NodeJS
Premiers pas

Support de cours Version 1.2

Auteur : Grégory Jarrige Document sponsorisé par : Le Défrichoir SAS

Sommaire

Notes de l'auteur :

Je m'appelle Grégory Jarrige.

Je suis développeur professionnel depuis 1991. Après avoir longtemps travaillé sur des gros systèmes et des langages et technos propriétaires, j'ai fait le pari de me former aux technos et langage open source vers 2005-2006. J'ai commencé à développer des applications webs professionnelles à partir de 2007, avant d'en faire mon activité principale à partir de 2010. L'arrivée du HTML5 dans la même période a été pour moi une véritable bénédiction, et surtout un formidable terrain d'expérimentation (avec des API comme Canvas, WebAudio, etc…).

En plus de mon activité de développeur freelance, je suis également formateur - sur des sujets tels que PHP, HTML5, Javascript, SQL – tantôt en entreprise, tantôt dans le cadre de programmes de reconversion (GRETA notamment).

J'ai rédigé ce support en janvier 2018, en vue de proposer une introduction rapide à NodeJS, aux étudiants en informatique, aux développeurs amateurs et professionnels qui sont intéressés par le sujet, et aux codeurs créatifs du meetup CreativeCodeParis. Je tiens à remercier ici les élèves du BTS Systèmes Numérique, du GRETA d'Arpajon, qui ont été les bêta-testeurs de ce support, et m'ont aidé à l'améliorer.

Ce document est disponible en téléchargement libre sur mon compte Github :

https://github.com/gregja/NodeJSCorner

Il est publié sous Licence Creative Commons n° 6.

Ce support vient compléter les supports de cours consacrés à Javascript qui sont disponibles dans le dépôt suivant :

https://github.com/gregja/JSCorner

Si vous n'êtes pas à l'aise avec Javascript, commencez à minima par étudier le support de cours « Cours Javascript Premiers Pas », avant de vous attaquer à l'étude de Node.js.

Si vous n'êtes pas à l'aise avec le SQL (autre sujet présenté dans ce support), je vous recommande de découvrir le sujet au travers du support « Cours SQL Premier Pas » qui se trouve dans le dépôt suivant :

https://github.com/gregja/SQLCorner

1 Introduction

1.1 Une histoire fulgurante

L'histoire du projet NodeJS est suffisamment atypique pour qu'il soit intéressant de la relater ici. Pour la rédaction de ce rapide historique, je me suis appuyé sur des éléments provenant de la page Wikipédia suivante (que je recommande pour une étude plus approfondie) :

https://en.wikipedia.org/wiki/Node.js

Le projet Node.js a été développé par Ryan Dahl en 2009, en premier lieu pour les plateformes Linux et Mac OS X.

Ryan Dahl a conçu Node.js avec l'idée de dépasser les limites - en termes de performances - imposées par le serveur Web le plus populaire de l'époque : Apache HTTP Server. Après plusieurs tentatives avortées avec des langages tels que C, Lua et Haskell, Ryan Dahl s'est intéressé à Javascript en découvrant l'existence du projet V8 développé par Google.

Le projet V8 (ou plus exactement "V8 JavaScript engine") est un moteur JavaScript libre et open source développé par Google au Danemark. Il est notamment utilisé dans les navigateurs Chromium et Google Chrome. Le fait qu'il soit open source offrait à Ryan Dahl la possibilité de l'utiliser comme moteur de son projet de serveur web.

Ryan Dahl a donc implémenté une plateforme de type serveur embarquant V8, combiné avec une boucle d'événements et une API d'Entrées/Sorties (E/S) bas niveau.

Dahl présenta le projet lors de l'inauguration de la JSConf européenne le 8 novembre 2009, où il rencontra un grand succès.

En janvier 2010, le gestionnaire de paquets NPM (pour "Node Package Manager") est introduit dans l'écosystème Node.js. Il facilite la tâche des programmeurs pour publier et partager le code source des bibliothèques de code Node.js, et il est conçu pour simplifier l'installation, la mise à jour et la désinstallation des bibliothèques de codes (appelées aussi "librairies de code").

