du temps en nous permettant de faire d'autres choses en attendant que les actions longues soient terminées !

En résumé

- **Node.js** est un *environnement de développement* qui permet de coder côté serveur en **JavaScript**. En ce sens il peut être comparé à **PHP**, **Python/Django**, **Ruby on Rails**, etc.
- Écrire une application en **Node.js** demande une gymnastique d'esprit particulière : tout est basé sur des **événements** !
- **Node.js** est reconnu et apprécié pour sa *rapidité* : un programme **Node.js** n'attend jamais inutilement sans rien faire !

Chapitre 2

Installer Node.js

Difficulté : 

Vous êtes convaincus ? Vous voulez vous mettre à **Node.js** ? Très bien ! N'attendons pas et installons la bête ! Dans ce chapitre, nous couvrirons l'installation pour Windows, Mac OS X et Linux. Vous devez juste lire la section qui correspond à votre système d'exploitation. Il n'y aura rien de bien difficile ici rassurez-vous.

Ensuite, nous testerons une toute première application très simple pour vérifier que **Node.js** est bien installé et fonctionne correctement. Je vous conseille de lire cette section si vous voulez être prêts pour le prochain chapitre.



Installation de Node.js sous Windows

Pour installer **Node.js** sous Windows, il suffit de télécharger l'installateur qui est proposé sur le code web suivant :

▷ Site de Node.js
Code web : 435670.

Cliquez simplement sur le lien **Install**. Vous pouvez aussi vous rendre sur la page des téléchargements pour avoir plus d'options (voir figure 2.7).

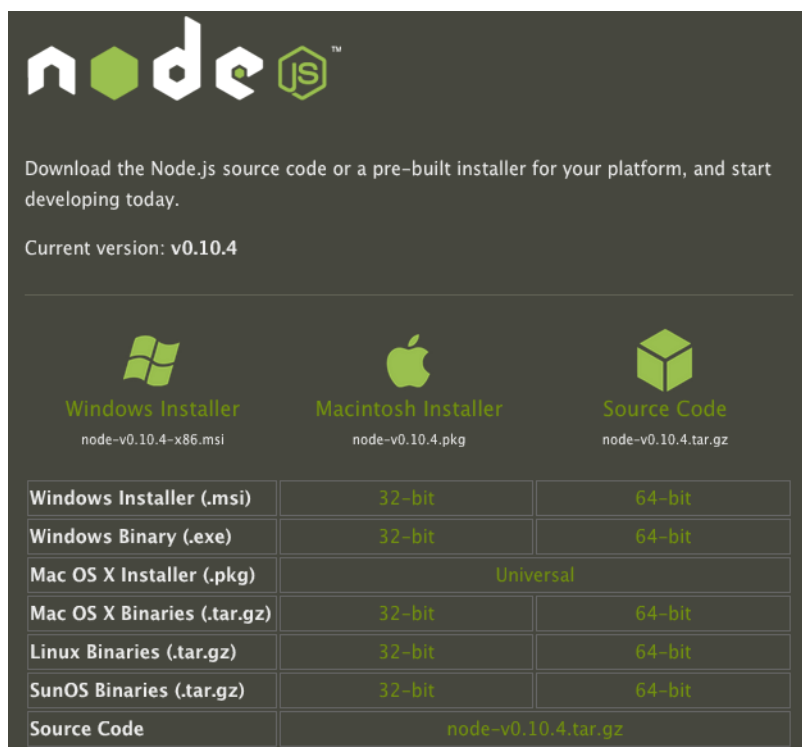


FIGURE 2.1 – La page de téléchargement de Node.js

Vous pouvez télécharger soit le .msi, soit le .exe (le résultat sera le même). Prenez la version 64 bits si vous avez un Windows 64 bits (cas de la plupart des PC récents). Dans le doute, prenez la version 32 bits.

Lancez ensuite l'installateur (voir figure 2.2).

Après quelques écrans classiques, on vous demandera ce que vous voulez installer. Je vous invite à tout laisser coché (voir figure 2.3).

L'installation se lance ensuite. Elle ne prend que quelques secondes ! À la fin, on vous dit que **Node.js** est installé (voir figure 2.4).

Mais où ? Comment ça marche ? En fait, vous devriez avoir deux programmes installés :

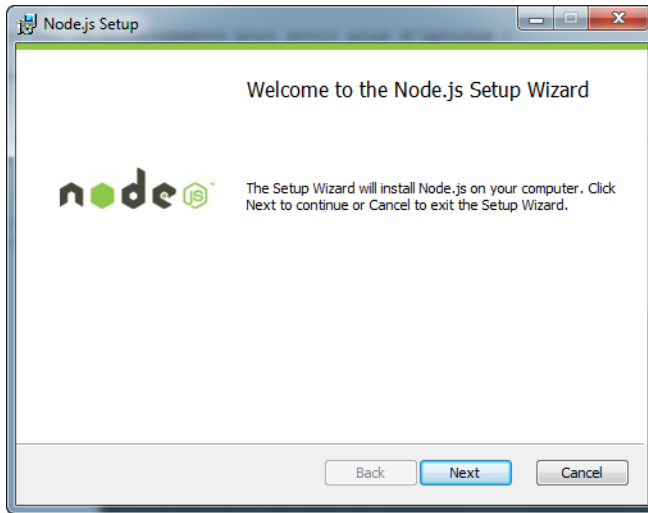


FIGURE 2.2 – Installation de Node.js

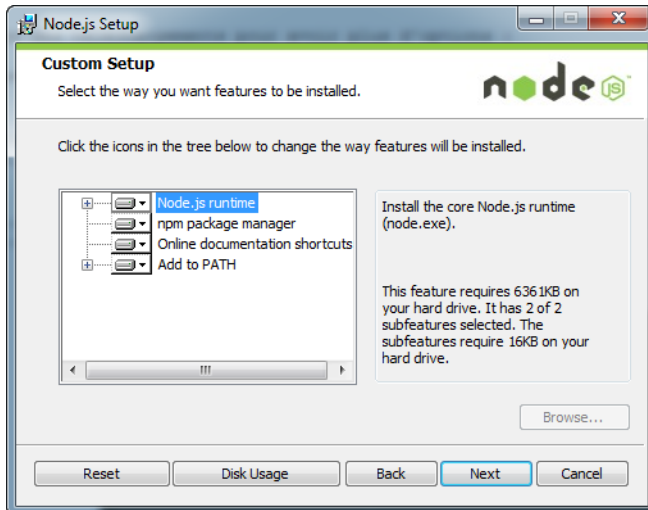


FIGURE 2.3 – Choix des éléments à installer

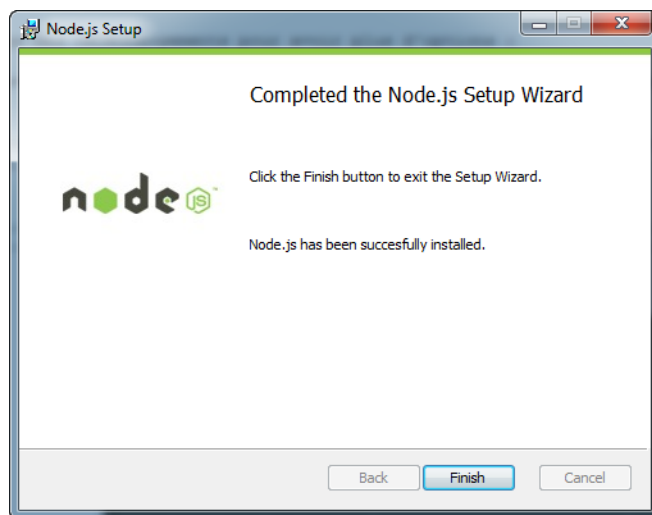


FIGURE 2.4 – L’installation est terminée !

- **Node.js** : c’est l’**interpréteur de commandes de Node.js** (dont nous parlerons à la fin de ce chapitre). Nous l’utiliserons assez peu en pratique. Il sert à tester des commandes **JavaScript**.
- **Node.js command prompt** : c’est une console de Windows configurée pour reconnaître **Node.js**. C’est par là que vous passerez pour lancer vos programmes **Node.js**, c’est donc ce que nous utiliserons le plus souvent. (Voir les figures 2.5 et 2.6)



FIGURE 2.5 – Node.js - L’interpréteur Node.js sous Windows (peu utilisé)

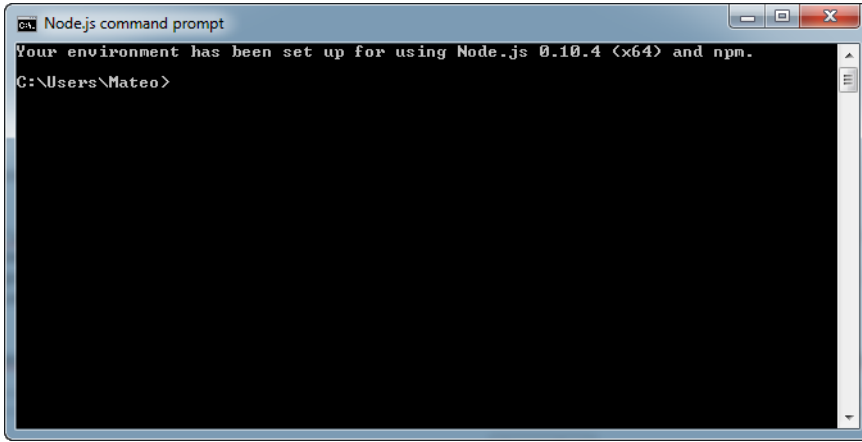


FIGURE 2.6 – Node.js command prompt - La console Node.js (fréquemment utilisée)

Installation de Node.js sous Mac OS X

Si vous êtes sous Mac OS X, vous pouvez cliquer sur le lien **Install** grâce au code web suivant (voir figure 2.7) :

- ▷ Page d'accueil de Node.js
Code web : 435670

Si vous voulez plus d'options, affichez la page de téléchargement grâce au code web suivant :

- ▷ Page des téléchargements
Code web : 914330

Le mieux est de prendre l'installateur (fichier .pkg). Il ouvre un assistant d'installation (voir figure 2.8) dans lequel il suffit de cliquer frénétiquement sur **Continuer**, **Continuer**, **Continuer**, **Terminer**.

Une fois l'installation terminée, vous pouvez vérifier que **Node.js** fonctionne correctement en tapant la commande **node** dans la console. Ouvrez une fenêtre de Terminal (le Terminal étant installé par défaut sous Mac OS X), en allant dans le Finder, section « Applications », « Terminal ». Je vous conseille de mettre un raccourci dans le dock !

Tapez quelques commandes comme **node -v** (pour avoir le numéro de version) ou **node** tout court pour lancer l'interpréteur interactif (voir figure 2.9).

Lorsque l'interpréteur est lancé, vous pouvez taper des commandes JavaScript et obtenir une réponse. Ici, par exemple, j'ai demandé combien font 1 et 1.

Pour quitter l'interpréteur, faites **Ctrl** + **D** (c'est la commande classique qui demande à quitter un interpréteur sous Linux et Mac).



FIGURE 2.7 – La page de téléchargement de Node.js

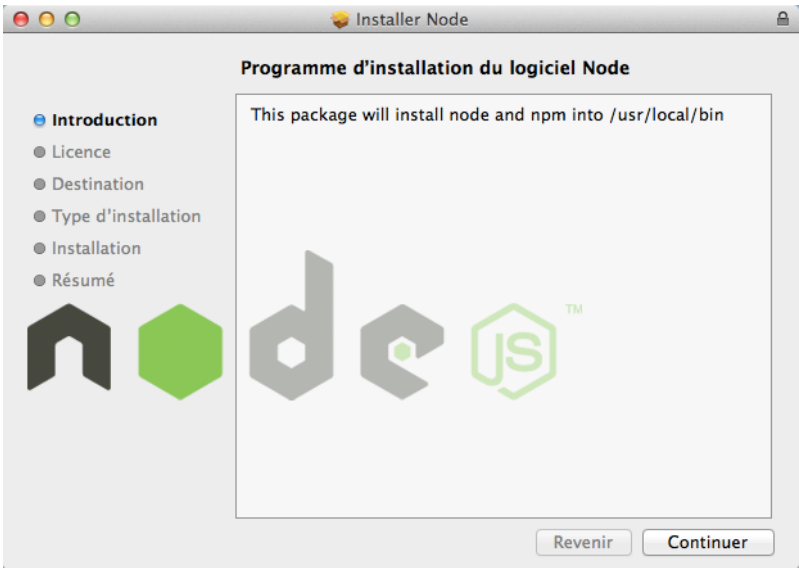
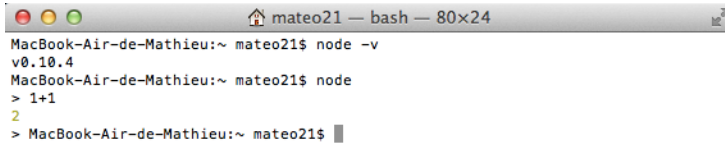


FIGURE 2.8 – L'installation de Node.js sous Mac OS X

A terminal window titled 'mateo21 — bash — 80x24' on a MacBook-Air-de-Mathieu. The prompt is 'mateo21\$'. The user enters 'node -v', which outputs 'v0.10.4'. The user enters 'node', which outputs '1+1' and then '2'. The prompt returns to 'mateo21\$'.

```
MacBook-Air-de-Mathieu:~ mateo21$ node -v
v0.10.4
MacBook-Air-de-Mathieu:~ mateo21$ node
> 1+1
2
> MacBook-Air-de-Mathieu:~ mateo21$
```

FIGURE 2.9 – Exécution de Node.js dans le Terminal



Rassurez-vous, nous n'écrirons pas nos programmes **Node.js** là-dedans ! Cela nous sert ici surtout pour faire des tests. En pratique, nous allons écrire des fichiers .js et demander à Node de les exécuter. Nous verrons ça un peu plus loin.

Installation de Node.js sous Linux

Sous Linux, comme d'habitude, vous avez deux choix :

1. La méthode « warrior », qui consiste à télécharger les sources et à les compiler.
2. La méthode douce, qui consiste à utiliser le gestionnaire de paquets de sa distribution.

Là, c'est une question de goût. Je n'ai rien contre la méthode « warrior », mais comme j'ai en général moins de problèmes de configuration avec la méthode douce, je préfère passer par un gestionnaire de paquets.

Sous Ubuntu par exemple, vous devrez rentrer les commandes suivantes :

```
sudo apt-get install python-software-properties python g++ make
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
sudo apt-get install nodejs
```

Pour avoir la commande `add-apt-repository` il vous faudra peut-être télécharger `software-properties-common` : `sudo apt-get install software-properties-common`.

Et voilà le travail ! Pour vérifier que Node est bien installé, tapez quelques commandes dans la console comme :

```
node -v
node
```

La première affiche le numéro de version de **Node.js** que vous avez installé. La seconde lance l'interpréteur interactif de **Node.js**. Vous pouvez y taper du **code JavaScript** (essayez simplement de taper « 1+1 » pour voir). Pour sortir de l'interpréteur, faites **Ctrl + D**.

Rassurez-vous, nous n'écrirons pas nos programmes dans l'interpréteur interactif. Nous créerons plutôt des fichiers `.js` et demanderons à Node de les exécuter.



Depuis les versions les plus récentes de **Node.js**, il faut savoir que **NPM** est installé en même temps automatiquement. **NPM** est le gestionnaire de paquets de **Node.js** (c'est un peu l'équivalent de **apt**, mais pour les extensions **Node.js**). **NPM** est vraiment un outil formidable qui nous permet d'étendre les possibilités de **Node.js** à l'infini, nous le découvrirons un peu plus tard.

Tester Node.js avec un programme minimal

Il est temps de vérifier que **Node.js** fonctionne bien ! Pour commencer, nous allons écrire un tout petit programme qui se contente d'afficher un message dans la console. Ce sera l'occasion de voir comment fonctionne l'exécution de fichiers `.js` avec Node.

Pour commencer, ouvrez votre éditeur de texte favori (vim, Emacs, Sublime Text, Notepad++...) et rentrez le **code JavaScript** suivant :

```
1 | console.log('Bienvenue dans Node.js !');
```

Enregistrez votre fichier sous l'extension `.js`. Par exemple `test.js`. Ensuite, ouvrez une console dans le dossier où se trouve votre fichier `test.js` et entrez la commande `node test.js`. Vous devriez avoir le résultat suivant dans la console :

```
$ node test.js
Bienvenue dans Node.js !
```

Bravo, vous avez créé votre tout premier programme **Node.js** ! C'était vraiment ce que l'on pouvait faire de plus simple. On a simplement demandé à écrire un message dans la console.

Vous avez vu que pour lancer un programme **Node.js**, il suffisait d'indiquer le nom du fichier `.js` à exécuter. Vous savez tout ce qu'il faut savoir pour le moment !

Dans le prochain chapitre, nous attaquerons les choses sérieuses : nous allons créer notre première vraie **application Node.js**. Attention, ça va se corser !

En résumé

- Que ce soit sous Windows, Mac ou Linux, l'installation de **Node.js** est simple.
- Une application **Node.js** s'écrit dans un fichier **JavaScript** à l'extension `.js`.
- Une application **Node.js** se lance avec la commande `node nomdufichier.js`.

Chapitre 3

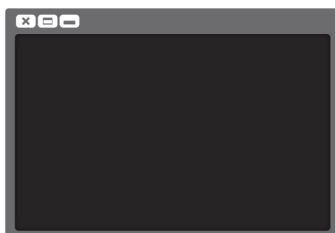
Une première application avec Node.js

Difficulté : 

Les choses sérieuses commencent ! Fini de rire, nous entrons maintenant dans le vif du sujet. Ceci est l'un des chapitres les plus importants du cours car il introduit de nombreux concepts de **Node.js** qui seront, pour la plupart, nouveaux pour vous. Il faudra donc lire ce chapitre dans un environnement calme, progressivement, et ne pas hésiter à le lire une seconde fois le lendemain pour vous assurer que vous avez bien compris.

Dans ce chapitre, nous allons créer une véritable **application Node.js** de bout en bout. Vous allez voir ce que « bas niveau » veut dire ! **Nous allons en effet devoir gérer tous les rouages du serveur web qui va traiter les requêtes HTTP des visiteurs et leur retourner une page web HTML.**

Ce sera pour vous l'occasion d'expérimenter les fameux **callbacks** dont je vous avais parlé dans le premier chapitre, ces fonctions qui s'exécutent dès lors qu'un évènement survient. **Node.js** en est rempli, vous ne pourrez pas y échapper !



Des serveurs web et des threads

Je crois vous l'avoir déjà dit plusieurs fois, mais j'ai envie de le répéter une nouvelle fois ici : **Node.js** est *bas niveau*. Tellement bas niveau que vous allez devoir faire des choses que vous n'avez pas l'habitude de faire pour que votre programme fonctionne correctement.

Quand vous créez des sites web avec **PHP** par exemple, vous associez le langage avec un **serveur web HTTP** comme **Apache** ou **Nginx**. Chacun se répartit les rôles :

- **Apache** gère les demandes de connexion HTTP au serveur. C'est lui qui fait en quelque sorte la circulation et qui gère les entrées/sorties.
- **PHP** exécute le code des fichiers `.php` et renvoie le résultat à **Apache**, qui se charge à son tour de l'envoyer au visiteur.