En juin 2011, Microsoft et Joyent implémentent une version Windows native de Node.js, qui devient véritablement multiplateforme.

En janvier 2012, Dahl s'éloigne du projet, laissant la direction à Isaac Schlueter, collègue et créateur de NPM. En janvier 2014, Schlueter passe la main à Timothy J. Fontaine.

En Décembre 2014, Fedor Indutny crée un fork de Node.js en créant le projet io.js. Il semble qu'à l'époque, une partie de la communauté des développeurs Node.js était mécontente des orientations - ou peut être du manque de transparence - de Joyent, dans la poursuite du projet Node.js. Du fait de conflits sur la gouvernance du projet par Joyent, io.js a été créé avec l'ambition d'offrir une alternative, avec une gouvernance ouverte pilotée par un comité technique séparé. Contrairement à Joyent, les contributeurs de io.js prévoyaient de maintenir le projet à jour, en suivant les dernières versions du moteur JavaScript de Google V8.

En février 2015, l'intention de former une fondation neutre Node.js est annoncée. En juin 2015, les communautés Node.js et io.js décident de travailler ensemble sous la Fondation Node.js.

En septembre 2015, Node.js v0.12 et io.js v3.3 sont « fusionnés » pour former la v4.0. Cette fusion ne s'est passée sans difficultés pour les développeurs, car elle a coïncidé avec la livraison par Google d'une nouvelle version de V8 qui intégrait beaucoup de changements. Cela a complexifié considérablement le chantier de fusion. Mais la nouvelle version de V8 embarquait le support de la toute dernière version du langage Javascript, à savoir ES6 (nom abrégé de la norme ECMAScript 6).

Le projet Node.js a dès lors bénéficié de l'intégration d'ES6, et les développeurs Node.js (développeurs back-end) ont ainsi pu bénéficier des nouveautés d'ES6 avec un temps d'avance sur les développeurs front-end, obligés eux de tenir compte des problèmes de compatibilité entre navigateurs internet.

Avec cette nouvelle version, Node.js embarquait aussi pour la première fois un support à long terme (en anglais « long term support »), ce qui était très rassurant pour les développeurs et les entreprises souhaitant utiliser cette plateforme.

A partir de 2016, le site Web io.js recommande aux développeurs de revenir à Node.js et indique qu'aucune nouvelle version de io.js n'est prévue à l'avenir, du fait de la fusion des deux projets.

1.2 Une liste de projets pléthorique

On le voit ici, l'historique de Node.js est assez atypique et chaotique. Apparu en 2009, ce projet n'a que 8 ans, et c'est pourtant le projet qui a connu la plus forte croissance durant cette période. En plus de servir de serveur web et de serveur d'API à un nombre incalculable de projets, il sert aussi de socle à des projets plus spécifiques tels que :

- Atom, l'éditeur de code développé par Github,

- Electron, le générateur d'applications desktop également développé par Github

- Meteor, la plateforme de développement d'applications JS isomorphiques

- le gestionnaire de tests d'API Postman

- l'éditeur de code Brackets développé par Adobe

- l'IDE Visual Studio Code (de Microsoft), lui même basé sur le projet Electron

- le gestionnaire de tests Mocha,

- le projet Babeljs, compilateur de code JS destiné aux frameworks « old-school »

etc.

On notera que la plupart des frameworks JS du moment (Angular, React, etc.) ne peuvent pas fonctionner sans le support de Node.js et de son écosystème.

Vous trouverez sur internet des pages présentant des listes de projets reposant sur Node.js, comme par exemple la page suivante :

https://github.com/sqreen/awesome-nodejs-projects

1.3 Des conditions très favorables

Le succès de Node.js est indéniable, mais il doit aussi beaucoup aux évolutions récentes du langage Javascript, dont on rappelle qu'il est normalisé et que son vrai nom est ECMAScript.

Le démarrage de Node.js en 2009 a coïncidé avec l'arrivée d'ECMAScript version 5 (généralement abrégé en ES5). ES5 embarquait beaucoup de nouveautés qui amélioraient nettement le langage et lui apportaient un nouvel élan. L'année 2009 est aussi l'année où les navigateurs ont commencé à implémenter massivement les nouvelles normes HTML5 et CSS3, et les nombreuses API conçues par le W3C (APIs accessibles dans les navigateurs exclusivement via Javascript). Le développement dans le même temps de nouveaux frameworks JS de type SPA (Single Page Application), comme AngularJS, a contribué à renforcer l'engouement pour Javascript. Les développeurs de frameworks SPA ont très vite compris les avantages qu'ils pouvaient tirer de Node.js, qui est devenu très vite le socle de nombreux projets.

L'arrivée de ES6 en 2015 est venue encore renforcer la crédibilité de Javascript, en tant que langage de développement d'application, robuste et pérenne.

Le petit tableau de la page suivante, emprunté à Wikipédia, donne un aperçu des nouveautés embarquées dans les dernières versions de Javascript.

ES5	Décembre 2009	Clarification des ambiguïtés des éditions précédentes, ajout des notions suivantes : accesseurs, introspection, contrôle des attributs, fonctions de manipulation de tableaux supplémentaires, support du format JSON, mode strict pour la vérification des erreurs.
ES6	Juin 2015	Modules, classes, portée lexicale au niveau des blocs, itérateurs et générateurs, promesses pour la programmation asynchrone, patrons de destructuration, optimisation des appels terminaux, nouvelles structures de données (tableaux associatifs, ensembles, tableaux binaires), support de caractères Unicode supplémentaires dans les chaînes de caractères et les expressions rationnelles, possibilité d'étendre les structures de données prédéfinies.
ES7	Juin 2016	Opérateur d'exponentiation, nouvelle méthode pour les prototypes de tableaux.
ESnext	En cours de développement	Async/await, opérateur de *binding*, décorateurs, SIMD, observable, attributs d'instances publics et privés.

Source du tableau ci-dessus :

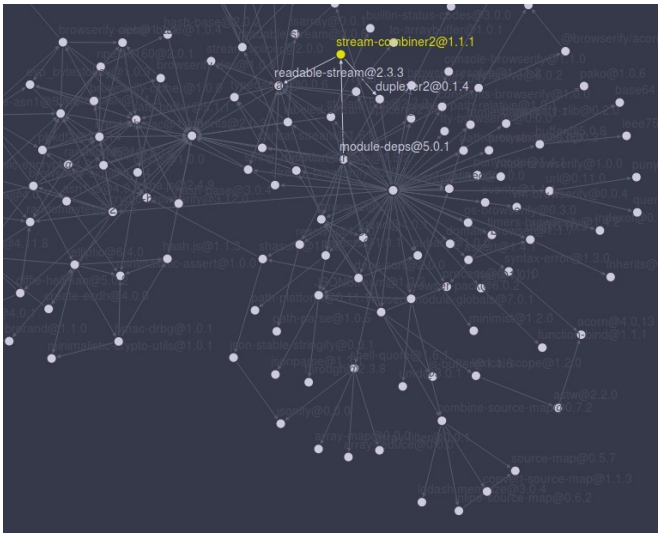
https://fr.wikipedia.org/wiki/ECMAScript#ECMAScript_Edition_5_(ES5)

1.4 Un éco-système extrêmement riche

On en verra un petit aperçu tout au long de ce tuto d'introduction, le nombre de packages disponible dans l'environnement Node.js est tout simplement phénoménal. Mais ce choix pléthorique s'accompagne d'une certaine complexité au niveau de la gestion des projets, en particulier du fait des dépendances inévitables entre packages.

On peut en avoir un aperçu en s'appuyant sur des outils graphiques de visualisation de dépendances, comme par exemple :

http://npm.anvaka.com



(exemple ci-dessus avec les dépendances du package « browserify »)

1.5 Du patinage dans les versions

L'histoire somme toute récente, et quelque peu cahotique de Node.js, a plusieurs conséquences.

Tout d'abord la numérotation des versions est à peu près aussi cahotique que l'histoire que nous venons de conter (cf. tableau ci-dessous).

Ensuite on relève sur le terrain une grande disparité dans les versions utilisées sur les projets. Cette problématique des versions se répercute aussi sur les packages et les problèmes de compatibilité entre le socle (Node.js) et ses dépendances (les packages). La montée de version peut donc s'accompagner de nombreuses révisions au niveau des librairies de code dont certaines deviennent très vite obsolètes.