Comme plusieurs visiteurs peuvent demander une page en même temps au serveur, **Apache** se charge de les répartir et de les exécuter en parallèle dans des **threads** différents. Chaque **thread** utilise un processeur différent sur le serveur (ou un noyau de processeur) (voir figure 3.1) :

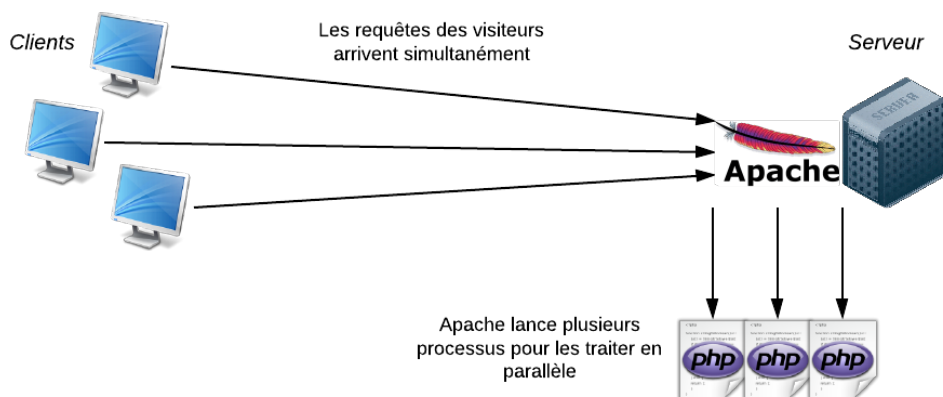


FIGURE 3.1 – Le serveur Apache est multithread

Avec **Node.js**, on n'utilise pas de serveur web HTTP comme **Apache**. En fait, *c'est à nous de créer le serveur* ! Génial non ?

Node.js est **monothread**, contrairement à **Apache**. Cela veut dire qu'il n'y a qu'un seul processus, une seule version du programme qui peut tourner à la fois en mémoire.



Mais je croyais que **Node.js** était très rapide parce qu'il pouvait gérer des tonnes de requêtes simultanées. S'il est **monothread**, il ne peut faire qu'une action à la fois non ?

En effet, il ne peut faire qu'une chose à la fois et ne tourne donc que sur un noyau de processeur. Mais il fait ça de façon ultra efficace, et malgré cela, il est quand même

beaucoup plus rapide ! **Cela est dû à la nature orientée évènements de Node.js.** Les applications utilisant Node ne restent jamais les bras croisés sans rien faire. Dès qu'il y a une action un peu longue, le programme donne de nouveau la main à **Node.js** qui va effectuer d'autres actions en attendant qu'un évènement survienne pour dire que l'opération est terminée. (Voir figure 3.2)

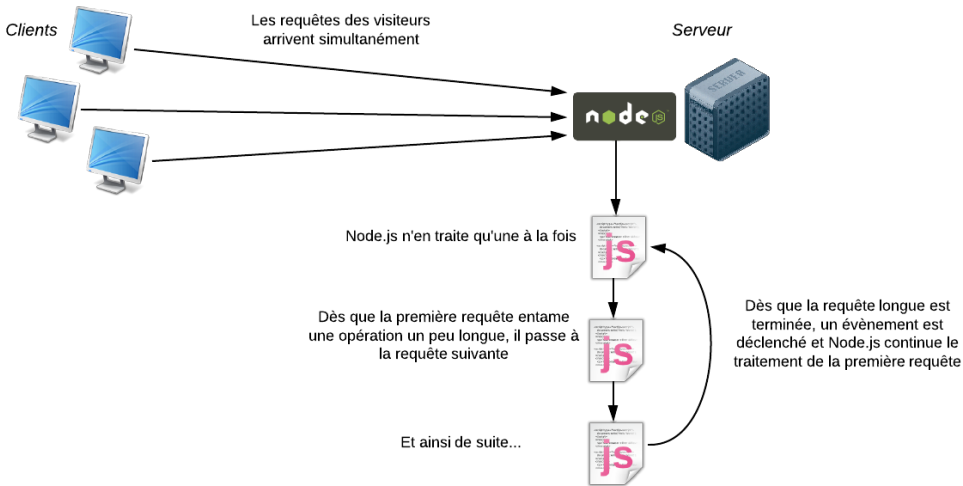


FIGURE 3.2 – Node.js est monothread, mais il est souple grâce aux évènements

Construire son serveur HTTP

Je vous propose d'entrer dans le vif du sujet avec ce tout premier code **Node.js** :

```
1 | var http = require('http');
2 |
3 | var server = http.createServer(function(req, res) {
4 |   res.writeHead(200);
5 |   res.end('Salut tout le monde !');
6 | });
7 | server.listen(8080);
```

C'est en quelque sorte le « code minimal » pour un projet **Node.js**. Placez-le dans un fichier que vous appellerez `serveur.js` (par exemple).



Que fait ce code ?

Il crée un mini-serveur web qui renvoie un message « Salut tout le monde » dans tous les cas, quelque soit la page demandée. Ce serveur est lancé sur le port 8080 à la dernière ligne.

Disséquons du code

Décomposons le code :

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

`require` effectue un appel à une bibliothèque de **Node.js**, ici la bibliothèque « `http` » qui nous permet de créer un serveur web. Il existe des tonnes de bibliothèques comme celle-là, la plupart pouvant être téléchargées avec **NPM**, le gestionnaire de paquets de **Node.js** (nous apprendrons à l'utiliser plus tard).

La variable `http` représente un objet **JavaScript** qui va nous permettre de lancer un serveur web. C'est justement ce qu'on fait avec :

```
1 | var server = http.createServer();
```

On appelle la fonction `createServer()` contenue dans l'objet `http` et on enregistre ce serveur dans la variable `server`. Vous remarquerez que la fonction `createServer` prend un paramètre... et que ce paramètre est une fonction ! C'est pour ça que l'instruction est un peu compliquée, puisqu'elle s'étend sur plusieurs lignes :

```
1 | var server = http.createServer(function(req, res) {  
2 |     res.writeHead(200);  
3 |     res.end('Salut tout le monde !');  
4 | });
```

Tout ce code correspond à l'appel à `createServer()`. Il comprend en paramètre la *fonction à exécuter quand un visiteur se connecte à notre site*.

Notez que vous pouvez faire ça en deux temps comme je vous l'avais dit. La fonction à exécuter est la **fonction de callback**. On peut la définir avant dans une variable et transmettre cette variable à `createServer()`. Ainsi, le code suivant est strictement identique au précédent :

```
1 | // Code identique au précédent  
2 |  
3 | var instructionsNouveauVisiteur = function(req, res) {  
4 |     res.writeHead(200);  
5 |     res.end('Salut tout le monde !');  
6 | }  
7 |  
8 | var server = http.createServer(instructionsNouveauVisiteur);
```

Il est très important que vous compreniez ce principe, car **Node.js** ne fonctionne que comme ça. Il y a des **fonctions de callback** partout et, en général, elles sont placées à l'intérieur des arguments d'une autre fonction comme je l'ai fait dans mon premier code. Cela paraît un peu délicat à lire, mais vous prendrez vite le pli, rassurez-vous.

N'oubliez pas de bien fermer la **fonction de callback** avec une accolade, de fermer les parenthèses d'appel de la fonction qui l'englobe, puis de placer le fameux point-virgule. C'est pour ça que vous voyez les symboles `});` à la dernière ligne de mon premier code.

La **fonction de callback** est donc appelée à chaque fois qu'un visiteur se connecte à notre site. Elle prend deux paramètres :

1. La requête du visiteur (**req** dans mes exemples) : cet objet contient toutes les informations sur ce que le visiteur a demandé. On y trouve le nom de la page appelée, les paramètres, les éventuels champs de formulaires remplis.
2. La réponse que vous devez renvoyer (**res** dans mes exemples) : c'est cet objet qu'il faut remplir pour donner un retour au visiteur. Au final, **res** contiendra en général le code HTML de la page à renvoyer au visiteur.

Ici, on effectue deux choses très simples dans la réponse :

```
1 | res.writeHead(200);
2 | res.end('Salut tout le monde !');
```

On renvoie le code 200 dans l'en-tête de la réponse, qui signifie au navigateur « OK tout va bien » (on aurait par exemple répondu 404 si la page demandée n'existait pas). Il faut savoir qu'en plus du code HTML, le serveur renvoie en général tout un tas de paramètres en en-tête. Il faut connaître la norme HTTP qui indique comment clients et serveurs doivent communiquer pour bien l'utiliser. Voilà encore un exemple de la complexité due au fait que **Node.js** est *bas niveau*. Mais en même temps, cela nous fait comprendre tout un tas de choses.

Ensuite, on termine la réponse (avec **end()**) en envoyant le message de notre choix au navigateur. Ici, on n'envoie même pas de HTML, juste du texte brut.

Enfin, le serveur est lancé et « écoute » sur le port 8080 avec l'instruction :

```
1 | server.listen(8080);
```



On évite d'utiliser ici le port 80 qui est normalement réservé aux serveurs web car celui-ci est peut-être déjà utilisé par votre machine. Ce port 8080 sert juste pour nos tests évidemment, une fois en production il est conseillé au contraire d'écouter cette fois sur le port 80, car c'est à cette porte (à ce port) que vos visiteurs iront taper en arrivant sur votre serveur.

Tester le serveur HTTP

Pour tester votre premier serveur, rendez-vous dans la console et tapez :

```
node serveur.js
```

La console n'affiche rien et ne répond pas, ce qui est parfaitement normal. Ouvrez maintenant votre navigateur et rendez-vous à l'adresse **http://localhost:8080**. Vous allez vous connecter sur votre propre machine sur le port 8080 sur lequel votre programme **Node.js** est en train d'écouter ! Vous devriez obtenir quelque chose ressemblant à la figure 3.3.

Pour arrêter votre serveur **Node.js**, retournez dans la console et faites **Ctrl** + **C** pour couper la commande.

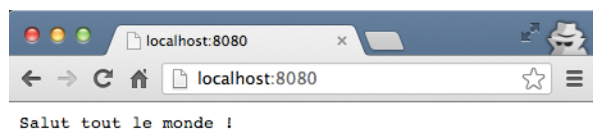


FIGURE 3.3 – Notre premier programme Node.js s’affiche dans le navigateur !

Retourner du code HTML

Résumons ! Nous avons créé notre première véritable application avec son serveur web embarqué. Mais l’application est pour l’instant minimaliste :

- Le message renvoyé est du texte brut, il ne comporte même pas de HTML !
- L’application renvoie toujours le même message, quelle que soit la page appelée (`http://localhost:8080`, `http://localhost:8080/mapage`, `http://localhost:8080/dossier/autrepape`).

Pour que ce chapitre soit complet, nous allons voir comment remédier à ces deux problèmes. Commençons ici par voir comment faire pour renvoyer du HTML.

Comme je vous l’ai dit, il y a des règles à respecter entre le client et le serveur. Ils communiquent en se basant sur la **norme HTTP** inventée par Tim Berners-Lee. Cette norme est à la base du web (tout comme le **langage HTML** qui a aussi été inventé par ce même monsieur).

Que dit la **norme HTTP** ? Que le serveur doit indiquer le *type de données* qu’il s’apprête à envoyer au client. Et oui, un serveur peut renvoyer différents types de données :

- Du **texte brut** : `text/plain`.
- Du **HTML** : `text/html`.
- Du **CSS** : `text/css`.
- Une **image JPEG** : `image/jpeg`.
- Une **vidéo MPEG4** : `video/mp4`.
- Un **fichier ZIP** : `application/zip`.
- Etc.

Ce sont ce qu’on appelle les **types MIME**. Ils sont envoyés dans l’en-tête de la réponse du serveur. Vous vous souvenez comment on écrit dans l’en-tête de la réponse avec **Node.js** ? Nous avons écrit ceci :

```
1 | res.writeHead(200);
```

Nous avons seulement indiqué le code de réponse 200 qui signifie « OK, pas d'erreur ». Nous devons rajouter un paramètre qui indique le **type MIME** de la réponse. Pour HTML, ce sera donc :

```
1 | res.writeHead(200, {"Content-Type": "text/html"});
```



Le second paramètre est entre accolades, car on peut y envoyer plusieurs valeurs sous forme de tableau.

Maintenant que c'est fait, nous pouvons renvoyer du HTML dans la réponse !

```
1 | res.end('<p>Voici un paragraphe <strong>HTML</strong> !</p>');
```

Au final, notre code ressemble donc maintenant à ceci :

```
1 | var http = require('http');
2 |
3 | var server = http.createServer(function(req, res) {
4 |     res.writeHead(200, {"Content-Type": "text/html"});
5 |     res.end('<p>Voici un paragraphe <strong>HTML</strong> !</p>');
6 | });
7 | server.listen(8080);
```

Essayez-le comme vous l'avez appris, en lançant l'application avec la commande **node** dans la console et en ouvrant votre navigateur sur <http://localhost:8080> (voir figure 3.4).

Voici un paragraphe **HTML** !

FIGURE 3.4 – Un paragraphe HTML renvoyé par notre appli Node.js

Votre paragraphe de texte s'affiche et est bien mis en forme comme prévu !



Mais le code HTML n'est pas valide non ? On n'a pas écrit de doctype, ni la balise `<html>`, ni la balise `<body>`.

Grmpf, vous m'avez pris la main dans le sac ! Un **code HTML** invalide, j'ai honte. Réparons ça ! C'est facile, il suffit d'envoyer toutes les autres balises qui manquent.



Jusqu'ici, nous avons toujours écrit le **code HTML** dans `res.end()`. Pour mieux découper le code, à partir de maintenant, j'utilise la commande `res.write()` qui permet d'écrire la réponse en plusieurs temps. Ça revient au même, mais notre code est mieux découpé comme ça. `res.end()` doit toujours être appelé en dernier pour terminer la réponse et faire en sorte que le serveur envoie le résultat au client.

```
1 | var http = require('http');
2 |
3 | var server = http.createServer(function(req, res) {
4 |     res.writeHead(200, {"Content-Type": "text/html"});
5 |     res.write('<!DOCTYPE html>'+
6 | '<html>'+
7 |     '<head>'+
8 |         '<meta charset="utf-8" />'+
9 |         '<title>Ma page Node.js !</title>'+
10 |     '</head>'+
11 |     '<body>'+
12 |         '<p>Voici un paragraphe <strong>HTML</strong> !</p>'+
13 |     '</body>'+
14 | '</html>');
15 |     res.end();
16 | });
17 | server.listen(8080);
```



Mais c'est atroce d'écrire du HTML comme ça !

On fait ce qu'on peut avec ce qu'on a ! Rappelez-vous que **Node.js** est *bas niveau*...

Je vous rassure, aucun développeur ne s'amusera vraiment à faire des **pages web HTML** complexes comme ça là-dedans. Il existe des moyens de séparer le **code HTML** du **code JavaScript** : ce sont les systèmes de **templates**. C'est un peu hors sujet pour le moment, étant donné que nous commençons tout juste à découvrir les bases de **Node.js**. Mais si le sujet vous intéresse, sachez qu'il existe des tonnes de modules (voir le code web suivant) :

▷ Des modules **Node.js** dédiés
aux **templates**
Code web : 272512

Le choix est immense !

Déterminer la page appelée et les paramètres

Nous savons renvoyer du **code HTML**, mais pour le moment notre application renvoie toujours la même chose ! Comment fait-on pour créer différentes pages avec **Node.js** ? Essayez notre petite application sur différentes URL. Quelque soit la page appelée, la page qui s'affiche est toujours la même !

Il faut qu'on sache quelle est la page demandée par le visiteur. Pour l'instant, vu que nous ne faisons aucun test, notre application renvoie toujours la même chose.

Nous allons découvrir comment récupérer :

- Le nom de la page demandée (`/mapage`, `/page.html`, `/dossier/autrepape...`).
- Les paramètres qui circulent dans l'URL.

Quelle est la page demandée par le visiteur ?

Pour récupérer la page demandée par le visiteur, on va faire appel à un nouveau module de Node appelé « **url** ». On demande son inclusion avec :

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

Ensuite, il nous suffit de « parser » la requête du visiteur comme ceci pour obtenir le nom de la page demandée :

```
1 | url.parse(req.url).pathname;
```

Voici un code très simple qui nous permet de tester ça :

```
1 | var http = require('http');
2 | var url = require('url');
3 |
4 | var server = http.createServer(function(req, res) {
5 |   var page = url.parse(req.url).pathname;
6 |   console.log(page);
7 |   res.writeHead(200, {"Content-Type": "text/plain"});
8 |   res.write('Bien le bonjour');
9 |   res.end();
10 | });
11 | server.listen(8080);
```

Exécutez ce script et lancez votre navigateur à l'adresse `http://localhost:8080` pour commencer. Retournez ensuite dans la console. Nous y loggons le nom de la page demandée. Vous devriez y voir :

```
/
/favicon.ico
```



Je n'ai chargé que la page d'accueil, pourquoi est-ce que je vois `/favicon.ico` ?

La plupart des navigateurs font en réalité une seconde requête pour récupérer l'icône du site (la « favicon » qu'on voit dans les onglets en général). C'est normal, ne vous en préoccupez pas.

Essayez maintenant de charger de « fausses pages » de votre site pour voir ce que ça fait.

```
/testpage
/favicon.ico
/un/long/chemin/
/favicon.ico
/faussepage.html
/favicon.ico
```

Si on omet les `favicon.ico` qui viennent un peu polluer la console, on voit que j'ai essayé de charger les pages suivantes :

- `http://localhost:8080/testpage`
- `http://localhost:8080/un/long/chemin`
- `http://localhost:8080/faussepage.html`

Vous pouvez retrouver la page directement grâce au code web suivant :

▷

Page faussepage
Code web : 528764



Et alors ? Mon site renvoie toujours la même chose quelle que soit la page appelée !

En effet. Mais il suffit d'écrire une condition, et le tour est joué !

```
1 | var http = require('http');
2 | var url = require('url');
3 |
4 | var server = http.createServer(function(req, res) {
5 |     var page = url.parse(req.url).pathname;
6 |     console.log(page);
7 |     res.writeHead(200, {"Content-Type": "text/plain"});
8 |     if (page == '/') {
9 |         res.write('Vous êtes à l\'accueil, que puis-je pour
          vous ?');
10 |     }
11 |     else if (page == '/sous-sol') {
12 |         res.write('Vous êtes dans la cave à vins, ces
          bouteilles sont à moi !');
13 |     }
14 |     else if (page == '/etage/1/chambre') {
15 |         res.write('Hé ho, c\'est privé ici !');
16 |     }
17 |     res.end();
```

```
18 | });  
19 | server.listen(8080);
```

Allez un petit défi pour vous entraîner : faites en sorte d'afficher un message d'erreur si le visiteur demande une page inconnue. Et n'oubliez pas de renvoyer un code d'erreur 404!

Quels sont les paramètres ?

Les paramètres sont envoyés à la fin de l'URL, après le chemin du fichier. Prenez cette URL par exemple : `http://localhost:8080/page?prenom=Robert&nom=Dupont`.

Les paramètres sont contenus dans la chaîne `?prenom=Robert&nom=Dupont`. Pour récupérer cette chaîne, il suffit de faire appel à :

```
1 | url.parse(req.url).query
```

Le problème, c'est qu'on vous renvoie toute la chaîne sans découper au préalable les différents paramètres. Heureusement, il existe un module **Node.js** qui s'en charge pour nous : **querystring**!

Incluez ce module :

```
1 | var querystring = require('querystring');
```

Vous pourrez ensuite faire :

```
1 | var params = querystring.parse(url.parse(req.url).query);
```

Vous disposerez alors d'un tableau de paramètres `params`. Pour récupérer le paramètre `prenom` par exemple, il suffira d'écrire : `params['prenom']`.

Amusons-nous avec un code complet qui affiche votre prénom et votre nom (pourvu que ceux-ci soient définis) :

```
1 | var http = require('http');  
2 | var url = require('url');  
3 | var querystring = require('querystring');  
4 |  
5 | var server = http.createServer(function(req, res) {  
6 |     var params = querystring.parse(url.parse(req.url).query);  
7 |     res.writeHead(200, {"Content-Type": "text/plain"});  
8 |     if ('prenom' in params && 'nom' in params) {  
9 |         res.write('Vous vous appelez ' + params['prenom'] + ' '  
10 |             + params['nom']);  
11 |     }  
12 |     else {  
13 |         res.write('Vous devez bien avoir un prénom et un nom,  
14 |             non ?');  
15 |     }  
16 |     res.end();  
17 | });  
18 | server.listen(8080);
```

Essayez d'aller sur `http://localhost:8080?prenom=Robert&nom=Dupont` pour voir, puis changez le prénom et le nom pour les remplacer par les vôtres !



Deux petites précisions par rapport à ce code : `'prenom'` in `params` me permet en **JavaScript** de tester si le tableau contient bien une entrée `'prenom'`. S'il manque un paramètre, je peux alors afficher un message d'erreur (sinon mon script aurait affiché `undefined` à la place). Par ailleurs, vous constaterez que je ne vérifie pas la page qui est appelée. Ce code fonctionne aussi bien que l'on soit sur `http://localhost:8080` ou sur `http://localhost:8080/pageimaginaire`. Il faudrait combiner ce code et le précédent pour gérer à la fois la page et les paramètres.

Schéma résumé

Résumons ce que nous venons d'apprendre dans un seul et unique schéma (voir figure 3.5) avant de terminer.

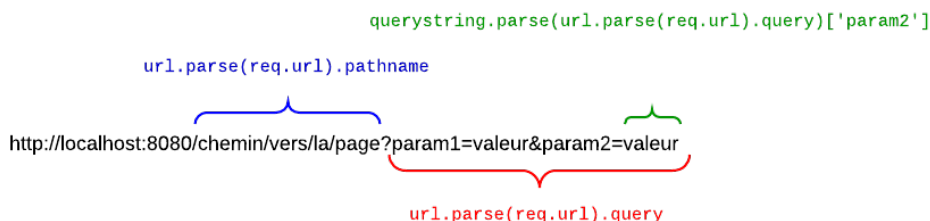


FIGURE 3.5 – Récupérer l'URL et les paramètres avec Node.js

En résumé

- Avec **Node.js**, votre application doit gérer le serveur web (équivalent d'Apache) en plus de ses propres fonctionnalités. Elle doit notamment déterminer le nom de la page demandée et ses paramètres pour savoir ce qu'elle doit renvoyer au visiteur.
- Les bibliothèque de **Node.js** sont appelées des **modules** et on les charge avec `require()`
- Avec **Node.js**, on envoie fréquemment des fonctions en paramètre d'autres fonctions. Cela permet d'indiquer la fonction suivante à appeler lorsqu'une tâche est terminée.
- Votre application devra généralement renvoyer du **HTML** au navigateur du visiteur avec la méthode `write()`.

Deuxième partie

Structurer son application Node.js

Chapitre 4

Les évènements

Difficulté : 

Node.js est un environnement de développement **JavaScript** basé sur les **évènements**. Je vous ai montré dans le premier chapitre ce que ça signifie : il y a un seul **thread** mais aucune opération n'est bloquante. Ainsi, les opérations un peu longues (chargement d'un fichier, téléchargement d'une page web, démarrage d'un serveur web...) sont lancées en tâche de fond et une **fonction de callback** est appelée quand l'opération est terminée.

Les **évènements** sont à la base de **Node.js**. C'est ce qui fait que **Node.js** est puissant mais aussi un peu plus difficile à appréhender puisque il nous impose de coder avec beaucoup de **fonctions de callback**. Je vous propose de rentrer ici dans les détails du fonctionnement des **évènements Node.js**. Nous verrons en particulier comment on peut « écouter » et « créer » des **évènements**. Tout développeur **Node.js** qui se respecte doit savoir le faire, alors au boulot !



Écouter des évènements

Surprise ! Vous savez déjà **écouter des évènements en JavaScript**. Vous n'allez pas me faire croire que vous n'avez jamais utilisé une bibliothèque comme **jQuery** pour **écouter des évènements** sur votre page web !

Par exemple :

```
1 | $("canvas").on("mouseleave", function() { ... });
```

Avec ce genre d'instruction, vous demandez à exécuter une **fonction de callback** quand la souris sort d'un élément `<canvas>` de la page. On dit que vous **attachez l'évènement au DOM de la page**.

Avec **Node.js**, le principe est exactement le même. Un très grand nombre d'objets **Node.js** émettent des **évènements**. Leur particularité ? Ils héritent tous d'un objet **EventEmitter** fourni par Node.

Prenons par exemple le module « http » que nous avons utilisé pour créer notre serveur web. Il comprend un objet **Server** qui émet des évènements. La figure 4.1 est un extrait du document que vous retrouverez grâce au code web suivant :

▷ La doc
Code web : 438864

Table of Contents

- [HTTP](#)
 - [http.STATUS_CODES](#)
 - [http.createServer\(\[requestListener\]\)](#)
 - [http.createClient\(\[port\], \[host\]\)](#)
 - [Class: http.Server](#)
 - [Event: 'request'](#)
 - [Event: 'connection'](#)
 - [Event: 'close'](#)
 - [Event: 'checkContinue'](#)
 - [Event: 'connect'](#)
 - [Event: 'upgrade'](#)
 - [Event: 'clientError'](#)

FIGURE 4.1 – La doc de Node.js indique les évènements que les objets émettent

Comment **écouter ces évènements** ? Supposons par exemple qu'on souhaite écouter l'évènement `close` qui survient quand le serveur est arrêté. Il suffit de faire appel à la méthode `on()` et d'indiquer :

- Le **nom de l'évènement** que vous écoutez (ici `close`).
- La **fonction de callback** à appeler quand l'évènement survient.

Exemple :

```
1 | server.on('close', function() {  
2 |     // Faire quelque chose quand le serveur est arrêté  
3 | })
```

Je vous propose un exemple concret et complet. On va lancer un serveur et l'arrêter juste après. On écoute l'évènement `close` qui survient lorsque le serveur est arrêté. On affiche un message dans la console quand le serveur s'apprête à s'arrêter.

```

1 | var http = require('http');
2 |
3 | var server = http.createServer(function(req, res) {
4 |     res.writeHead(200);
5 |     res.end('Salut tout le monde !');
6 | });
7 |
8 | server.on('close', function() { // On écoute l'évènement close
9 |     console.log('Bye bye !');
10 | })
11 |
12 | server.listen(8080); // Démarre le serveur
13 |
14 | server.close(); // Arrête le serveur. Déclenche l'évènement
    close

```



Mais au fait... `createServer()` comprend une fonction de callback lui aussi. Pourquoi on n'utilise pas `on()` ici ?

Bien vu ! En fait, c'est une **contraction de code**. Lisez la section "CreateServer" sur la doc indiquée sur le code web précédent. Elle dit que la **fonction de callback** qu'on lui envoie en paramètre est automatiquement ajoutée à l'évènement `request` !

Donc ce code :

```

1 | var server = http.createServer(function(req, res) { });

```

... peut être réécrit comme ceci de façon plus détaillée :

```

1 | // Code équivalent au précédent
2 | var server = http.createServer();
3 | server.on('request', function(req, res) { });

```

Bref, les **événements** sont partout, vous ne pouvez pas y échapper ! Certains sont simplement un peu « masqués » comme c'est le cas ici, mais il est important de savoir ce qui se passe derrière.



Vous pouvez écouter *plusieurs fois* un même événement. Faites deux fois appel à la fonction `on()` pour le même événement : les deux fonctions de callback seront appelées quand l'évènement aura lieu.

Émettre des événements

Si vous voulez **émettre des événements** vous aussi, c'est très simple : incluez le module `EventEmitter` et créez un objet basé sur `EventEmitter`.

```
1 | var EventEmitter = require('events').EventEmitter;
2 |
3 | var jeu = new EventEmitter();
```

Ensuite, pour émettre un **événement** dans votre code, il suffit de faire appel à `emit()` depuis votre objet basé sur `EventEmitter`. Indiquez :

- Le nom de l'événement que vous voulez générer (ex : `gameover`). À vous de le choisir.
- Un ou plusieurs éventuels paramètres à passer (facultatif).

Ici, je génère un événement `gameover` et j'envoie un message à celui qui réceptionnera l'événement *via* un paramètre :

```
1 | jeu.emit('gameover', 'Vous avez perdu !');
```

Celui qui veut **écouter l'événement** doit ensuite faire :

```
1 | jeu.on('gameover', function(message) { });
```

Voici un code complet pour tester l'**émission d'événements** :

```
1 | var EventEmitter = require('events').EventEmitter;
2 |
3 | var jeu = new EventEmitter();
4 |
5 | jeu.on('gameover', function(message){
6 |     console.log(message);
7 | });
8 |
9 | jeu.emit('gameover', 'Vous avez perdu !');
```

Je l'admets, c'est un peu trop simple. Ce code se contente d'**émettre un événement**. Dans la réalité, les **événements** seront émis depuis des fonctions imbriquées dans d'autres fonctions, c'est de là que **Node.js** tire toute sa richesse.

Comme vous le voyez, le principe n'est pas franchement compliqué à comprendre !

N'oubliez pas que vous pouvez envoyer autant de paramètres que nécessaire à la **fonction de callback**. Émettez simplement plus de paramètres :

```
1 | jeu.emit('nouveaujoueur', 'Mario', 35); // Envoie le nom d'un
   |     nouveau joueur qui arrive et son âge
```

En résumé

- Toutes les applications **Node.js** sont basées sur un mécanisme d'**événements**, qui détermine quelle est la prochaine fonction à appeler.

- Avec la méthode `on()`, vous pouvez **écouter un évènement** et indiquer quelle fonction doit être appelée lorsque l'évènement survient.
- Avec la méthode `emit()` vous pouvez **créer votre propre évènement** et donc provoquer le déclenchement des fonctions qui attendent cet évènement.

Chapitre 5

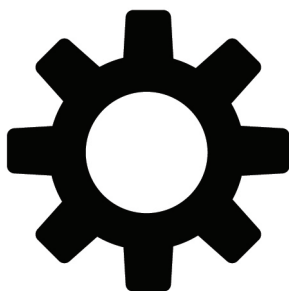
Les modules Node.js et NPM

Difficulté : 

Croyez-le ou non, le noyau de **Node.js** est tout petit. A la base, **Node.js** ne sait en fait pas faire grand chose. Pourtant, **Node.js** est très riche grâce à son *extensibilité*. Ces extensions de **Node.js** sont appelées **modules**.

Il existe des milliers de **modules** qui offrent des fonctionnalités variées : de la gestion des fichiers uploadés à la connexion aux **bases de données MySQL** ou à **Redis**, en passant par des **frameworks**, des systèmes de **templates** et la gestion de la communication temps réel avec le visiteur ! Il y a à peu près tout ce dont on peut rêver et de nouveaux **modules** apparaissent chaque jour.

Nous allons commencer par voir comment sont gérés les **modules** par **Node.js** et nous verrons que nous pouvons facilement en créer un nous aussi. Puis, nous découvrirons **NPM** (Node Package Manager), l'outil indispensable qui vous permet de télécharger facilement tous les **modules** de la communauté **Node.js** ! Enfin, je vous montrerai comment accéder à la gloire éternelle en publiant votre **module** sur **NPM**.



Créer des modules

Vous souvenez-vous de cette ligne ?

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

Elle était tout au début de notre premier code. Je vous avais dit que c'était un appel à la bibliothèque « http » de **Node.js** (ou devrais-je dire au **module** « http »).

Lorsque l'on effectue cela, **Node.js** va chercher sur notre disque un fichier appelé `http.js`. De même, si on demande le module « url », **Node.js** va rechercher un fichier appelé `url.js`.

```
1 | var http = require('http'); // Fait appel à http.js
2 | var url = require('url'); // Fait appel à url.js
```



Où sont ces fichiers `.js` ? Je ne les vois pas !

Ils sont quelque part bien au chaud sur votre disque, leur position ne nous intéresse pas. Étant donné qu'ils font partie du noyau de **Node.js**, ils sont tout le temps disponibles.

Les **modules** sont donc de *simples fichiers .js*. Si nous voulons créer un **module**, disons le module « test », nous devons créer un fichier `test.js` dans le même dossier et y faire appel comme ceci :

```
1 | var test = require('./test'); // Fait appel à test.js (même dossier)
```



Il ne faut pas mettre l'extension `.js` dans le `require()` !

C'est un chemin relatif. Si le module se trouve dans le dossier parent, nous pouvons l'inclure comme ceci :

```
1 | var test = require('../test'); // Fait appel à test.js (dossier parent)
```



Et si je ne veux pas mettre de chemin relatif ? Je ne peux pas juste faire `require('test')` ?

Si vous pouvez ! Il faut mettre votre fichier `test.js` dans un sous-dossier appelé `node_modules`. C'est une convention de **Node.js** :

```
1 | var test = require('test'); // Fait appel à test.js (sous-dossier node_modules)
```

La figure 5.1 résume tout ça.

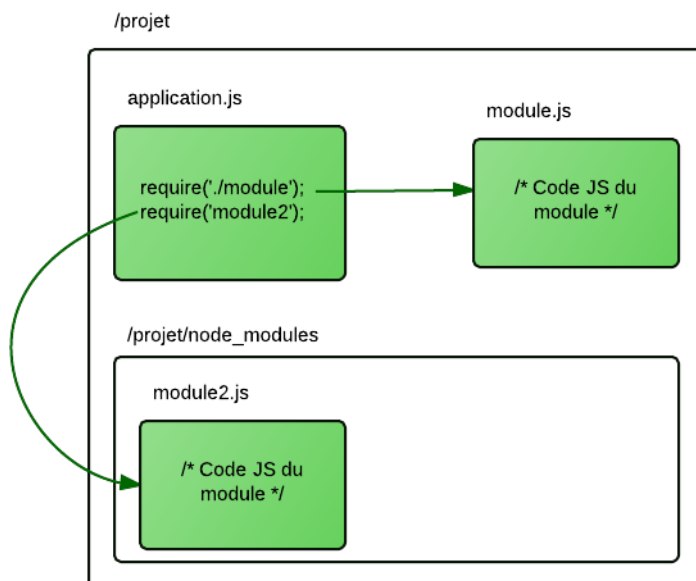


FIGURE 5.1 – Node.js sait où chercher les modules

Notez que si le dossier **node_modules** n'existe pas, **Node.js** ira chercher un dossier qui a le même nom plus haut dans l'arborescence. Ainsi, si votre projet se trouve dans le dossier : `/home/mateo21/dev/nodejs/projet`, il ira chercher un dossier nommé :

- `/home/mateo21/dev/nodejs/projet/node_modules`, et si ce dossier n'existe pas il ira le chercher dans...
- `... /home/mateo21/dev/nodejs/node_modules`, et si ce dossier n'existe pas il ira le chercher dans...
- `... /home/mateo21/dev/node_modules`, et ainsi de suite !



À quoi ressemble le code des fichiers .js des modules ?

C'est du **code JavaScript** tout ce qu'il y a de plus classique. Vous y créez des fonctions. Une seule particularité : vous devez « exporter » les fonctions que vous voulez que d'autres personnes puissent réutiliser.

Testons cela ! Nous allons créer un module tout bête qui sait dire « Bonjour ! » et « Bye bye ! ». Créez un fichier `monmodule.js` avec le code suivant :

```

1 | var direBonjour = function() {
2 |     console.log('Bonjour !');
3 | }
  
```

```
4 |  
5 | var direByeBye = function() {  
6 |     console.log('Bye bye !');  
7 | }  
8 |  
9 | exports.direBonjour = direBonjour;  
10| exports.direByeBye = direByeBye;
```

Le début du fichier ne contient rien de nouveau. Nous créons des fonctions que nous plaçons dans des variables. D'où le `var direBonjour = function()`.

Ensuite, et c'est la nouveauté, nous exportons ces fonctions pour qu'elles soient utilisables de l'extérieur : `exports.direBonjour = direBonjour;`. Notez d'ailleurs qu'on aurait aussi pu faire directement :

```
1 | exports.direBonjour = function() { ... };
```



Toutes les fonctions que vous n'exportez pas dans votre fichier de module resteront privées. Elles ne pourront pas être appelées de l'extérieur. En revanche, elles pourront tout à fait être utilisées par d'autres fonctions de votre module.

Maintenant, dans le fichier principal de votre application (ex : `app.js`), vous pouvez faire appel à ces fonctions issues du module !

```
1 | var monmodule = require('./monmodule');  
2 |  
3 | monmodule.direBonjour();  
4 | monmodule.direByeBye();
```

`require()` renvoie en fait un objet qui contient les fonctions que vous avez exportées dans votre module. Nous stockons cet objet dans une variable du même nom `monmodule` (mais on aurait pu lui donner n'importe quel autre nom), comme à la figure 5.2.

Voilà, vous savez tout ! Tous les **modules de Node.js** sont basés sur ce principe très simple. Cela vous permet de découper votre projet en plusieurs petits fichiers pour répartir les rôles.

Utiliser NPM pour installer des modules

Je vous ai parlé en introduction de NPM, le Node Package Manager (voir le code web suivant) :

▷

NPM
Code web : 953624

Je vous ai dit que c'était un moyen (formidable) d'installer de nouveaux modules développés par la communauté.

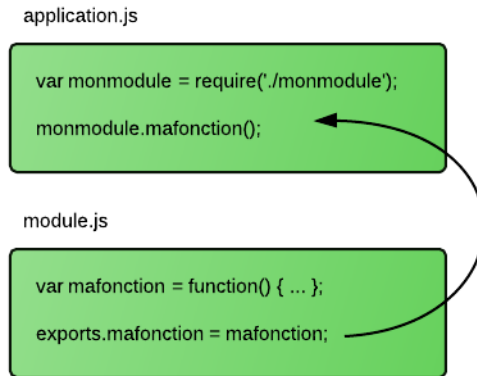


FIGURE 5.2 – Exportation d’une fonction

Imaginez que **NPM** est un peu l’équivalent d’apt-get sous Linux pour installer des programmes. Une simple commande et le module est téléchargé et installé ! En plus, **NPM** gère les dépendances. Cela signifie que, si un module a besoin d’un autre module pour fonctionner, **NPM** ira le télécharger automatiquement !

NPM est très actif, il y a plusieurs dizaines de milliers de modules disponibles, et on compte plusieurs millions de téléchargements par semaine ! Vous y trouverez sans doute votre bonheur.

Comme le dit le site web de **NPM**, il est aussi simple d’installer de nouveaux modules que de publier ses propres modules. C’est en bonne partie ce qui explique le grand succès de **Node.js**.



Comment trouver un module là-dedans ? Autant chercher une aiguille dans une botte de foin !

C’est assez facile en fait. Je vais vous montrer plusieurs moyens de trouver des modules !

Trouver un module

Si vous savez ce que vous cherchez, le site web de **NPM** vous permet de faire une recherche. Mais **NPM**, c’est avant tout une commande ! Vous pouvez faire une recherche dans la console, comme ceci :

```
npm search postgresql
```

Ce qui aura pour effet de rechercher tous les modules en rapport avec la base de données PostgreSQL.

Si comme moi vous voulez flâner un peu et que vous ne savez pas trop ce que vous recherchez, vous aimerez sûrement le site Nodetoolbox (voir le code web suivant) :

▷ Le site [nodetoolbox](#)
Code web : 537828

Il organise les modules par thématique (voir figure 5.3).