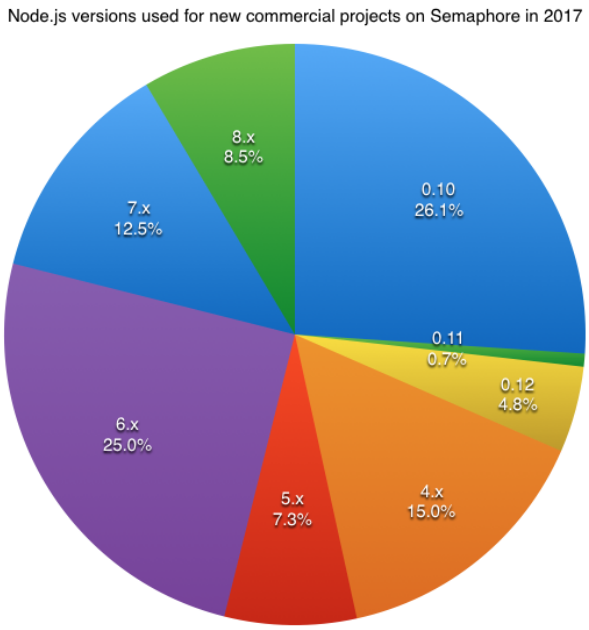
La situation tend néanmoins à se stabiliser, avec l'arrivée en 2017 des versions 8 (stable) et 9 (développement). Le tableau des versions emprunté à Wikipédia permet d'y voir un peu plus clair :

Release	Code name	Release date	LTS status	Active LTS start	Maintenance start	Maintenance end
v0.10.x		2013-03-11	End-of-life	-	2015-10-01	2016-10-31
v0.12.x		2015-02-06	End-of-life	-	2016-04-01	2016-12-31
4.x	Argon	2015-09-08	Maintenance	2015-10-01	2017-04-01	April 2018
5.x		2015-10-29	No LTS	N/A		
6.x	Boron	2016-04-26	**Active**	2016-10-18	April 2018	April 2019
7.x		2016-10-25	No LTS	N/A		
8.x	Carbon	2017-05-30	**Active**	2017-10-31	April 2019	December 2019
9.x		2017-10-31	No LTS	N/A		
10.x			Pending	October 2018	April 2020	April 2021

Cette problématique des versions perdurera certainement quelques temps dans l'écosystème de Node.js, comme le laisse entendre l'article ci-dessous, publié en décembre 2017 par le site developpez.com :

https://www.developpez.com/actu/180621/Semaphore-de-nombreux-nouveaux-projets-commerciaux-sont-developpes-dans-des-langages-qui-ne-sont-plus-supportes-quelles-en-sont-les-raisons/

« Beaucoup de choses sont arrivées au runtime Node.js ces dernières années. Après avoir eu une adoption précoce, il a vu sa croissance ralentie, été forké, avant d'arriver enfin à une consolidation avec un nouveau calendrier de diffusion. En conséquence, la réalité est que près d'un tiers de tous les projets sont basés sur une version obsolète de Node, alors que moins de 10 % utilisent une version publiée en 2017 (v8 ou v9). Node 9 a été publié à l'automne 2017, mais Semaphore ne note pas encore d'adoption significative. Il est ici important de noter que c'est seulement depuis le mois de mars qu'AWS Lambda prend en charge la version 6.10 de Node.js alors que la version 0.10 de Node.js a été dépréciée fin avril. Cela peut-il expliquer en partie l'utilisation de versions Node.js non prises en charge dans les projets commerciaux. »



Nous verrons par la suite que cette question des versions a un impact aussi sur la procédure d'installation.

1.6 Des caractéristiques uniques

Une des raisons du succès indéniable de Node.js, c'est certainement le fait qu'il permette aux développeurs d'utiliser un seul et même langage pour développer du code back-end (côté serveur) et du code front-end (côté navigateur). Avant l'arrivée de Node.js, il était nécessaire de maîtriser un langage côté front (le JS, souvent couplé avec un ou plusieurs frameworks), et un autre langage côté back (qui pouvait être Java, PHP, Python, Ruby, etc.).