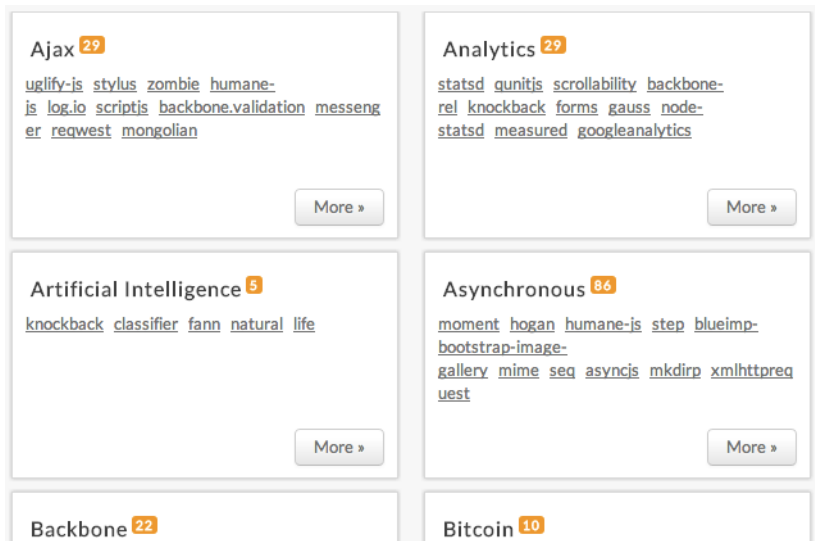


FIGURE 5.3 – Le site Node Toolbox est un bon moyen de découvrir de nouveaux modules

Installer un module

Pour **installer un module**, rien de plus simple. Placez-vous dans le dossier de votre projet et tapez :

```
npm install nomdumodule
```

Le module sera installé *localement* spécialement pour votre projet. Si vous avez un autre projet, il faudra donc relancer la commande pour l'installer à nouveau pour cet autre projet. Cela vous permet d'utiliser des versions différentes d'un même module en fonction de vos projets.

Allez faisons un test. On va installer le module `markdown` qui permet de convertir du code markdown en HTML.

```
npm install markdown
```

NPM va télécharger automatiquement la dernière version du module et il va la placer dans un sous-dossier `node_modules`. Vérifiez donc bien que vous êtes dans le dossier de votre projet **Node.js** avant de lancer cette commande !

Une fois que c'est fait, vous avez accès aux fonctions offertes par le module « markdown ». Lisez la documentation du module pour savoir comment l'utiliser (voir code web suivant) :

▷ Doc du module
Code web : 353493

On nous apprend qu'il faut faire appel à l'objet `markdown` à l'intérieur du module et qu'on peut appeler la fonction `toHTML` pour traduire du Markdown en HTML.

Essayons ça :

```
1 | var markdown = require('markdown').markdown;
2 |
3 | console.log(markdown.toHTML('Un paragraphe en markdown !'))
   | ;
```

Cela affichera dans la console :

```
<p>Un paragraphe en <strong>markdown</strong> !</p>
```



Ne soyez pas surpris par le `require('markdown').markdown`. La doc du module nous dit que les fonctions sont dans l'objet `markdown`, donc on va chercher directement cet objet au sein du module.

L'installation locale et l'installation globale

NPM installe les modules localement pour chaque projet. C'est pour cela qu'il crée un sous-dossier `node_modules` à l'intérieur de votre projet. Si vous utilisez le même module dans trois projets différents, il sera téléchargé et copié trois fois. C'est normal, cela nous permet de gérer les différences de versions. C'est donc une bonne chose.

Par contre, il faut savoir que **NPM** permet aussi d'installer des **modules globaux**. Cela ne nous sert que dans de rares cas où le module fournit des exécutables (et pas juste des fichiers `.js`).

C'est le cas de notre module « markdown » par exemple. Pour l'installer globalement, on va ajouter le paramètre `-g` à la commande `npm` :

```
npm install markdown -g
```

Vous aurez alors accès à un exécutable `md2html` dans votre console :

```
echo 'Hello *World*!' | md2html
```



Les modules installés globalement ne peuvent pas être inclus dans vos projets **Node.js** avec `require()` ! Ils servent juste à fournir des commandes supplémentaires dans la console. Si vous voulez les utiliser en **JavaScript**, vous devez donc aussi les installer normalement (sans le `-g`) en local.

Mettre à jour les modules

Avec une simple commande :

```
npm update
```

NPM va chercher sur les serveurs s'il y a de nouvelles versions des modules, puis mettre à jour les modules installés sur votre machine (en veillant à ne pas casser la compatibilité) et il supprimera les anciennes versions. Bref, c'est la commande magique.

Déclarer et publier son module

Si votre programme a besoin de **modules externes**, vous pouvez les installer un à un comme vous venez d'apprendre à le faire, mais vous allez voir que ça va vite devenir assez compliqué à maintenir. C'est d'autant plus vrai si vous utilisez de nombreux modules. Comme ces modules évoluent de version en version, votre programme pourrait devenir incompatible suite à une mise à jour d'un **module externe** !

Heureusement, on peut régler tout ça en définissant les dépendances de notre programme dans un simple **fichier JSON**. C'est un peu la *carte d'identité* de notre application.

Créez un fichier `package.json` dans le même dossier que votre application. Commençons simplement avec ce code :

```
1 {  
2   "name": "mon-app",  
3   "version": "0.1.0",  
4   "dependencies": {  
5     "markdown": "~0.4"  
6   }  
7 }
```

Ce fichier JSON contient 3 paires clé-valeur :

- **name** : c'est le nom de votre application. Restez simple, évitez espaces et accents.
- **version** : c'est le numéro de version de votre application. Il est composé d'un numéro de version majeure, de version mineure et de patch. J'y reviens juste après.
- **dependencies** : c'est un tableau listant les noms des modules dont a besoin votre application pour fonctionner ainsi que les versions compatibles.



Le fichier peut être beaucoup plus complet, je ne vous ai montré ici que les valeurs essentielles. Pour tout connaître sur le fonctionnement de ce fichier `package.json`, je vous recommande de consulter la page indiquée dans le code web suivant.)

▷ Cheat sheet
Code web : 967025

Le fonctionnement des numéros de version

Pour bien **gérer les dépendances** et savoir **mettre à jour le numéro de version de son application**, il faut savoir comment fonctionnent les numéros de version avec **Node.js**. Il y a pour chaque application :

- **Un numéro de version majeure.** En général on commence à 0 tant que l'application n'est pas considérée comme mature. Ce numéro change très rarement, uniquement quand l'application a subi des changements très profonds.
- **Un numéro de version mineure.** Ce numéro est changé à chaque fois que l'application est un peu modifiée.
- **Un numéro de patch.** Ce numéro est changé à chaque petite correction de bug ou de faille. Les fonctionnalités de l'application restent les mêmes entre les patches, il s'agit surtout d'optimisations et de corrections indispensables.

La figure 5.4 récapitule tout ceci :

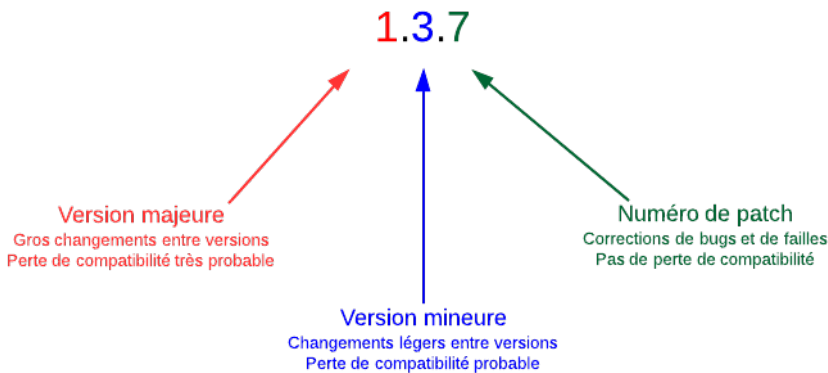


FIGURE 5.4 – Les numéros de version d'une application Node.js

Ici, j'ai donc choisi de commencer à numéroter mon application à la version 0.1.0 (on aurait aussi pu commencer à 1.0.0 mais cela aurait été prétentieux).

- Si je corrige un bug, l'application passera à la version 0.1.1 et il me faudra mettre à jour ce numéro dans le fichier `package.json`.
- Si j'améliore significativement mon application, elle passera à la version 0.2.0, puis 0.3.0 et ainsi de suite.
- Le jour où je considère qu'elle a atteint un jalon important, et qu'elle est mature, je pourrai la passer en version 1.0.0.

La gestion des versions des dépendances

C'est à vous d'indiquer avec quelles versions de ses dépendances votre application fonctionne. Si votre application dépend du module `markdown` v0.3.5 très précisément, vous écririez :

```
1 | "dependencies": {  
2 |   "markdown": "0.3.5" // Version 0.3.5 uniquement  
3 | }
```

Si vous faites un `npm update` pour mettre à jour les modules externes, `markdown` ne sera jamais mis à jour (même si l'application passe en version 0.3.6). Vous pouvez mettre un tilde (`~`) devant le numéro de version pour autoriser les mises à jour jusqu'à la prochaine version mineure :

```
1 | "dependencies": {  
2 |   "markdown": "~0.3.5" // OK pour les versions 0.3.5, 0.3.6,  
   0.3.7, etc. jusqu'à la version 0.4.0 non incluse  
3 | }
```

Si vous voulez, vous pouvez ne pas indiquer de numéro de patch. Dans ce cas, les modules seront mis à jour même si l'application change de version mineure :

```
1 | "dependencies": {  
2 |   "markdown": "~0.3" // OK pour les versions 0.3.X, 0.4.X, 0.  
   5.X jusqu'à la version 1.0.0 non incluse  
3 | }
```

Attention néanmoins : entre deux versions mineures, un module peut changer suffisamment et votre application pourrait être incompatible. Je recommande d'accepter uniquement les mises à jour de patch, c'est le plus sûr.

Publier un module

Avec **Node.js**, vous pouvez créer une application pour vos besoins, mais vous pouvez aussi créer des modules qui offrent des fonctionnalités. Si vous pensez que votre module pourrait servir à d'autres personnes, n'hésitez pas à le partager ! Vous pouvez très facilement le publier sur **NPM** pour que d'autres personnes puissent l'installer à leur tour.



Je rappelle qu'un **module** n'est rien d'autre qu'une **application Node.js** qui contient des instructions **exports** pour partager des fonctionnalités.

Commencez par vous créer un compte utilisateur sur `npm` :

```
npm adduser
```

Une fois que c'est fait, placez-vous dans le répertoire de votre projet à publier. Vérifiez que vous avez :

- Un fichier `package.json` qui décrit votre module (au moins son nom, sa version et ses dépendances).
- Un fichier `README.md` (écrit en markdown) qui présente votre module de façon un peu détaillée. N'hésitez pas à y inclure un mini-tutoriel expliquant comment utiliser votre module !

Il ne vous reste plus qu'à faire :

```
npm publish
```

Et voilà, c'est fait !

Il ne vous reste plus qu'à parler de votre module autour de vous, le présenter sur les mailing-lists de **Node.js**... Les portes de la gloire des geeks barbus vous attendent ! (Voir figure 5.5)



FIGURE 5.5 – Votre module est publié, à vous la gloire !

En résumé

- Vous pouvez créer des **modules Node.js** pour mieux découpler votre code et éviter de tout écrire dans un seul et même fichier.
- Vous appellerez vos modules avec `require()` comme vous le faites pour les **modules officiels de Node.js**.
- Toutes les fonctions dans le fichier de module qui peuvent être appelées depuis l'extérieur doivent être exportées avec la commande `exports`.
- NPM est le gestionnaire de **modules de Node.js**. Comparable à « apt-get » et « aptitude » sous Linux / Debian, il permet d'ajouter de nouveaux modules codés par la communauté en un clin d'œil.
- Les modules téléchargés avec NPM sont par défaut installés localement dans un sous-dossier `node_modules` de votre application.
- L'extension `-g` permet d'installer un module globalement pour tout votre ordinateur. Elle n'est utile que pour certains modules précis.
- Vous pouvez diffuser vos propres **modules** au reste de la communauté en créant

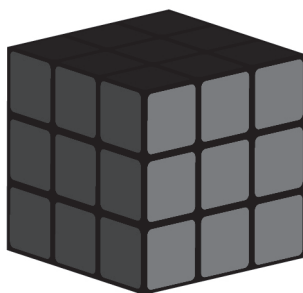
un fichier `package.json`, un `README.md` et avec un simple appel à « npm publish ».

Chapitre 6

Le framework Express.js

Difficulté : 

Tout coder à la main, ça va cinq minutes. Cela peut être utile dans certains cas précis, mais la plupart du temps, on aime avoir des outils à disposition pour aller plus vite. C'est pour cela qu'on a inventé les **bibliothèques** puis les **frameworks**, qui sont des sortes de super-bibliothèques.



Si vous flânez un peu sur **NPM**, vous allez vite voir qu'il existe un module plébiscité : « **Express.js** ». Il s'agit en fait d'un **micro-framework** pour **Node.js**. Il vous fournit des outils de base pour aller plus vite dans la création d'**applications Node.js**.

Mais attention : n'allez pas comparer **Express.js** avec des poids lourds tels **Django** ou **Symfony2** ! Ceux-ci vous offrent des fonctionnalités très complètes et puissantes (comme la génération d'interfaces d'administration), ce qui n'est pas vraiment le cas d'Express. Pourquoi ? Parce que nous partons de loin avec **Node.js**. Express vous permet donc d'être « un peu moins bas niveau » et de gérer par exemple plus facilement les routes (URLs) de votre application et d'utiliser des **templates**. Rien que ça sera déjà une petite révolution pour nous !

Pour suivre ce chapitre, créez-vous un dossier pour faire une application de test **Node.js**. Installez-y **Express** avec la commande `npm install express`.

Les routes

Nous avons vu à quel point il était fastidieux de vérifier l'URL demandée avec **Node.js**. On aboutissait à du bon vieux code spaghetti du type :

```
1 | if (page == '/') {
2 |     // ...
3 | }
4 | else if (page == '/sous-sol') {
5 |     // ...
6 | }
7 | else if (page == '/etage/1/chambre') {
8 |     // ...
9 | }
```

Lorsque l'on crée des applications web, on manipule des **routes** comme ici. Ce sont les différentes **URLs** auxquelles notre application doit répondre.

La **gestion des routes** est un sujet important qui doit être traité avec sérieux. Si vous avez déjà manipulé des **frameworks** comme **Django** ou **Symfony2**, vous voyez certainement ce que je veux dire. Sinon, retenez juste ceci : bien gérer les **URLs** de son site est important, surtout lorsque celui-ci grossit. Et **Express** nous aide à faire cela correctement.

Routes simples

Voici une application très basique utilisant **Express** pour commencer :

```
1 | var express = require('express');
2 |
3 | var app = express();
4 |
5 | app.get('/', function(req, res) {
6 |     res.setHeader('Content-Type', 'text/plain');
```

```

7 |     res.end('Vous êtes à l\'accueil');
8 | });
9 |
10 | app.listen(8080);

```



N'oubliez pas de faire `npm install express` pour que ce code fonctionne !

Vous commencez par demander l'inclusion d'**Express** et vous créez un objet `app` en appelant la fonction `express()`.

Ensuite, il vous suffit d'indiquer les différentes **routes** (les différentes **URLs**) à laquelle votre application doit répondre. Ici, j'ai créé une seule route, la racine « / ». Une **fonction de callback** est appelée lorsque quelqu'un demande cette route.

Ce système est beaucoup mieux conçu que nos `if` imbriqués. On peut écrire autant de routes qu'on le souhaite de cette façon :

```

1 | app.get('/', function(req, res) {
2 |     res.setHeader('Content-Type', 'text/plain');
3 |     res.end('Vous êtes à l\'accueil, que puis-je pour vous ?');
4 | });
5 |
6 | app.get('/sous-sol', function(req, res) {
7 |     res.setHeader('Content-Type', 'text/plain');
8 |     res.end('Vous êtes dans la cave à vins, ces bouteilles sont
   |         à moi !');
9 | });
10 |
11 | app.get('/etage/1/chambre', function(req, res) {
12 |     res.setHeader('Content-Type', 'text/plain');
13 |     res.end('Hé ho, c\'est privé ici !');
14 | });

```

Si vous voulez gérer les **erreurs 404**, vous devez inclure les lignes suivantes à la fin de votre code obligatoirement (juste avant `app.listen`) :

```

1 | // ... Tout le code de gestion des routes (app.get) se trouve
   | au-dessus
2 |
3 | app.use(function(req, res, next){
4 |     res.setHeader('Content-Type', 'text/plain');
5 |     res.send(404, 'Page introuvable !');
6 | });
7 |
8 | app.listen(8080);

```

Ne vous préoccupez pas trop pour l'instant de la syntaxe et des paramètres, nous y reviendrons un peu plus tard. Nous apprenons juste à **gérer les routes avec Express** pour le moment.

Express vous permet de chaîner les appel à `get()` et `use()` :

```
1 | app.get('/', function(req, res) {
2 |
3 | })
4 | .get('/sous-sol', function(req, res) {
5 |
6 | })
7 | .get('/etage/1/chambre', function(req, res) {
8 |
9 | })
10 | .use(function(req, res, next){
11 |
12 | });
```

Cela revient à faire `app.get().get().get()...` Cela fonctionne parce que ces fonctions se renvoient l'une à l'autre l'objet `app`, ce qui nous permet de raccourcir notre code. Ne soyez donc pas étonnés si vous voyez des codes utilisant **Express** écrits sous cette forme.

Routes dynamiques

Express vous permet de gérer des **routes dynamiques**, c'est-à-dire des routes dont certaines portions peuvent varier. Vous devez écrire `:nomvariable` dans l'**URL** de la route, ce qui aura pour effet de créer un paramètre accessible depuis `req.params.nomvariable`. Démonstration :

```
1 | app.get('/etage/:etagenum/chambre', function(req, res) {
2 |     res.setHeader('Content-Type', 'text/plain');
3 |     res.end('Vous êtes à la chambre de 1\'étage n°' + req.
      params.etagenum);
4 | });
```

Cela vous permet de créer de belles **URLs** et vous évite d'avoir à passer par le suffixe ("`?variable=valeur`") pour gérer des variables. Ainsi, toutes les routes suivantes sont valides :

- `/etage/1/chambre`
- `/etage/2/chambre`
- `/etage/3/chambre`
- `/etage/nawak/chambre`



Comment ça, on peut mettre n'importe quoi ? Comment fait-on pour imposer la présence d'un nombre ?

Le visiteur peut en effet écrire n'importe quoi dans l'**URL**. C'est donc à vous de vérifier dans votre **fonction de callback** que le paramètre est bien un nombre et de renvoyer une erreur (ou une 404) si ce n'est pas le cas.

Les templates

Jusqu'ici, nous avons renvoyé le **code HTML** directement en **JavaScript**. Cela nous avait donné du code lourd et délicat à maintenir qui ressemblait à ceci :

```

1 | res.write('<!DOCTYPE html>'+
2 | '<html>'+
3 | '    <head>'+
4 | '        <meta charset="utf-8" />'+
5 | '        <title>Ma page Node.js !</title>'+
6 | '    </head>'+
7 | '    <body>'+
8 | '        <p>Voici un paragraphe <strong>HTML</strong> !</p>'+
9 | '    </body>'+
10| '</html>');

```

Horrible, n'est-ce pas ? Heureusement, Express nous permet d'utiliser des **templates** pour sortir de cet enfer. Les **templates** sont en quelque sorte des langages faciles à écrire qui nous permettent de produire du **HTML** et d'insérer au milieu du contenu variable.