Les raisons du succès de Node.js sont aussi à rechercher du côté de ses caractéristiques uniques.

Les solutions webs concurrentes de Node.js utilisent généralement un mode de fonctionnement dans lequel plusieurs threads sont activés par défaut, et mis en attente de requêtes HTTP entrantes. Comme on n'a pas de visibilité sur le nombre de threads susceptibles d'être consommés au plus fort de l'activité, on a tendance à surdimensionner le nombre de threads, pour pouvoir supporter les pics de charge inopinée. Or chaque thread activé, consomme de la mémoire et de la CPU, même quand il ne fait rien. Le serveur en charge des threads, et de la répartition des tâches sur les différents threads disponibles, est lui aussi consommateur de mémoire et de CPU... tout cela n'est pas très bon pour la planète.

Node.js fonctionne selon un mode très différent, car il est « mono-thread », c'est à dire qu'il ne parallélise pas les tâches dans des threads séparés, il les place dans une file d'attente unique (en anglais « job queue »), pilotée par une boucle d'événements (en anglais « event loop »).

Le problème avec cette solution, c'est que si une tâche est trop longue, elle va pénaliser les autres tâches qui attendent derrière, dans la file d'attente. En général, les traitements qui sont longs, le sont parce qu'ils sont pénalisés par des E/S lentes (*). Les E/S lentes, ce sont pricipalement les accès disques (pour les lectures de fichiers) et les accès bases de données (SQL notamment). Pour éviter cela, Node.js implémente un modèle dit d'E/S non bloquant (en anglais « non blocking I/O »). Ce modèle permet de supporter des dizaines de milliers de connexions simultanées, avec une consommation de mémoire et de CPU très inférieure à ceux d'une gestion traditionnelle de type « multi-thread ».

Pour permettre l'utilisation de ce modèle d'E/S non bloquant, il va être nécessaire de recourir à des techniques de programmation comme les fonctions de rappel (en anglais « callbacks »). Mais je préfère en rester là pour le moment, nous verrons ces questions en détail, quand nous aborderons la gestion des accès fichiers.

(*) : E/S pour « Entrée/Sorties », ou en anglais I/O pour « Input/Output »

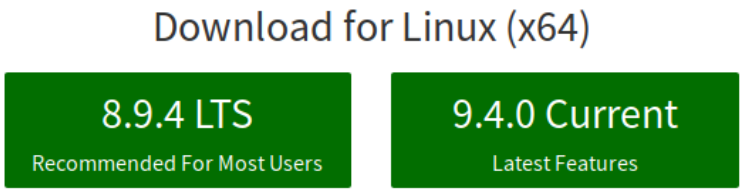
2 Introduction à Node.js

2.1 Installation

La procédure d'installation de Node.js diffère selon les systèmes d'exploitation.

Le site officiel de Node.js est le suivant :
https://nodejs.org/fr/

Au moment où je rédige la première version de ce support (janvier 2018), les versions proposées sur le site officiel sont les suivantes :
- version 8.9.4, version stable offrant un support long terme (LTS)
- version 9.4.0, version de développement offrant l'accès aux toutes dernières fonctionnalités



La version LTS est généralement celle que l'on recommande pour des environnements de production. Si vous souhaitez développer des applications devant être livrées en production rapidement, c'est la version que vous devez privilégier. Si vous êtes davantage dans une optique de formation et/ou de veille technologique, alors vous préférerez peut être la version de développement (mais attention, elle sera peut être moins stable).

Pour une installation sur Windows, on n'a pas trop de questions à se poser. Il suffit d'aller sur le site officiel, de télécharger le binaire d'installation, et de l'exécuter. Ce binaire installe à la fois Node.js et NPM (le gestionnaire de dépendances).