PHP lui-même est en réalité un **langage de template** qui nous permet de faire ceci :

```

1 | <p>Vous êtes le visiteur n°<?php echo $numvisiteur; ?></p>

```

Il existe beaucoup d'autres **langages de templates**, comme **Twig**, **Smarty**, **Haml**, **JSP**, **Jade**, **EJS**, etc. **Express** vous permet d'utiliser la plupart d'entre eux, chacun ayant son lot d'avantages et d'inconvénients. En général, ils gèrent tous l'essentiel, à savoir les **variables**, les **conditions**, les **boucles**, etc.

Le principe est le suivant : depuis votre fichier **JavaScript**, vous appelez le template de votre choix en lui transmettant les variables dont il a besoin pour construire la page (voir figure 6.1).

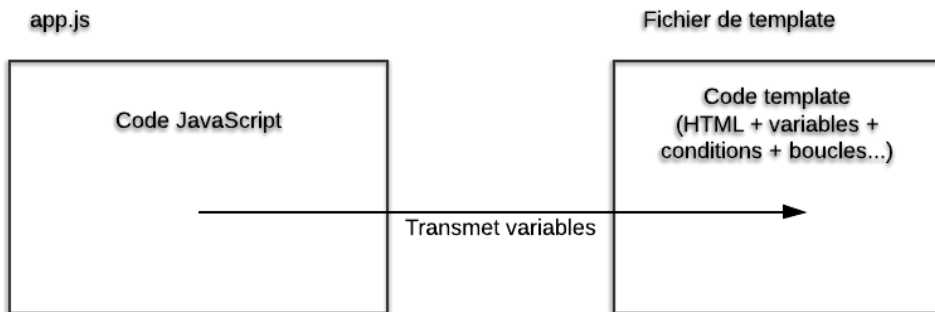


FIGURE 6.1 – Les templates avec Node.js

Les bases d'EJS

Comme il existe de nombreux systèmes de **templates**, je vais en choisir un dans le lot. Je vous propose ici d'utiliser celui proposé dans le code web suivant :

▷

EJS - (Embedded JavaScript)
Code web : 177569

Installez-le pour le projet :

```
npm install ejs
```

Nous pouvons maintenant déléguer la gestion de la vue (du HTML) à notre **moteur de template**. Plus besoin d'écrire du **HTML** au milieu du code **JavaScript** comme un cochon !

```
1 | app.get('/etage/:etagenum/chambre', function(req, res) {
2 |     res.render('chambre.ejs', {etage: req.params.etagenum});
3 | });
```

Ce code fait appel à un fichier `chambre.ejs` qui doit se trouver dans un sous-dossier appelé « **views** ». Créez donc un fichier `/views/chambre.ejs` et placez-y le code suivant :

```
1 | <h1>Vous êtes dans la chambre</h1>
2 |
3 | <p>Vous êtes à l'étage n'<%= etage %></p>
```

La balise `<%= etage %>` sera remplacée par la variable `etage` que l'on a transmise au template avec `{etage: req.params.etagenum}` !

Plusieurs paramètres et des boucles

Sachez que vous pouvez envoyer **plusieurs paramètres** à vos **templates**, y compris des **tableaux** ! Pour cette démonstration, nous allons faire une application qui compte jusqu'à un nombre envoyé en paramètre et qui affiche un nom au hasard au sein d'un tableau (je suis en manque d'inspiration, je le sais bien!).

Voici le code JavaScript :

```
1 | app.get('/compter/:nombre', function(req, res) {
2 |     var noms = ['Robert', 'Jacques', 'David'];
3 |     res.render('page.ejs', {compteur: req.params.nombre, noms:
      noms});
4 | });
```

On transmet le nombre envoyé en paramètre et une liste de noms sous forme de tableau. Ensuite, dans le **template EJS** :

```
1 | <h1>Je vais compter jusqu'à <%= compteur %></h1>
2 |
```

```

3 | <p><%
4 |     for(var i = 1 ; i <= compteur ; i++) {
5 |         %>
6 |
7 |         <%= i %>...
8 |
9 | <% } %></p>
10 |
11 | <p>Tant que j'y suis, je prends un nom au hasard qu'on m'a
    envoyé :
12 | <%= noms[Math.floor(Math.random() * (noms.length + 1))] %>
13 | </p>

```

Vous voyez qu'on peut faire des **boucles** avec les **templates EJS**. En fait, on utilise la même syntaxe que **JavaScript** (d'où la boucle `for`). Ma petite manipulation à la fin du code me permet de prendre un nom au hasard dans le tableau qui a été envoyé au template.

Et le résultat à la figure 6.2 (pour `/compter/66`).

Je vais compter jusqu'à 66

1... 2... 3... 4... 5... 6... 7... 8... 9... 10... 11... 12... 13... 14... 15... 16... 17... 18... 19... 20... 21... 22... 23... 24... 25... 26... 27... 28... 29... 30...
 31... 32... 33... 34... 35... 36... 37... 38... 39... 40... 41... 42... 43... 44... 45... 46... 47... 48... 49... 50... 51... 52... 53... 54... 55... 56... 57... 58...
 59... 60... 61... 62... 63... 64... 65... 66...

Tant que j'y suis, je prends un nom au hasard qu'on m'a envoyé : Robert

FIGURE 6.2 – Cette application est inutile, je sais :o)

De la même façon, vous pouvez avoir recours à des conditions (`if`) et des boucles (`while`) au sein de vos **templates EJS**.



N'hésitez pas à regarder aussi d'autres systèmes de templates comme sur les sites jade-lang.com ou haml.info qui proposent une toute autre façon de créer ses pages web !

Aller plus loin : Connect et les middlewares

Nous venons de voir deux fonctionnalités essentielles d'**Express** :

- Les **routes** : elles permettent de gérer efficacement les **URLs**.
- Les **vues** : elles permettent un accès aux systèmes de templates comme **EJS**.

Tout ceci est déjà très utile et l'on pourrait être tenté de s'arrêter là, mais ce serait passer à côté du cœur d'**Express**. Je vous propose d'aller plus loin en découvrant ensemble comment fonctionnent les entrailles d'**Express**.

Express, connect et les middlewares

Express est un **framework** construit au-dessus d'un autre module appelé **Connect**. En fait, **Express** n'est rien d'autre que le **module Connect** auquel on a ajouté certaines fonctionnalités comme les **routes** et les **vues**.



D'accord, mais qu'est-ce que **Connect** dans ce cas ?

C'est un **module** qui permet à d'autres modules appelés **middlewares** de communiquer entre eux, d'où son nom « **Connect** ». Il étend la fonctionnalité `http.createServer()` fournie de base par **Node.js** et offre donc de nouvelles fonctionnalités.

Connect est fourni avec plus de dix-huit **middlewares** de base, et les développeurs peuvent bien entendu en proposer d'autres *via* **NPM**. Les **middlewares** livrés avec **Connect** fournissent chacun des micro-fonctionnalités. Il y a par exemple :

- **Compress** : permet la **compression gzip** de la page pour un envoi plus rapide au navigateur.
- **CookieParser** : permet de manipuler les **cookies**.
- **Session** : permet de gérer des **informations de session** (durant la visite d'un visiteur).
- **Static** : permet de renvoyer des **fichiers statiques** contenus dans un dossier (images, fichiers à télécharger, etc.)
- **BasicAuth** : permet de demander une **authentification HTTP Basic**, c'est-à-dire une saisie d'un login et d'un mot de passe avant d'accéder à une section du site.
- **Csrf** : fournit une protection contre les **failles CSRF**.
- Etc.

Tous ces **middlewares** offrent vraiment des micro-fonctionnalités. Il y en a des tous petits comme « **favicon** » par exemple, qui se contente de gérer la **favicon** de votre site (l'icône qui représente votre site).

Grâce à **Connect**, ces **middlewares** sont interconnectés et peuvent communiquer entre eux. **Express** ne fait qu'ajouter les routes et les vues par-dessus l'ensemble (voir schéma à la figure 6.3).

Le principe de **Connect** est que tous ces **middlewares** communiquent entre eux en se renvoyant jusqu'à quatre paramètres :

- **err** : les erreurs.
- **req** : la requête du visiteur.
- **res** : la réponse à renvoyer (la page HTML et les informations d'en-tête).
- **next** : un callback vers la prochaine fonction à appeler.

Si je devais résumer les potentialités de **Connect** dans un schéma, cela donnerait la figure 6.4.

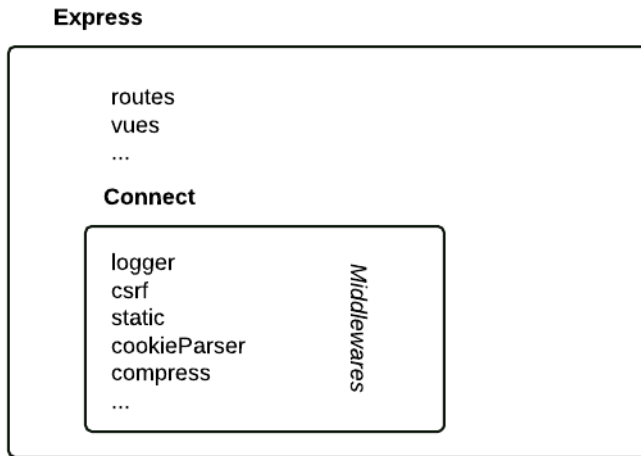


FIGURE 6.3 – Les fonctionnalités offertes par Express viennent en grande partie de Connect

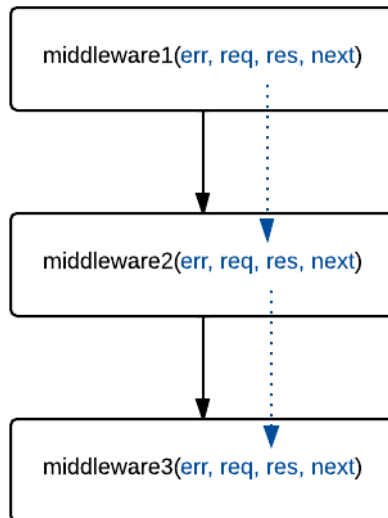


FIGURE 6.4 – Les middlewares communiquent les paramètres grâce à Connect



N'hésitez pas à lire la doc. de Connect6 puis à lire celle d'Express que vous trouverez dans le code web suivant. Vous y trouverez toutes les informations dont vous avez besoin pour utiliser les **middlewares**.



La doc. d'Express
Code web : 822156

Utiliser les fonctionnalités de Connect au sein d'Express

Si je vous raconte tout cela, c'est parce que vous en avez besoin pour tirer pleinement partie d'**Express**. En effet, vous pouvez appeler les différents **modules de Connect** avec la méthode `app.use()`. Par exemple, vous pouvez faire :

```
1 | app.use(express.logger()) // Active le middleware de logging
2 | .use(express.static(__dirname + '/public')) // Indique que le
   |     dossier /public contient des fichiers statiques
3 | .use(express.favicon(__dirname + '/public/favicon.ico')) //
   |     Active la favicon indiquée
4 | .use(function(req, res){ // Répond enfin
5 |     res.send('Hello');
6 | });
```



L'ordre d'appel des **middlewares** est extrêmement important. Par exemple, on commence par activer le logger. Si on le faisait en dernier, on ne loggerait rien ! Quand vous faites appel aux **middlewares**, réfléchissez donc à l'ordre, car il peut impacter fortement le fonctionnement de votre application.

Comme vous le voyez, j'ai fait appel aux **middlewares** « **logger** », « **static** » et « **favicon** » dans le cas présent. Chaque **middleware** va se renvoyer des données (la requête, la réponse, la fonction suivante à appeler, etc.). Chacun a un rôle très précis. Pour savoir les utiliser, il suffit de lire la documentation indiquée dans le code web suivant :



La doc. de Connect
Code web : 235240

Je pourrais m'étendre et présenter les **middlewares** un à un, mais ce serait long et fastidieux (pour moi comme pour vous). Je ne souhaite pas rentrer plus dans le détail, mais j'estime que vous savez l'essentiel, ce qui était le plus compliqué à comprendre.

Je résume : **Express** utilise **Connect** qui propose un ensemble de **middlewares** qui communiquent entre eux. Appelez ces **middlewares** pour utiliser leurs fonctionnalités et faites attention à l'ordre d'appel qui est important (on n'active pas un logger à la fin des opérations!).

Vous êtes prêts à affronter le monde terrible et passionnant d'**Express**, un **framework** incontournable qui évolue rapidement. Bravo et bonne lecture de la documentation maintenant !

En résumé

- **Express.js** est un **micro-framework** pour **Node.js**. Les tâches les plus courantes sont ainsi grandement facilitées.
- **Express** permet en particulier de gérer plus simplement les **routes** (les différentes **URL** acceptées par votre application).
- **Express** permet de faire le pont avec des **moteurs de templates** (comme **EJS**). Ceux-ci nous permettent de séparer le **code métier (backend)** du **code HTML (frontend)**. Finis les appels interminables à la méthode **write()** !
- **Express** est basé sur un autre module appelé **Connect**. Celui-ci permet d'appeler des mini-couches applicatives appelées « **middlewares** » qui offrent des fonctionnalités comme les sessions, la compression gzip des pages, la manipulation des cookies, etc.

Chapitre 7

TP : la todo list

Difficulté : 

Je crois qu'il est grand temps de passer à la pratique ! Nous avons fait le tour d'un bon nombre de fonctionnalités de base de **Node.js** et nous avons même appris à nous servir du **micro-framework Express**. Pourtant, on a vraiment le sentiment d'avoir compris que lorsque l'on a pratiqué et réalisé une première véritable application. C'est ce que je vous propose de faire dans ce TP.

Nous allons réaliser ici une « to do list » (une liste de tâches). Le visiteur pourra simplement ajouter et supprimer des tâches. Nous ferons donc quelque chose de très simple pour commencer. Vous pourrez ensuite l'améliorer comme bon vous semble !



Besoin d'aide ?

Avant toute chose, je vais vous présenter la page que nous devons créer ; il s'agit de la figure 7.1.

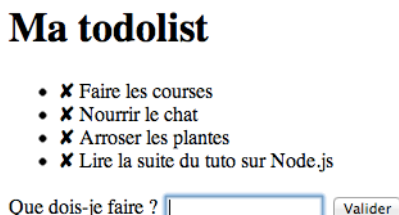


FIGURE 7.1 – L'application que nous allons créer

- On peut ajouter des éléments à la to do list *via* le formulaire.
- On peut supprimer des éléments en cliquant sur les croix dans la liste.
- La liste est stockée dans la session du visiteur. Si quelqu'un d'autre se connecte, il aura sa propre liste.

Les instructions sont simples et le code final ne sera pas très long. En revanche, si c'est votre première application un peu complète avec **Node.js** vous allez sûrement tâtonner et avancer doucement. Ne prenez pas peur, c'est parfaitement normal ! Persévérez et, si vraiment vous avez besoin d'aide, lisez la section « Un peu d'aide » avant de comparer votre travail avec le mien.



N'oubliez pas d'ouvrir les documents d'Express6 et de Connect6 proposé dans les code web correspondants. Je n'aurais jamais pu faire ce TP moi-même sans avoir recours à la documentation !

Vous avez toutes les infos, à vous de jouer !

Les modules et le package.json

Respirons un grand coup et prenons les choses dans l'ordre. De quoi a-t-on besoin ?

- **Express** : sans le **framework Express** la mission sera vraiment très difficile. Maintenant que nous savons utiliser ce **framework**, ce serait dommage de s'en priver.
- **EJS** (ou un autre système de template) : cela nous permettra de construire facilement la **page HTML** qui affiche la liste des tâches et le formulaire.

A priori, ces modules suffisent. Une première bonne pratique serait de se créer un fichier `package.json` dans le dossier de notre projet :

```
1 | {  
2 |   "name": "ma-todolist",
```

```

3 |     "version": "0.1.0",
4 |     "dependencies": {
5 |       "express": "~3.2.1",
6 |       "ejs": "~0.8.3"
7 |     },
8 |     "author": "Mateo21 <mateo21@email.com>",
9 |     "description": "Un gestionnaire de todolist ultra basique"
10|   }

```

Vous pouvez évidemment remplacer le nom, l'auteur et la description comme bon vous semble.

Pour les numéros de version d'**Express** et d'**EJS**, je me suis basé sur les versions disponibles au moment de la rédaction de ce cours. Comme l'univers de **Node.js** avance vite, vous aurez sûrement des versions plus récentes. Le tilde (`~`) permet d'autoriser les futurs **patches** de ces modules, mais pas les nouvelles versions mineures ou majeures, ce qui nous garantit que leur **API** ne changera pas, et donc que notre code continuera à fonctionner même avec ces mises à jour.

Maintenant que vous avez votre `package.json`, installez les **dépendances** avec un simple :

```
npm install
```

Nous sommes prêts à travailler !

Les routes

Je vous avais dit que bien définir ses **routes** était important lorsque l'on construit une application web. Si vous hésitez et ne savez pas par où commencer, je vous invite à lister les **routes** de votre application. Que doit-elle faire ?

- Lister les tâches.
- Ajouter une tâche.
- Supprimer une tâche.

A priori, on peut associer une **route** à chacune de ces fonctionnalités :

- `/todo` : liste les tâches.
- `/todo/ajouter` : ajoute une tâche.
- `/todo/supprimer/:id` : supprime la tâche n° id.

Vous allez donc écrire les routes comme ceci dans votre fichier `app.js` :

```

1 | .get('/todo', function(req, res) {
2 |
3 | });
4 |
5 | // ...

```

En revanche, il faudrait que nous nous penchions un instant sur la question du **formulaire**. En général, les **formulaires** envoient les données avec la **méthode POST** et

non la **méthode GET**. L'ajout d'une tâche va donc se faire sur la route `/todo/ajouter`, mais avec la **méthode POST**. C'est pour cela qu'au lieu de faire appel à `.get()`, nous devrions faire appel à `.post()` pour cette route :

```
1 | .post('/todo/ajouter/', function(req, res) {  
2 |  
3 | })
```

Bien chaîner les appels aux middlewares

Nous savons que nous aurons besoin d'un **système de sessions**. La documentation de **Connect** nous indique comment utiliser les sessions dans le code web suivant. On apprend que les **sessions** utilisent les **cookies** et qu'il faut donc activer les **cookies** *avant* d'activer les **sessions**. Voilà encore un exemple de l'importance de *l'ordre d'appel* des **middlewares** !

▷ Comment utiliser les sessions
Code web : 527765

Autre chose : nous aurons besoin de récupérer les données du formulaire dans `/todo/ajouter`. Nous avons appris à récupérer des paramètres depuis l'**URL** mais pas depuis les **formulaires**. En fait, c'est vraiment simple : il vous suffit d'inclure le middleware que vous trouverez sur le site proposé dans le code web suivant. Vous aurez ensuite accès à `req.body.nomDuChamp`.

▷ bodyParser
Code web : 267545

Du coup, le début de notre **code JavaScript** devrait ressembler à quelque chose comme ça :

```
1 | var express = require('express');  
2 |  
3 | var app = express();  
4 |  
5 | /* On utilise les cookies, les sessions et les formulaires */  
6 | app.use(express.cookieParser())  
7 | .use(express.session({secret: 'todotopsecret'}))  
8 | .use(express.bodyParser())  
9 |  
10 | /* Gestion des routes en-dessous  
11 | .... */
```

Le paramètre **secret** envoyé au **module de session** est obligatoire : il permet de sécuriser les **cookies de session**. Envoyez la valeur de votre choix. Notez que vous pouvez envoyer d'autres options, comme la durée de vie du **cookie de session** (par défaut, la session durera tant que le navigateur restera ouvert). Consultez la liste dans ce code web7.

Allez, je vous en ai trop dit déjà, à vous de coder !

Correction

Allez, c'est l'heure de la correction ! Ne lisez cette section que si vous avez réussi ou que toutes vos tentatives ont échoué et que vous êtes désespérés.

Le code n'est pas si difficile à lire au final. Il n'est pas très long non plus, mais il fallait bien réfléchir pour trouver quoi écrire exactement.

La solution

Voici le **code JavaScript** que j'ai produit dans le fichier principal `app.js`. Notez bien que c'est *une* façon de faire et que vous n'êtes pas obligés d'avoir produit un code strictement identique au mien !

```
1 | var express = require('express');
2 |
3 | var app = express();
4 |
5 | /* On utilise les cookies, les sessions et les formulaires */
6 | app.use(express.cookieParser())
7 | .use(express.session({secret: 'todotopsecret'}))
8 | .use(express.bodyParser())
9 |
10 | /* S'il n'y a pas de todolist dans la session,
11 | on en crée une vide sous forme d'array avant la suite */
12 | .use(function(req, res, next){
13 |     if (typeof(req.session.todolist) == 'undefined') {
14 |         req.session.todolist = [];
15 |     }
16 |     next();
17 | })
18 |
19 | /* On affiche la todolist et le formulaire */
20 | .get('/todo', function(req, res) {
21 |     res.render('todo.ejs', {todolist: req.session.todolist});
22 | })
23 |
24 | /* On ajoute un élément à la todolist */
25 | .post('/todo/ajouter/', function(req, res) {
26 |     if (req.body.newtodo != '') {
27 |         req.session.todolist.push(req.body.newtodo);
28 |     }
29 |     res.redirect('/todo');
30 | })
31 |
32 | /* Supprime un élément de la todolist */
33 | .get('/todo/supprimer/:id', function(req, res) {
34 |     if (req.params.id != '') {
35 |         req.session.todolist.splice(req.params.id, 1);
36 |     }
37 | })
```

```
37     res.redirect('/todo');
38 })
39
40 /* On redirige vers la todolist si la page demandée n'est pas
   trouvée */
41 .use(function(req, res, next){
42     res.redirect('/todo');
43 })
44
45 .listen(8080);
```

Beau chaînage de middlewares n'est-ce pas? :-°

Il y a aussi un **fichier de template** qui va avec. Le fichier `todo.ejs` :

```
1  <!DOCTYPE html>
2
3  <html>
4      <head>
5          <title>Ma todolist</title>
6          <style>
7              a {text-decoration: none; color: black;}
8          </style>
9      </head>
10
11     <body>
12         <h1>Ma todolist</h1>
13
14         <ul>
15             <% todolist.forEach(function(todo, index) { \%>
16                 <li><a href="/todo/supprimer/<%= index \%>">X</a>
17                     <%= todo \%></li>
18             <% }>; \%>
19         </ul>
20
21         <form action="/todo/ajouter/" method="post">
22             <p>
23                 <label for="newtodo">Que dois-je faire ?</label>
24                 <input type="text" name="newtodo" id="newtodo"
25                     autofocus />
26                 <input type="submit" />
27             </p>
28         </form>
29     </body>
30 </html>
```

Et avec ça, l'indispensable **package.json** qui permet d'installer les **dépendances** avec un simple `npm install` :

```
1  {
2      "name": "ma-todolist",
```

```

3 |     "version": "0.1.0",
4 |     "dependencies": {
5 |       "express": "~3.2.1",
6 |       "ejs": "~0.8.3"
7 |     },
8 |     "author": "Mateo21 <mateo21@email.com>",
9 |     "description": "Un questionnaire de todolist ultra basique"
10| }

```

Les explications

Mon code est déjà assez commenté, cela devrait vous permettre de vous y retrouver.

Vous remarquerez que je me suis permis de rediriger le visiteur vers la liste (`/todo`) après un ajout ou une suppression d'élément, avec `res.redirect('/todo')`.

La liste des tâches est stockée dans un **array** (**tableau**). C'est d'ailleurs lui qui est la cause de la petite subtilité de ce programme : comme **JavaScript** n'apprécie pas que l'on essaie de parcourir des **arrays** qui n'existent pas, j'ai créé un **middleware** qui crée un **array** vide si le visiteur n'a pas de to do list (parce qu'il vient de commencer sa session par exemple). C'est le rôle de ce code :

```

1 | .use(function(req, res, next){
2 |   if (typeof(req.session.todolist) == 'undefined') {
3 |     req.session.todolist = [];
4 |   }
5 |   next();
6 | })

```

Cette **fonction middleware** reçoit la requête, la réponse et la prochaine fonction à exécuter. J'ai donc écrit moi-même un **middleware** à l'instar de `cookieParser()`, `session()` et `bodyParser()`. Son rôle est très simple : il vérifie s'il y a une to do list dans la session et, si ce n'est pas le cas, il crée un **array** vide (d'où les `[]`). Cela nous évite beaucoup d'erreurs par la suite.

Pourquoi avoir créé un **middleware** ? Parce que c'est le seul moyen à ma disposition pour exécuter des fonctionnalités *avant* le chargement de n'importe quelle page. Pour que le **middleware** « passe le bébé à son voisin », je dois finir impérativement par un appel à `next()` (la fonction suivante). Dans le cas présent, `next()` fait référence à `.get('/todo', function() {})`.

À part ça, rien de vraiment particulier. J'ajoute des éléments à la fin du tableau avec `.push()` et je retire des éléments avec `.splice()`, mais ça c'est du **JavaScript** de base.

Du côté du **template**, qu'est-ce qu'il y a de particulier ? Pas grand chose, si ce n'est que je parcours la to do list avec un **forEach** (là encore, c'est du **JavaScript**) :

```

1 | <% todolist.forEach(function(todo, index) { %>
2 |   <li><a href="/todo/supprimer/<%= index %>">X</a> <%=
   |     todo %></li>

```

3 | <\% }); \%>



La croix pour supprimer un élément est un simple **caractère Unicode** (vive Unicode). Vous n'imaginez pas le nombre de **symboles Unicode** qu'on peut trouver en faisant une recherche Google du type « unicode cross », « unicode smiley », « unicode plane », « unicode arrow », etc.

Télécharger le projet

Je vous ai donné tout le code du projet, mais si vous insistez pour télécharger le tout dans un fichier .zip, aller sur le code web suivant. N'oubliez pas de faire un `npm install` pour installer les dépendances avant de l'exécuter !

▷

Fichier .zip
Code web : 543618

Allez plus loin !

Ma petite to do list est très basique. Vous pouvez lui ajouter de nombreuses fonctionnalités :

- Modification des noms des tâches.
- Réagencement des tâches entre elles.
- Exportation CSV.
- Attribution d'une priorité et d'une date limite.
- Persistance de la to do list (stockage dans une base de données ou une base NoSQL).
- Partage d'une to do list entre plusieurs personnes.
- Synchronisation de la to do list en temps réel entre les personnes sans avoir besoin de recharger la page.

Certaines de ces fonctionnalités sont plus faciles à réaliser que d'autres. Si vous souhaitez en créer de nouvelles, il vous faudra découvrir et utiliser de nouveaux modules.

Vous avez de quoi vous amuser pendant un bon petit moment, bon courage !

Troisième partie

La communication temps réel avec socket.io

Chapitre 8

socket.io : passez au temps réel !

Difficulté : 

Socket.io est l'une des bibliothèques les plus prisées par ceux qui développent avec **Node.js**. Pourquoi ? Parce qu'elle permet de faire très simplement de la **communication synchrone** dans votre application, c'est-à-dire de la communication en temps réel ! Vous ne voyez pas ce que ça signifie ? Laissez-moi vous le dire autrement : **socket.io** vous permet par exemple de mettre en place très facilement un Chat sur votre site !

Les possibilités que vous offre **socket.io** sont en réalité immenses et vont bien au-delà du Chat : tout ce qui nécessite une communication immédiate entre les visiteurs de votre site peut en bénéficier. Cela peut être par exemple une brique de base pour mettre en place un jeu où on voit en direct les personnages évoluer dans le navigateur, le tout *sans avoir à recharger la page* !

Cela donne envie, non !



Que fait socket.io ?

Avant de commencer à coder, je voudrais vous présenter rapidement le principe de **socket.io**. C'est une bibliothèque qui nous simplifie beaucoup les choses, mais on pourrait croire à tort que c'est « de la magie ». Or, **socket.io** se base sur plusieurs techniques différentes qui permettent la communication en temps réel (et qui pour certaines existent depuis des années !). La plus connue d'entre elles, et la plus récente, est **WebSocket**.



WebSocket ? Ce n'est pas une des nouveautés de HTML5 ?

C'est une nouveauté récente apparue plus ou moins en même temps que **HTML5**, mais ce n'est pas du HTML : c'est une **API JavaScript**. **WebSocket** est une fonctionnalité supportée par l'ensemble des navigateurs récents. Elle permet un **échange bilatéral synchrone** entre le client et le serveur.

Comment ça je parle chinois ?! Reprenons les bases. Habituellement, sur le web, la communication est **asynchrone**. Le web a toujours été conçu comme cela : le client demande et le serveur répond (voir figure 8.1).

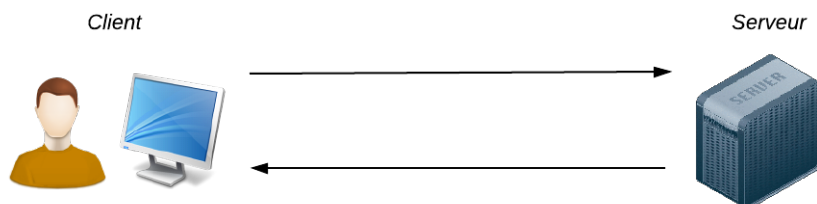


FIGURE 8.1 – Habituellement, la communication est asynchrone : le client demande, le serveur répond

C'était suffisant aux débuts du web, mais cela est devenu trop limitant ces derniers temps. Nous avons besoin d'une communication plus réactive et immédiate. Dans ce schéma par exemple, le serveur ne peut pas décider de lui-même d'envoyer quelque chose au client (par exemple pour l'avertir : « eh il y a un nouveau message ! »). Il faut que le client recharge la page ou fasse une action pour solliciter le serveur car celui-ci n'a pas le droit de s'adresser au client tout seul.

WebSocket est une nouveauté du web qui permet de laisser une sorte de « tuyau » de communication ouvert entre le client et le serveur. Le navigateur et le serveur restent connectés entre eux et peuvent s'échanger des messages, dans un sens comme dans l'autre, dans ce tuyau. Désormais, le serveur peut donc lui-même décider d'envoyer un message au client comme un grand (voir figure 8.2) !

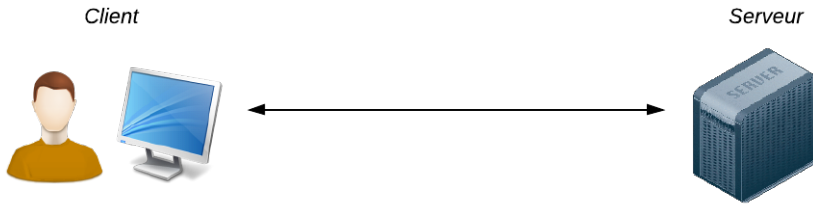


FIGURE 8.2 – Avec les WebSocket, la communication est synchrone : un tuyau de communication reste ouvert entre client et serveur



Ne confondez pas **WebSocket** et **AJAX** ! **AJAX** permet effectivement au client et au serveur d'échanger des informations sans recharger la page. Mais en **AJAX**, c'est toujours le client qui demande et le serveur qui répond. Le serveur ne peut pas décider de lui-même d'envoyer des informations au client. Avec **WebSocket**, cela devient possible !

Socket.io nous permet d'utiliser les **WebSockets** très facilement. Et, comme tous les navigateurs ne gèrent pas **WebSocket**, il est capable d'utiliser d'autres techniques de communication synchrones si elles sont gérées par le navigateur du client. Par exemple, rendez-vous sur le lien du site indiqué dans le code web suivant :

```
Browser support de So-  
cket.io/  
Code web : 602570
```

On voit que **socket.io** détermine pour chaque client quelle est la méthode de communication temps réel la plus adaptée pour le client :

- **WebSocket.**
- **Adobe Flash Socket.**
- **AJAX long polling.**
- **AJAX multipart streaming.**
- **Forever Iframe.**
- **JSONP Polling.**



Par exemple, si le navigateur ne supporte pas **WebSocket** mais que **Flash** est installé, **socket.io** passera par **Flash** pour faire de la communication temps réel. Sinon, il peut utiliser d'autres techniques comme l'**AJAX Long Polling** (le client demande en continu au serveur s'il a des nouveautés pour lui (pas le plus « propre » ni le plus efficace mais ça marche) ou encore la « **Forever Iframe** » qui se base sur une **iframe** invisible qui se charge progressivement pour récupérer les nouveautés du serveur. La bonne nouvelle, c'est que vous n'avez pas besoin de connaître le détail du fonctionnement de ces techniques. Par contre, je pense que c'est bien au moins de connaître leur nom et de savoir qu'elles existent.

Grâce à toutes ces différentes techniques de communication, **socket.io** supporte un très grand nombre de navigateurs, même anciens :

- **Internet Explorer 5.5+** (oui oui, vous avez bien lu!).
- **Safari 3+**.
- **Google Chrome 4+**.
- **Firefox 3+**.
- **Opera 10.61+**.
- **Safari sur iPhone et iPad**.
- Le navigateur **Android**.

Maintenant que nous en savons un petit peu plus sur le fonctionnement de **socket.io**, si nous commençons à l'utiliser ?

Émettre et recevoir des messages avec socket.io

Parlons peu, mais parlons bien. Comment utiliser **socket.io** ?

Installer socket.io

La première étape, aussi évidente soit-elle, est d'installer **socket.io**. Ne riez pas, la première fois que j'ai voulu l'utiliser j'ai bien perdu quinze minutes avant de comprendre que j'avais oublié de faire un simple :

```
npm install socket.io
```

Je vous ai fait donc gagner 15 minutes de votre vie ! Ne me remerciez pas c'est tout naturel.

Premier code : un client se connecte

Lorsque l'on utilise **socket.io**, on doit toujours s'occuper de deux fichiers en même temps :

- Le **fichier serveur** (ex : `app.js`) : c'est lui qui centralise et gère les connexions des différents clients connectés au site.
- Le **fichier client** (ex : `index.html`) : c'est lui qui se connecte au serveur et qui affiche les résultats dans le navigateur.

Le serveur (`app.js`)

J'ai volontairement séparé le code du serveur en deux parties : au début, on charge le serveur comme d'habitude (et on récupère et renvoie le contenu de la page `index.html`) ; ensuite, on charge `socket.io` et on gère les événements de `socket.io`.

```
1 | var http = require('http');
2 | var fs = require('fs');
3 |
4 | // Chargement du fichier index.html affiché au client
5 | var server = http.createServer(function(req, res) {
6 |     fs.readFile('./index.html', 'utf-8', function(error,
7 |         content) {
8 |         res.writeHead(200, {"Content-Type": "text/html"});
9 |         res.end(content);
10 |     });
11 | });
12 |
13 | // Chargement de socket.io
14 | var io = require('socket.io').listen(server);
15 |
16 | // Quand on client se connecte, on le note dans la console
17 | io.sockets.on('connection', function (socket) {
18 |     console.log('Un client est connecté !');
19 | });
20 |
21 | server.listen(8080);
```

Ce code fait deux choses :

- Il renvoie le fichier `index.html` quand un client demande à charger la page dans son navigateur.
- Il se prépare à recevoir des requêtes *via* `socket.io`. Ici, on s'attend à recevoir un seul type de message : la connexion. Lorsqu'on se connecte *via* `socket.io`, on logge ici l'information dans la console.

Imaginez-vous en tant que visiteur. Vous ouvrez votre navigateur à l'adresse où se trouve votre application (`http://localhost:8080` ici). On vous envoie le fichier `index.html`, la page se charge. Dans ce fichier, que nous allons voir juste après, un **code JavaScript** se connecte au serveur, cette fois pas en `http` mais *via* `socket.io` (donc *via* les **WebSockets** en général). Le client effectue donc deux types de connexion :

- Une connexion « classique » au serveur en `HTTP` pour charger la page `index.html`.
- Une connexion « temps réel » pour ouvrir un tunnel *via* les **WebSockets** grâce à `socket.io`.

Le client (index.html)

Intéressons-nous maintenant au client. Le fichier `index.html` est envoyé par le serveur **Node.js**. C'est un fichier HTML tout ce qu'il y a de plus classique, si ce n'est qu'il contient un peu de code **JavaScript** qui permettra ensuite de communiquer avec le serveur en temps réel *via* **socket.io** :

```

1 | <!DOCTYPE html>
2 | <html>
3 |   <head>
4 |     <meta charset="utf-8" />
5 |     <title>Socket.io</title>
6 |   </head>
7 |
8 |   <body>
9 |     <h1>Communication avec socket.io !</h1>
10 |
11 |     <script src="/socket.io/socket.io.js"></script>
12 |     <script>
13 |       var socket = io.connect('http://localhost:8080');
14 |     </script>
15 |   </body>
16 | </html>

```



J'ai placé le **code JavaScript** à la fin du code HTML volontairement. Bien sûr, on pourrait le mettre dans la balise `<head>` comme beaucoup le font, mais le placer à la fin du code HTML permet d'éviter que le chargement du **JavaScript** ne bloque le chargement de la page HTML. Au final, cela donne l'impression d'une page web qui se charge plus rapidement.

Dans un premier temps, on fait récupérer au client le fichier `socket.io.js`. Celui-ci est automatiquement fourni par le serveur **Node.js** *via* le **module socket.io** (le chemin vers le fichier n'est donc pas choisi au hasard) :

```

1 | <script src="/socket.io/socket.io.js"></script>

```

Le code qu'il contient permet de gérer la communication avec le serveur du côté du client, soit avec les **WebSockets**, soit avec l'une des autres méthodes dont je vous ai parlé si le navigateur ne les supporte pas.

Ensuite, nous pouvons effectuer des actions du côté du client pour communiquer avec le serveur. Pour le moment, j'ai fait quelque chose de très simple : je me suis contenté de me connecter au serveur. Celui-ci se trouve sur ma machine, d'où l'adresse `http://localhost:8080`. Évidemment, sur le web, il faudra adapter ce chemin pour indiquer l'adresse de votre site (ex : `http://monsite.com`).

```

1 | var socket = io.connect('http://localhost:8080');

```

Testons le code !

Il nous suffit de lancer l'application :

```
node app.js
```

On peut alors se rendre avec notre navigateur à cette adresse **Node.js** :
`http://localhost:8080`.

Une page basique va se charger. Votre ordinateur va ensuite ouvrir une connexion avec **socket.io** et le serveur devrait afficher des informations de débogage dans la console :

```
$ node app.js
info - socket.io started
debug - client authorized
info - handshake authorized Z2E7aqIv0PPqv_XBn421
debug - setting request GET /socket.io/1/websocket/
      Z2E7aqIv0PPqv_XBn421
debug - set heartbeat interval for client
      Z2E7aqIv0PPqv_XBn421
debug - client authorized for
debug - websocket writing 1::
Un client est connecté !
```

Super ! Cela veut dire que notre code fonctionne.

Pour le moment il ne fait rien de bien extraordinaire, mais nous avons les bases. Ça va être le moment de s'amuser à échanger des messages avec le serveur !