Pour une installation sur Mac OS X, privilégiez si possible une installation via homebrew, avec la commande suivante :

```
brew install node
```

Si vous avez le moindre doute, un petit tour sur cette page pourra vous aider :
https://nodejs.org/en/download/package-manager/#macos

Sur Linux, la situation est plus contrastée. Vous pouvez sans problème installer Node.js via le gestionnaire de package standard de votre distribution Linux, et d'ailleurs c'est ce que je vous recommande de faire (plutôt que de vous embêter à le télécharger sur le site officiel). Par exemple, sur Linux Debian, et sur Ubuntu, vous pouvez lancer l'installation simultanée de

Node.js et de NPM via la commande suivante :

```
sudo apt-get update
sudo apt-get install nodejs npm
```

Vous pouvez aussi en profiter pour installer Curl (si pas déjà fait), car nous en aurons besoin dans un prochain chapitre :

```
sudo apt-get install curl
```

Une fois l'installation terminée, tapez dans un terminal la commande suivante :

```
node -v
```
(ou la commande « node --version » qui est strictement équivalente)

Vous allez très certainement constater que vous avez installé une version 6.x de Node.js.
Plus anecdotique, mais néanmoins intéressant à savoir, vous pouvez connaître la version de NPM avec la commande suivante :

```
npm -v
```
(ou la commande « npm --version » qui est strictement équivalente)

Si votre version de Node.js est une 6.x, votre version de NPM devrait être une 4.x.

Bon, c'est très bien, mais ce serait encore mieux si on pouvait disposer d'une des versions de Node.js publiées en 2017, soit la 8.x, soit la 9.x. Pour faire cette mise à jour, je vous recommande de laisser de côté la doc du site officiel, et de suivre les consignes fournies par cette page :

https://github.com/nodesource/distributions#installation-instructions

Si vous utilisez Debian ou Ubuntu, les parties qui vont vous intéresser sont celles que j'ai copié-colle ci-dessous :

Pour installer Node.js v9.x:
NOTE: If you are using Ubuntu Precise or Debian Wheezy, you might want to read about running Node.js >= 6.x on older distros

```
# Using Ubuntu
curl -sL https://deb.nodesource.com/setup_9.x | sudo -E bash -
sudo apt-get install -y nodejs

# Using Debian, as root
curl -sL https://deb.nodesource.com/setup_9.x | bash -
apt-get install -y nodejs
```

Pour installer Node.js v8.x:

NOTE: If you are using Ubuntu Precise or Debian Wheezy, you might want to read about running Node.js >= 6.x on older distros

```
# Using Ubuntu
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs

# Using Debian, as root
curl -sL https://deb.nodesource.com/setup_8.x | bash -
apt-get install -y nodejs
```

Une fois la mise à jour terminée, vérifiez votre numéro de version via la commande :

```
node -v
```

Voilà, vous êtes fin prêt pour travailler, ou presque…
Refaites à tout à hasard un :

```
npm -v
```

Peut être allez-vous voir apparaître le message suivant :

```
gregja@gregja-UX303UB:~$ npm -v
5.5.1

  Update available 5.5.1 → 5.6.0
    Run npm i -g npm to update
```

Si c'est le cas, lancez la commande indiquée ci-dessus, sans oublier la commande « sudo » :

```
gregja@gregja-UX303UB:~$ sudo npm i -g npm
[sudo] Mot de passe de gregja :
Désolé, essayez de nouveau.
[sudo] Mot de passe de gregja :
/usr/bin/npm -> /usr/lib/node_modules/npm/bin/npm-cli.js
/usr/bin/npx -> /usr/lib/node_modules/npm/bin/npx-cli.js
+ npm@5.6.0
added 27 packages, removed 11 packages and updated 37 packages in 6.139s
```

Cette fois vous êtes fin prêt.

On notera qu'il est possible de basculer d'une version de Node.js à l'autre au moyen du projet NVM (Node Version Manager). Il s'agit d'un projet distinct de Node.js, dont vous trouverez la présentation sur cette page :

https://gist.github.com/d2s/372b5943bce17b964a79

Cet outil, qui ne fonctionne que sur Linux et Mac OS, s'appuie sur Git pour fonctionner. Je vous confirme qu'il s'installe sans problème sur Ubuntu Studio (distribution que j'utilise à l'heure actuelle). La procédure d'installation – très simple - est décrite dans la page de présentation, je préfère ne pas l'indiquer ici car elle est susceptible de changer (comme j'ai pu le constater en testant la procédure indiquée dans un livre, procédure qui n'était plus valide). Je n'ai pas de recul sur l'utilisation de cet outil, mais je pense qu'il est à prendre en considération, si vous avez besoin de tester du code sur différentes versions de Node.js.