Envoi et réception de messages

Maintenant que le client est connecté, on peut échanger des messages entre le client et le serveur. Il y a deux cas de figure :

- Le serveur veut envoyer un message au client.
- Le client veut envoyer un message au serveur.

Le serveur veut envoyer un message au client

Je propose que le serveur envoie un message au client lorsqu'il vient de se connecter, pour lui confirmer que la connexion a bien fonctionné. Rajoutez ceci au fichier `app.js` :

```
1 | io.sockets.on('connection', function (socket) {
2 | |     socket.emit('message', 'Vous êtes bien connecté !');
3 | | });
```

Lorsque l'on détecte une connexion, on émet un message au client avec `socket.emit()`. La fonction prend deux paramètres :

- Le *type de message* qu'on veut transmettre. Ici, mon message est de type `message` (je ne suis pas très original, je sais). Cela vous permettra de distinguer les différents types de message. Par exemple, dans un jeu, on pourrait envoyer des messages de type `deplacement_joueur`, `attaque_joueur`.
- Le *contenu du message*. Là vous pouvez transmettre ce que vous voulez.

Si vous voulez envoyer plusieurs données différentes avec votre message, regroupez-les sous forme d'objet comme ceci par exemple :

```
1 | socket.emit('message', { content: 'Vous êtes bien connecté !',
   |   importance: '1' });
```

Du côté du fichier `index.html` (le client), on va écouter l'arrivée de messages de type `message` :

```
1 | <script>
2 |     var socket = io.connect('http://localhost:8080');
3 |     socket.on('message', function(message) {
4 |         alert('Le serveur a un message pour vous : ' + message)
5 |     });
6 | </script>
```

Avec `socket.on()`, on écoute les messages de type `message`. Lorsque des messages arrivent, on appelle la **fonction de callback** qui, ici, affiche simplement une boîte de dialogue.

Essayez, vous verrez que lorsque vous chargez la page `index.html`, une boîte de dialogue s'affiche indiquant que la connexion a réussi (voir figure 8.3) !

Communication avec socket.io !

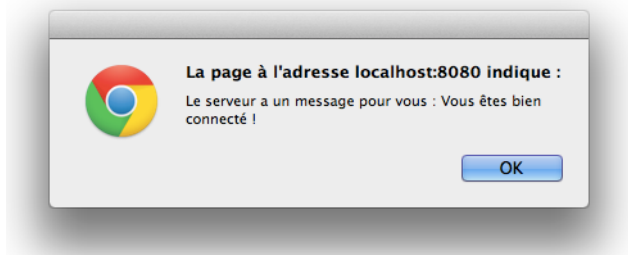


FIGURE 8.3 – Le client affiche le message du serveur dans une boîte de dialogue

Le client veut envoyer un message au serveur

Maintenant, faisons l'inverse. Je vous propose d'ajouter un bouton dans la page web et d'envoyer un message au serveur lorsqu'on clique dessus.

Du côté du client (`index.html`), je vais rajouter un bouton `Embêter le serveur`. Lorsqu'on cliquera dessus, j'émètrai un message au serveur. Voici le code complet :

```

1 | <!DOCTYPE html>
2 | <html>
3 |   <head>
4 |     <meta charset="utf-8" />
5 |     <title>Socket.io</title>
6 |   </head>
7 |
8 |   <body>
9 |     <h1>Communication avec socket.io !</h1>
10 |
11 |     <p><input type="button" value="Embêter le serveur" id="
12 |       poke" /></p>
13 |
14 |     <script src="http://code.jquery.com/jquery-1.10.1.min.
15 |       js"></script>
16 |     <script src="/socket.io/socket.io.js"></script>
17 |     <script>
18 |       var socket = io.connect('http://localhost:8080');
19 |       socket.on('message', function(message) {
20 |         alert('Le serveur a un message pour vous : ' +
21 |           message);
22 |       })
23 |
24 |       $('#poke').click(function () {
25 |         socket.emit('message', 'Salut serveur, ça va ?'
26 |           );
27 |       });
28 |     </script>
29 |   </body>
30 | </html>

```



J'utilise ici **jQuery** pour des raisons pratiques (parce que j'ai l'habitude) pour récupérer l'évènement du clic sur le bouton, mais ce n'est absolument pas obligatoire. On peut faire ça en pur **JavaScript** si on veut.

Finalement, le seul code nouveau et intéressant ici est :

```

1 | $('#poke').click(function () {
2 |   socket.emit('message', 'Salut serveur, ça va ?');
3 | })

```

Il est très simple. Lorsque l'on clique sur le bouton, on envoie un message de type **message** au serveur, assorti d'un contenu.

Si on veut récupérer cela du côté du serveur maintenant, il va nous falloir ajouter l'écoute des messages de type **message** dans la **fonction de callback** de la connexion :

```

1 | io.sockets.on('connection', function (socket) {
2 |   socket.emit('message', 'Vous êtes bien connecté !');
3 |

```

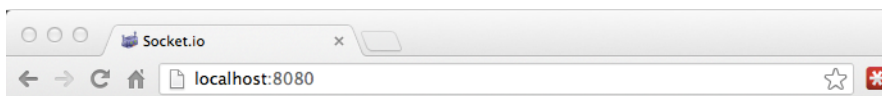
```

4      // Quand le serveur reçoit un signal de type "message" du
      client
5      socket.on('message', function (message) {
6          console.log('Un client me parle ! Il me dit : ' +
            message);
7      });
8  });

```

Lancez le code ! Cliquez sur le bouton **Embêter le serveur** dans la page et regardez dans la console du serveur. Vous devriez voir apparaître la figure 8.4.

Un client me parle ! Il me dit : Salut serveur, ça va ?



Communication avec socket.io !

Embêter le serveur



FIGURE 8.4 – Lorsque le client clique sur le bouton, le serveur réagit immédiatement dans la console

Communiquer avec plusieurs clients

Dans tous nos exemples précédents, on a travaillé avec un serveur et un seul client. Dans la pratique, vous aurez sûrement plusieurs clients connectés à votre application **Node.js** (en tout cas je vous le souhaite!). Pour simuler cela en local, c'est très simple : il suffit d'ouvrir deux onglets, chacun sur votre page `http://localhost:8080`. Le serveur verra deux clients différents connectés.

Lorsque l'on a plusieurs clients, il faut être capable :

- D'envoyer des messages à tout le monde d'un seul coup. On appelle ça les **broadcasts**.
- De se souvenir d'informations sur chaque client (comme son pseudo par exemple). On a besoin pour ça de **variables de session**.

Ô surprise, c'est justement ce que je comptais vous expliquer maintenant !

Envoyer un message à tous les clients (broadcast)

Quand vous faites un `socket.emit()` du côté du serveur, vous envoyez uniquement un message au client avec lequel vous êtes en train de discuter. Mais vous pouvez faire plus fort : vous pouvez envoyer un **broadcast**, c'est-à-dire un message destiné à tous les autres clients (excepté celui qui vient de solliciter le serveur).

Prenons un cas :

1. Le client A envoie un message au serveur.
2. Le serveur l'analyse.
3. Il décide de broadcaster ce message pour l'envoyer aux autres clients connectés : B et C.

La figure 8.5 montre cela.

Imaginez par exemple un Chat. Le client A écrit un message et l'envoie au serveur. Pour que les autres clients voient ce message, il doit le leur broadcaster. Pour cela, rien de plus simple !

```
1 | socket.broadcast.emit('message', 'Message à toutes les unités.  
  | Je répète, message à toutes les unités.');
```

Il suffit de faire un `socket.broadcast.emit()` et le message partira vers tous les autres clients connectés. Ajoutez par exemple un **broadcast** dans `app.js` lors de la connexion d'un client :

```
1 | io.sockets.on('connection', function (socket) {  
2 |     socket.emit('message', 'Vous êtes bien connecté !');  
3 |     socket.broadcast.emit('message', 'Un autre client vient  
  |     de se connecter !');  
4 |  
5 |     socket.on('message', function (message) {
```

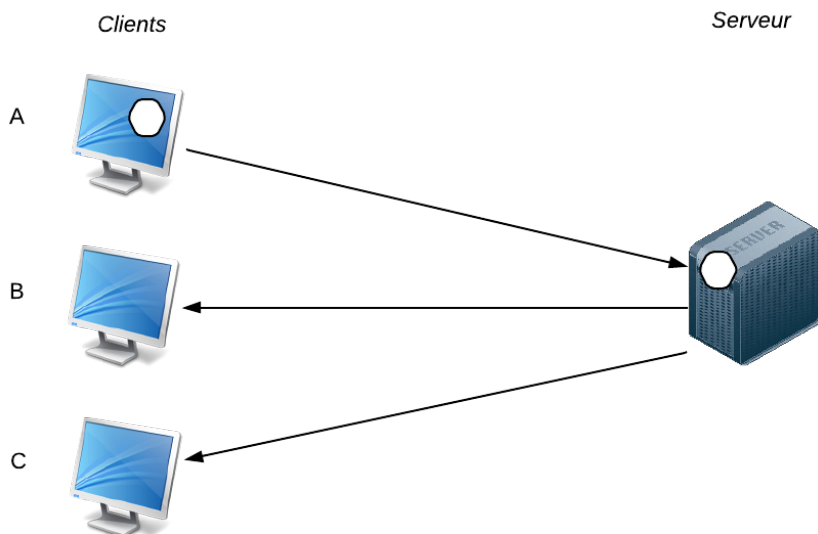


FIGURE 8.5 – Dans un broadcast, le serveur envoie un message à tous les autres clients connectés

```
6 | console.log('Un client me parle ! Il me dit : '
7 |           + message);
8 | });
```

Essayez maintenant d'ouvrir deux onglets (ou plus) sur votre page <http://localhost:8080>. Vous verrez que lorsqu'un nouveau client arrive, les autres pages réagissent instantanément pour dire : « Un autre client vient de se connecter ! »

Les variables de session

Lorsque vous aurez plusieurs clients connectés, vous allez vite vous rendre compte qu'il est délicat de les reconnaître. L'idéal serait de pouvoir mémoriser des informations sur chaque client. Et ça tombe bien, c'est justement à ça que servent les **variables de session** !

Le serveur peut retenir des informations sur chaque client connecté. Comme ça, le client n'aura pas besoin de rappeler qui il est à chaque fois qu'il envoie un message !

Pour stocker une **variable de session** côté serveur, il suffit d'écrire :

```
1 | socket.set('nomVariable', data);
```

Dans cet exemple, on stocke les données (contenues dans la variable `data`) dans une **variable de session** nommée `nomVariable`.

Pour récupérer cette information ensuite, il faudra faire appel à `get()`. Comme avec **Node.js** on adore les **fonctions de callback** (pour ne pas bloquer le serveur pendant qu'il récupère l'information), il va falloir donner une **fonction de callback** qui sera exécutée dès que le serveur aura la variable en main :

```
1 | socket.get('nomVariable', function (error, data) {  
2 |     console.log(data);  
3 | });
```

Ici, on demande à récupérer la variable de session `nomVariable`. Dès que le serveur l'a obtenu, il exécute la fonction située à l'intérieur. Cette dernière peut alors faire ce qu'elle veut avec les données.

Simple non ? Essayons d'imaginer un cas pratique. Lorsqu'un client se connecte, la page web va lui demander son pseudo. Le serveur stockera le pseudo en variable de session pour s'en souvenir lorsque le client cliquera sur Embêter le serveur.

Voyons les modifications que nous devons faire...

La page web (index.html) émet un signal contenant le pseudo

Au chargement de la page web, nous allons demander le pseudo du visiteur. On envoie ce pseudo au serveur *via* un signal de type `petit_nouveau` (je l'ai appelé comme ça pour le différencier des signaux de type `message`). Ce signal contient le pseudo du visiteur :

```
1 | var pseudo = prompt('Quel est votre pseudo ?');  
2 | socket.emit('petit_nouveau', pseudo);
```

Le serveur (app.js) stocke le pseudo

Le serveur doit récupérer ce signal. Nous écoutons alors les signaux de type `petit_nouveau` et, lorsque l'on en reçoit, on sauvegarde le pseudo en **variable de session** :

```
1 | socket.on('petit_nouveau', function(pseudo) {  
2 |     socket.set('pseudo', pseudo);  
3 | });
```

Le serveur (app.js) se rappelle du pseudo quand on lui envoie un message

Maintenant, nous voulons que le serveur se souvienne de nous lorsque nous l'embêtons en cliquant sur Embêter le serveur (ce qui provoque l'envoi d'un signal de type `message`). On va compléter la **fonction de callback** qui est appelée quand le serveur reçoit un message :

```

1 | socket.on('message', function (message) {
2 |     socket.get('pseudo', function (error, pseudo) {
3 |         console.log(pseudo + ' me parle ! Il me dit : ' +
4 |             message);
5 |     });
6 | });

```

Dès que l'on reçoit un `message`, on demande à récupérer la variable de session `pseudo`. Dès que l'on a récupéré la variable de session `pseudo`, on peut logger l'information dans la console pour se souvenir du nom de la personne qui vient d'envoyer le message.

J'avoue que ces **fonctions de callback imbriquées** peuvent un peu donner le tournis, mais vous avez voulu faire du **Node.js**, il faut assumer maintenant.

Tester le code

Essayez d'ouvrir deux fenêtres en donnant un pseudo différent à chaque fois. Cliquez ensuite sur **Embêter le serveur**. Vous verrez dans la console que le pseudo de la personne qui a cliqué sur le bouton apparaît !

J'ai fait un essai chez moi avec deux fenêtres, l'une avec le pseudo « mateo21 » et l'autre avec le pseudo « robert ». À la figure 8.6, vous voyez en bas de la console que celle-ci reconnaît bien qui vient de cliquer !

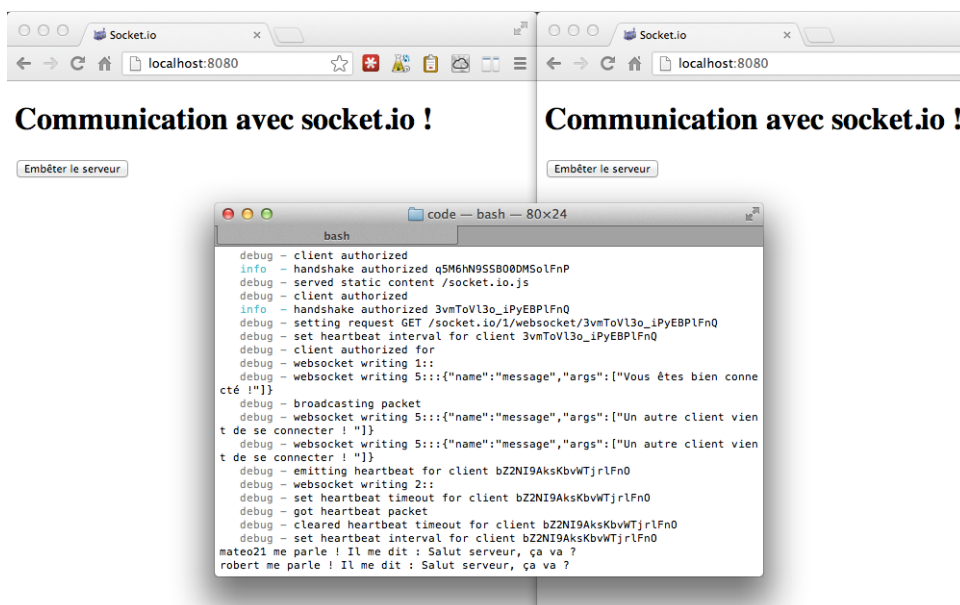


FIGURE 8.6 – Plusieurs clients connectés : le serveur se souvient de leurs noms !

Le code complet

Je vous ai volontairement montré des morceaux de code très courts pour vous expliquer le principe, mais je suis sûr que vous mourez d'envie d'avoir le code complet pour faire vos essais.

Alors allons-y! Voici index.html :

```
1 | <!DOCTYPE html>
2 | <html>
3 |   <head>
4 |     <meta charset="utf-8" />
5 |     <title>Socket.io</title>
6 |   </head>
7 |
8 |   <body>
9 |     <h1>Communication avec socket.io !</h1>
10 |
11 |     <p><input type="button" value="Embêter le serveur" id="
12 |       poke" /></p>
13 |
14 |     <script src="http://code.jquery.com/jquery-1.10.1.min.
15 |       js"></script>
16 |     <script src="/socket.io/socket.io.js"></script>
17 |     <script>
18 |       var socket = io.connect('http://localhost:8080');
19 |
20 |       // On demande le pseudo au visiteur...
21 |       var pseudo = prompt('Quel est votre pseudo ?');
22 |       // Et on l'envoie avec le signal "petit_nouveau" (
23 |         pour le différencier de "message")
24 |       socket.emit('petit_nouveau', pseudo);
25 |
26 |       // On affiche une boîte de dialogue quand le
27 |         serveur nous envoie un "message"
28 |       socket.on('message', function(message) {
29 |         alert('Le serveur a un message pour vous : ' +
30 |           message);
31 |       })
32 |     </script>
33 |   </body>
34 | </html>
```

... et voici l'application serveur app.js :


```
1 | var http = require('http');
2 | var fs = require('fs');
3 |
4 | // Chargement du fichier index.html affiché au client
5 | var server = http.createServer(function(req, res) {
6 |     fs.readFile('./index.html', 'utf-8', function(error,
7 |         content) {
8 |         res.writeHead(200, {"Content-Type": "text/html"});
9 |         res.end(content);
10 |     });
11 | });
12 | // Chargement de socket.io
13 | var io = require('socket.io').listen(server);
14 |
15 | io.sockets.on('connection', function (socket, pseudo) {
16 |     // Quand on client se connecte, on lui envoie un message
17 |     socket.emit('message', 'Vous êtes bien connecté !');
18 |     // On signale aux autres clients qu'il y a un nouveau venu
19 |     socket.broadcast.emit('message', 'Un autre client vient de
20 |         se connecter ! ');
21 |
22 |     // Dès qu'on nous donne un pseudo, on le stocke en variable
23 |     // de session
24 |     socket.on('petit_nouveau', function(pseudo) {
25 |         socket.set('pseudo', pseudo);
26 |     });
27 |
28 |     // Dès qu'on reçoit un "message" (clic sur le bouton), on
29 |     // le note dans la console
30 |     socket.on('message', function (message) {
31 |         // On récupère le pseudo de celui qui a cliqué dans les
32 |         // variables de session
33 |         socket.get('pseudo', function (error, pseudo) {
34 |             console.log(pseudo + ' me parle ! Il me dit : ' +
35 |                 message);
36 |         });
37 |     });
38 | });
39 |
40 | server.listen(8080);
```

J'espère avoir suffisamment commenté ce code pour que vous puissiez vous y retrouver.



Je rappelle que c'est une application très basique pour essayer les fonctionnalités de **socket.io**. Elle ne fait rien d'intéressant ou de passionnant, c'est à vous de vous amuser à la bidouiller pour vous entraîner. Faites-en quelque chose de bien... Enfin quelque chose de plus utile que ce que j'ai fait au moins.

En résumé

- **Socket.io** est un **module de Node.js** qui permet à vos visiteurs de *communiquer en continu* (en temps réel) avec le serveur lorsque la page est chargée.
- **Socket.io** se base notamment sur les **WebSockets**, une sorte de « super AJAX ». Votre application peut à tout moment solliciter le serveur sans recharger la page, et l'inverse est aussi vrai : vos visiteurs peuvent recevoir des messages du serveur à tout moment !
- Le serveur et le client s'envoient des **événements** (avec `socket.emit()`) et écoutent les événements qu'ils se sont envoyés (avec `socket.on()`).
- Pour que l'application fonctionne du côté du client, vous devez charger la bibliothèque de **socket.io** dans la **page HTML** envoyée au visiteur. La bibliothèque est située dans `/socket.io/socket.io.js`.
- Le serveur peut envoyer un message à tous les clients connectés au serveur avec `socket.broadcast.emit()`.
- Le serveur peut sauvegarder les données de session d'un client avec `set()` et les récupérer avec `get()`. Pratique pour que le client n'ait, par exemple, pas besoin de renvoyer son pseudo à chaque événement !

Chapitre 9

TP : le super Chat

Difficulté : 

Alors, **socket.io** vous plaît ? Une fois que l'on a réussi à le prendre en main, ce qui n'est pas vraiment compliqué quand on commence à avoir l'habitude de **Node.js**, il faut avouer qu'un grand nombre de possibilités s'offre tout d'un coup à nous ! Imaginez tout ce que vous pouvez commencer à faire en temps réel ! Un Chat, des jeux, des outils collaboratifs pour le travail, etc.

Parmi l'ensemble de ces possibilités, je pense que le Chat est de loin l'application la plus simple à réaliser tout en restant utile et efficace pour impressionner vos proches. Alors allons-y !



Besoin d'aide ?

Vous êtes perdus ? Vous ne savez pas par où commencer ?

Allons allons, si vous avez bien suivi le chapitre précédent sur **socket.io** cela ne devrait pas trop vous poser de problèmes. Comme je suis (décidément) trop sympa, je vais vous donner quelques indices pour vous aider à avancer.

Avant toute chose, regardez la figure 9.1. Elle vous montre à quoi ressemblera notre Chat.

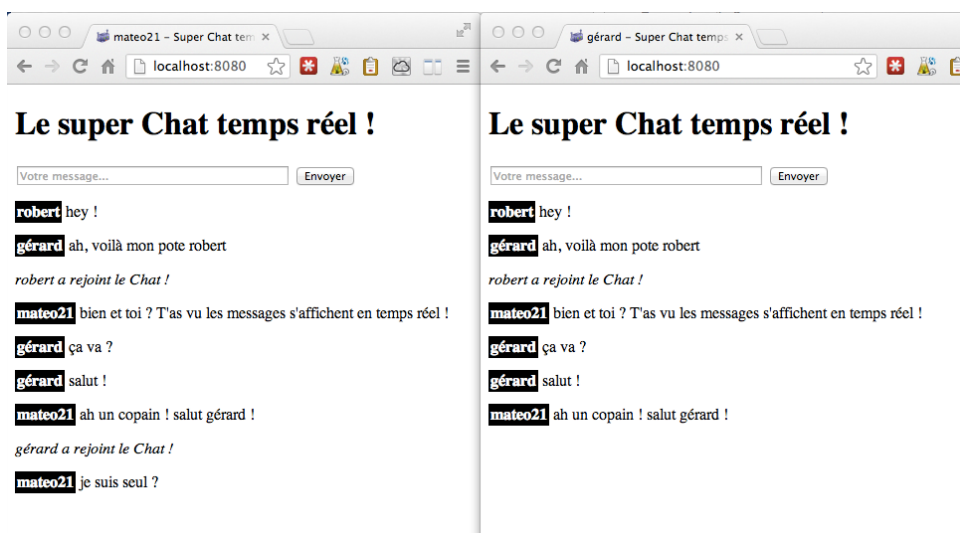


FIGURE 9.1 – Le Super Chat que vous allez devoir réaliser

Ici j'ai volontairement ouvert deux fenêtres. Celle de gauche est connectée sous le pseudo « mateo21 » et celle de droite sous le pseudo « gérard ».

- Je demande le pseudo avec une boîte de dialogue lors de la connexion.
- J'affiche le pseudo de celui qui vient de se connecter à tout le monde dans le Chat (ex : « *gérard a rejoint le Chat !* »).
- Lorsque l'on saisit un message, il est immédiatement affiché dans le navigateur sous le formulaire.

Bref, rien de bien compliqué ! Mais même en vous basant sur les codes du chapitre précédent, il vous faudra travailler un petit peu et réfléchir. Et c'est exactement le plan diabolique que j'ai en tête : vous faire travailler.

Posons-nous maintenant les bonnes questions. De combien de fichiers va-t-on avoir besoin ? De toute évidence, je dirais trois :

- Un fichier **package.json** qui décrit les **dépendances** de votre projet **Node.js**.
- Un fichier **app.js** pour l'**application côté serveur en Node.js**
- Un fichier **index.html** qui contiendra la page web et le code côté client de

gestion du Chat.

Voici un petit peu d'aide pour réaliser chacun de ces fichiers. Si vous vous sentez à l'aise, honnêtement, essayez de vous en passer, c'est encore mieux (et au pire revenez lire cette section si jamais vous coincez).

package.json

À ce stade, vous devriez savoir créer un **package.json** les yeux fermés. Je ne vais pas vous réexpliquer la syntaxe, si vous avez un trou de mémoire je vous laisse retourner à la section 5.

Par contre, côté **modules externes**, de quoi va-t-on avoir besoin ? Voilà ce que je suggère (et que j'utilise dans ma correction) :

- **socket.io** : si vous l'aviez oublié, vous êtes impardonnables.
- **express** : tant qu'à faire, autant utiliser **Express.js** dès le début. On ne s'en est pas servi dans le chapitre de découverte de **socket.io** mais il n'est pas très compliqué à utiliser en combinaison de **socket.io**. Dans le code web suivant, on vous explique comment l'utiliser en combinaison d'**Express.js**.
- **ent** : un tout petit module qui permet de protéger les chaînes de caractères envoyées par les visiteurs pour transformer le HTML en entités. Ce qui permet d'éviter que vos visiteurs s'envoient du code JavaScript dans le Chat !

▷ "How to use" sur le site Socket.io
Code web : 456713

Honnêtement, **Express.js** et **ent** ne sont pas obligatoires pour faire fonctionner le Chat. On peut très bien faire sans, mais j'ai choisi de les utiliser d'une part pour faciliter la maintenance du code (pour **Express.js**) et d'autre part pour la sécurité (pour **ent**).

app.js

Elle devra renvoyer une page web lorsqu'on appellera le serveur. À vous donc de renvoyer un fichier **index.html** à vos visiteurs qui se connectent sur votre site. Avec **Express.js**, la syntaxe est un petit peu différente mais aussi plus courte.

En plus de la page web « classique », votre serveur **Node.js** devra gérer les événements de **socket.io**. *A priori* je dirais que l'on a besoin de n'en gérer que deux :

- **nouveau_client** (vous l'appellez comme vous voulez) : signale qu'un nouveau client vient de se connecter au Chat. Devrait transmettre son pseudo pour pouvoir informer les clients avec des messages comme « robert a rejoint le Chat ! ».
- **message** : signale qu'un nouveau message a été posté. Votre rôle, en tant que serveur, sera tout simplement de le redistribuer aux autres clients connectés avec un petit **broadcast**. Tant qu'à faire, vous récupérerez le pseudo du posteur dans une **variable de session**.

En fin de compte, le fichier `app.js` n'est pas très gros ni très compliqué. Plus je le regarde, plus je le trouve même étonnamment court et simple.

`index.html`

En fait, c'est peut-être le fichier qui demande le plus de travail et qui devrait vous donner (légèrement) du fil à retordre. Vous allez devoir gérer pas mal de **code JavaScript** côté client.

Commencez par structurer une **page HTML5** basique, avec un titre, un formulaire composé d'un champ de texte et d'un bouton, et une `<div>` ou une `<section>` qui contiendra les messages du Chat (par défaut, elle sera vide).

À vous ensuite d'écrire le **code JavaScript** du client. Personnellement je l'ai placé en bas de la page (pour des raisons de performances, c'est une habitude à prendre). J'aurais aussi pu le mettre dans un fichier `.js` externe, mais je ne l'ai pas fait ici en l'occurrence.

Ce code devra :

- Se connecter à **socket.io**.
- Demander le pseudo au visiteur lors du chargement de la page (*via* un `prompt()` en JS, c'est le plus simple que j'aie trouvé) et envoyer un signal `nouveau_client`.
- Gérer la réception de signaux de type `nouveau_client` envoyés par le serveur. Cela signifie qu'un nouveau client vient de se connecter. Affichez son nom dans un message, comme par exemple « Robert a rejoint le Chat ! ».
- Gérer la réception de signaux de type `message` envoyés par le serveur. Cela signifie qu'un autre client vient d'envoyer un message sur le Chat et donc qu'il vous faut l'afficher dans la page.
- Gérer l'envoi du formulaire lorsque le client veut envoyer un message aux autres personnes connectées. Il vous faudra récupérer le message saisi dans le formulaire en JavaScript, émettre un signal de type `message` au serveur pour qu'il le distribue aux autres clients, et aussi insérer ce message dans votre propre page. Eh oui, n'oubliez pas que le **broadcast** du serveur envoie un message à toutes les autres personnes connectées... excepté vous-même. Il faut donc mettre à jour votre propre zone de Chat.

Je vous ai tellement prémâché le travail que j'en ai presque un coup de *blues*... Si vous n'y arrivez pas avec tout ça... essayez plus fort !

Correction

Rien ne va plus, c'est l'heure de la correction !

Alors, comment s'est passée la réalisation du Chat pour vous ? Vous ne devriez pas avoir rencontré trop de problèmes, ce TP était une façon de reformuler un peu plus proprement le chapitre précédent d'introduction à **socket.io**.

Mon projet est constitué de trois fichiers :

- `package.json` : la description du projet avec la liste de ses **dépendances**. Tout projet **Node.js** qui se respecte en a un !
- `app.js` : l'**application Node.js** côté serveur qui gère les interactions avec les différents clients.
- `index.html` : la page web envoyée au client qui contient du **code JavaScript** pour gérer le Chat côté client.

package.json

Notre `package.json` est très simple :

```

1 | {
2 |   "name": "super-chat",
3 |   "version": "0.1.0",
4 |   "dependencies": {
5 |     "express": "~3.3.4",
6 |     "socket.io": "~0.9.16",
7 |     "ent": "~0.1.0"
8 |   },
9 |   "author": "Mateo21 <mateo21@email.com>",
10 |  "description": "Un super Chat temps réel avec socket.io"
11 | }

```

Comme je vous le disais (pour ceux qui ont lu mes astuces d'aide juste avant), j'utilise évidemment **socket.io**, mais aussi **Express.js** (bien que ce ne soit pas obligatoire) et **ent** (pour faire appel à une fonction équivalente à `htmlentities()` de **PHP**, afin de sécuriser les échanges et d'éviter que les clients ne s'envoient des **codes JavaScript** malicieux).

app.js

Relativement court, le code serveur du Chat n'était pas le fichier le plus difficile à écrire de ce TP !

```

1 | var app = require('express')(),
2 |     server = require('http').createServer(app),
3 |     io = require('socket.io').listen(server),
4 |     ent = require('ent'), // Permet de bloquer les caractères
   |       HTML (sécurité équivalente à htmlentities en PHP)
5 |     fs = require('fs');
6 |
7 | // Chargement de la page index.html
8 | app.get('/', function (req, res) {
9 |   res.sendfile(__dirname + '/index.html');
10 | });
11 |
12 | io.sockets.on('connection', function (socket, pseudo) {
13 |   // Dès qu'on nous donne un pseudo, on le stocke en variable
   |     de session et on informe les autres personnes

```



```
14     socket.on('nouveau_client', function(pseudo) {
15         pseudo = ent.encode(pseudo);
16         socket.set('pseudo', pseudo);
17         socket.broadcast.emit('nouveau_client', pseudo);
18     });
19
20     // Dès qu'on reçoit un message, on récupère le pseudo de
21     son auteur et on le transmet aux autres personnes
22     socket.on('message', function (message) {
23         socket.get('pseudo', function (error, pseudo) {
24             message = ent.encode(message);
25             socket.broadcast.emit('message', {pseudo: pseudo,
26                                     message: message});
27         });
28     });
29     server.listen(8080);
```

Le début du fichier ne contient rien d'extraordinaire : quelques appels à des modules, la gestion de l'index (« / ») où on renvoie `index.html`... Non vraiment, plus simple, tu meurs.

En-dessous, nous avons toute la gestion des messages temps réel avec **socket.io**. En fait, on ne gère que deux types de messages différents :

- **nouveau_client** : envoyé par un nouveau client qui vient de charger la page. Il contient son pseudo en paramètre. J'ai choisi de l'encoder (avec `ent.encode()`) par sécurité. Ainsi, si le visiteur met du JavaScript dans son pseudo, on est paré ! Ensuite je n'ai plus qu'à « sauvegarder » ce pseudo dans une **variable de session**.
- **message** : envoyé par un client qui veut transmettre un message aux autres personnes connectées. Dans un premier temps, je récupère son pseudo que j'avais conservé en variable de session. La **fonction de callback** est appelée dès que le serveur a récupéré le pseudo (ça ne prend pas très longtemps en pratique, mais vous savez que **Node.js** déteste les **fonctions bloquantes** !). Dès qu'on a le pseudo, on encode le message (toujours par sécurité) et on le broadcast avec le pseudo. Pour envoyer plusieurs données dans un seul paramètre, je les encapsule dans un **objet JavaScript**, d'où le code `{pseudo: pseudo, message: message}`.

C'est tout ce que le serveur a besoin de faire !

`index.html`

OK, le code du client est « un poil » plus compliqué. Mais alors juste un poil honnêtement. Le voici tout de go, on le commente juste après, promis :

```
1 | <!DOCTYPE html>
2 | <html>
```

```

3    <head>
4        <meta charset="utf-8" />
5        <title>Super Chat temps réel !</title>
6        <style>
7            #zone_chat strong {
8                color: white;
9                background-color: black;
10               padding: 2px;
11            }
12        </style>
13    </head>
14
15    <body>
16        <h1>Le super Chat temps réel !</h1>
17
18        <form action="/" method="post" id="formulaire_chat">
19            <input type="text" name="message" id="message"
20                placeholder="Votre message..." size="50"
21                autofocus />
22            <input type="submit" id="envoi_message" value="
23                Envoyer" />
24        </form>
25
26        <section id="zone_chat">
27
28        </section>
29
30        <script src="http://code.jquery.com/jquery-1.10.1.min.
31            js"></script>
32        <script src="/socket.io/socket.io.js"></script>
33        <script>
34
35            // Connexion à socket.io
36            var socket = io.connect('http://localhost:8080');
37
38            // On demande le pseudo, on l'envoie au serveur et
39            // on l'affiche dans le titre
40            var pseudo = prompt('Quel est votre pseudo ?');
41            socket.emit('nouveau_client', pseudo);
42            document.title = pseudo + ' - ' + document.title;
43
44            // Quand on reçoit un message, on l'insère dans la
45            // page
46            socket.on('message', function(data) {
47                insereMessage(data.pseudo, data.message)
48            })
49
50            // Quand un nouveau client se connecte, on affiche
51            // l'information
52            socket.on('nouveau_client', function(pseudo) {

```

```
46         $('#zone_chat').prepend('<p><em>' + pseudo + '  
         a rejoint le Chat !</em></p>');  
47     })  
48  
49     // Lorsqu'on envoie le formulaire, on transmet le  
     message et on l'affiche sur la page  
50     $('#formulaire_chat').submit(function () {  
51         var message = $('#message').val();  
52         socket.emit('message', message); // Transmet le  
         message aux autres  
53         insereMessage(pseudo, message); // Affiche le  
         message aussi sur notre page  
54         $('#message').val('').focus(); // Vide la zone  
         de Chat et remet le focus dessus  
55         return false; // Permet de bloquer l'envoi "  
         classique" du formulaire  
56     });  
57  
58     // Ajoute un message dans la page  
59     function insereMessage(pseudo, message) {  
60         $('#zone_chat').prepend('<p><strong>' + pseudo  
         + '</strong>' + message + '</p>');  
61     }  
62     </script>  
63     </body>  
64 </html>
```

La partie un peu complexe se situe dans le **code JavaScript** à la fin du fichier. On y trouve tout le nécessaire côté client pour gérer le Chat :

- La connexion à **socket.io**.
- La demande de son pseudo au client et l'envoi au serveur *via* un signal de type **nouveau_client**. Petit bonus, je me suis permis d'afficher le pseudo dans le **<title>** de la page afin qu'il apparaisse dans les onglets de mon navigateur. Vu que pour mes tests j'ouvre plusieurs onglets sur ma machine, ça m'aide à me souvenir qui est qui !
- La récupération du signal **message** envoyé par le serveur. Dans ce cas, j'insère le message dans la zone **#zone_chat** de la page. J'ai choisi de créer une fonction pour ça car j'ai aussi besoin de cette fonctionnalité au moment de l'envoi du formulaire.
- La récupération du signal **nouveau_client** où j'affiche « XXX a rejoint le Chat ! ».
- La gestion de l'envoi du formulaire. Peut-être la partie la plus délicate. Il faut récupérer le message saisi par le client, l'envoyer au serveur et l'insérer dans notre page (car le serveur transmet le message à tout le monde sauf nous !). J'en profite aussi pour vider la zone de texte, remettre le focus dessus... et bloquer l'envoi « classique » du formulaire. Le **return false** est indispensable si on ne veut pas que la page se recharge suite à l'envoi du formulaire. En fait, **return false** est équivalent à la fonction de jQuery **preventDefault()**.

- Enfin, la fonction `insereMessage()` qui est ma foi très simple. Elle rajoute le message qu'on lui envoie avec le pseudo dans la zone de Chat, au début. La fonction `prepend()` fait partie de **jQuery**, je ne l'invente pas.

J'ai choisi d'utiliser **jQuery** dans mon code pour des raisons pratiques, mais vous pouvez bien entendu faire tout ça en pur **JavaScript** si vous en avez envie (ou avec une autre bibliothèque).

En fait, le **code JavaScript** côté client est peut-être le plus gros. La partie **Node.js** serveur est, comme vous le voyez, très simple : on se connecte, on envoie des signaux, on récupère des signaux... et c'est tout !

Télécharger le projet

Vous avez tout le nécessaire précédemment, mais si vous voulez vraiment un fichier .zip tout prêt, je vous propose de le télécharger sur le code web suivant :

▷ Télécharger le fichier .zip
Code web : 647648



Petit rappel : il faut faire un `npm install` pour installer les **dépendances** avant de lancer le projet avec `node app.js` !

Allez plus loin !

Vous voulez vous entraîner encore plus avec **socket.io** ? Voici de quoi vous occuper pendant un moment (par ordre approximatif de difficulté croissante) !

- Affichez un message aux autres personnes lors de la déconnexion d'un client.
- Permettez aux clients de changer leur pseudo en plein milieu d'une session de Chat.
- Et si vous rajoutiez des boutons sur la page pour faire jouer des sons préchargés à distance ? Un bon petit « ding » pour que les autres clients endormis se réveillent ?
- Essayez de sauvegarder les messages en mémoire sur le serveur pour qu'on retrouve la liste des derniers messages échangés lorsqu'on se connecte ! Vous pouvez sauvegarder les informations juste en mémoire vive comme on a appris à le faire... ou essayer carrément de coupler **Node.js** à une base de données **MySQL**, **MongoDB**, **redis**, etc.
- Et si vos clients pouvaient s'envoyer des images en plus du texte ?

Et si malgré tout ça vous vous ennuyez encore, reprenez le code du précédent TP7 pour en faire une « todo list partagée en temps réel entre plusieurs clients » ! L'ajout ou la suppression d'une tâche sera automatiquement répercuté auprès des autres clients !