Une fois installé, NVM permet de passer d'une version à l'autre très simplement via la commande « nvm », comme dans les exemples suivants :

```
nvm install v8.9.4
```

```
nvm install v9.3.0
```

2.2 Découverte de REPL

Node.js fournit en standard un mode ligne de commande, appelé REPL, pour Read Eval Print Loop. Ce mode est très pratique pour une initiation à JS, ou encore pour tester certaines fonctions Javascript, etc.

Pour passez en mode REPL, saisissez la commande « node » dans un terminal, et amusez-vous à tester les quelques exemples ci-dessous :

```
gregja@gregja-UX303UB:~$ node
> 1+1
2
> "Hello" + " " + "World"
'Hello World'
> 150 + 45.2 + 2*10 - 17/12
213.78333333333333
>
```

Vous voyez l'interpréteur Javascript de Node.js en pleine action. C'est cool non ?
Créez quelques variables et manipulez-les au travers de calculs simples :

```
> var a = 10, b = 20; a+b;
30
> console.log(a+b);
30
undefined
> var c = a+b ; console.log(c);
30
undefined
```

Vous voyez que la bonne vieille fonction « console.log » que vous utilisez certainement quand vous développez pour les navigateurs, est disponible aussi sur Node.js.

Ne vous arrêtez pas à ces quelques exemples, amusez-vous, testez ce qui vous passe par la tête.
Je vous propose maintenant de créer une fonction :

```
> mafonction = function(x) {return x * x}
[Function: mafonction]
> mafonction(10)
100
> mafonction(a)
100
> mafonction(c)
900
```

Au moment où j'ai fait ces tests, la variable « a » valait 10 et la variable « c » valait 30. Vous aurez peut être d'autres valeurs, en fonction des tests préalables que vous aurez réalisés.

On notera qu'il est possible d'exécuter du code JS directement à partir de la commende « node » en lui associant le paramètre « -e », comme dans les exemples suivants :

```
gregja@gregja-UX303UB:~$ node -e "console.log(process.versions.node)"
9.3.0
gregja@gregja-UX303UB:~$ node -e "console.log(process.versions)"
{ http_parser: '2.7.0',
  node: '9.3.0',
  v8: '6.2.414.46-node.15',
  uv: '1.18.0',
  zlib: '1.2.11',
  ares: '1.13.0',
  modules: '59',
  nghttp2: '1.25.0',
  openssl: '1.0.2n',
  icu: '60.1',
  unicode: '10.0',
  cldr: '32.0',
  tz: '2017c' }
```

Je ne pense pas que vous ferez un usage intensif de ce mode, mais c'est quand même bon de savoir que vous l'avez à disposition.

Une petite astuce intéressante à connaître : vous avez effectué un calcul, mais vous avez oublié d'indiquer à REPL que vous souhaitiez récupérer ce résultat dans une variable ? Eh bien vous pouvez rattraper le coup en déclarant une nouvelle variable à laquelle vous affectez le symbole « underscore », et le tour est joué :

```
> 10*20
200
> var distrait = _
undefined
> distrait
200
```

Quelques raccourcis claviers à connaître :

- ctrl + c : stoppe la commande en cours (si trop longue)
- ctrl + c (2 fois) : stoppe REPL
- ctrl + d : stoppe REPL
- touches flèches haut et bas: parcourt l'historique des commandes en avant et en arrière
- touche Tabulation : Affiche les objets et commandes usuels de Javascript (Array, Boolean, Date, Error, etc.), ou les méthodes associées à un objet saisi au préalable (essayez par exemple « Math. » suivi de la touche Tabulation)

La commande .help affiche la liste des commandes usuelles de REPL :

```
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the repl
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file
```

2.3 Premier script

Il est possible d'encapsuler le code de la fonction précédente dans un script, et de demander à Node.js d'exécuter ce script.

Pour ce faire, créez un répertoire de test sur votre machine et placez à l'intérieur de ce répertoire le script « test.js » ci-dessous, saisi avec votre éditeur de code préféré :

```
mafonction = function(x) {
    var temp = x * x
    return temp
}
console.log(mafonction(20))
console.log(mafonction(10))
```

Lancez un terminal à l'intérieur de votre répertoire, et saisissez la commande suivante :

```
node test.js
```

Vous devriez voir apparaître les données suivantes dans votre terminal :

```
400
100
```

Vous aurez peut être remarqué que, dans le script ci-dessus, j'ai omis de places des points virgules à la fin de certaines lignes. J'aurais pu en effet placer des points virgules, comme dans l'exemple ci-dessous :

```
mafonction = function(x) {
    var temp = x * x;
    return temp;
}
console.log(mafonction(20));
console.log(mafonction(10));
```

… le script fonctionnera de la même façon que le précédent.
Node.js est donc plus tolérant que les interpréteurs JS embarqués dans les navigateurs. Si vous vous sentez plus à l'aise avec la syntaxe standard utilisant les points virgules, n'hésitez pas à la privilégier. Cela vous permettra en outre d'utiliser le code de certaines fonctions aussi bien côté serveur que côté navigateur (sans nécessiter de retouche particulière).

2.4 Premier Helloworld

Je ne vous ferai pas l'injure de vous faire écrire un Helloworld avec une simple fonction « console.log ». Vous méritez beaucoup mieux que ça.

Alors créons tout de suite un petit serveur de page HTML, à la sauce Node.js.
Dans votre répertoire de travail, créez un nouveau script que vous appellerez « web1.js », et placez-y le code suivant :

```
var http = require('http');
var port = 8080;

http.createServer(function (req, res) {
    var message = 'Salut tout le monde !';
    res.writeHead(200, {
        'Content-Type': 'text/html; charset=utf-8'
    });
    res.write(message);
    res.end();
}).listen(port);

console.log("Serveur tourne sur localhost:"+port);
```

Lancez ce script dans un terminal avec la commande :

```
node web1.js
```

Allez sur votre navigateur préféré et saisissez l'URL suivante :
http://localhost:8080

Si tout s'est bien passé, vous devriez voir apparaître le message suivant dans votre navigateur :

Salut tout le monde !

Vous pouvez modifier le contenu de la variable message, sauvegarder votre modification, puis retourner sur le navigateur et rafraîchir le contenu de la page (via la touche F5).

Vous venez de créer votre premier serveur avec Node.js. C'est cool non ?
Ne vous préoccupez pas trop pour l'instant de la fonction « require », nous en reparlerons plus en détail un peu plus loin dans ce chapitre.

Quand vous en avez assez de jouer avec votre serveur web, vous pouvez le stopper via la combinaison de touches Ctrl+C.

Nous allons dupliquer le script web1.js en un nouveau script web2.js, et y apporter quelques modifications (indiquées en gras).

```
const HTTP = require('http');
const PORT = 8080;

const SERVER = HTTP.createServer(function(req, res) {
    var message = "coucou c'est encore moi !!!";
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.write(message);
    res.end();
});

SERVER.listen(PORT);

console.log('Serveur tourne sur localhost:'+PORT);
```

Lancez ce nouveau script dans un terminal avec la commande :

```
node web1.js
```

Retournez sur votre navigateur et réaffichez la page suivante :

http://localhost:8080

Vous ne devriez pas constater de différence notable côté navigateur, si ce n'est que ce dernier reçoit maintenant du texte pur, et non plus du HTML. Cela est dû au fait que nous avons modifié la valeur du paramètre « Content-Type ».

Nous venons de voir que la création d'un serveur web avec Node.js est une opération triviale. C'est beaucoup plus simple qu'en PHP, pas besoin d'un serveur d'application de type Apache ou Nginx, tout est piloté par Node.js. Mais il est temps d'expliquer un peu plus avant le code que nous venons d'écrire.

Côté serveur, nous avons utilisé le mot clé « const » pour créer les constantes « PORT » et « HTTP » en remplacement des variables « port » et « http ». En effet, Node.js nous permet d'utiliser la nouvelle norme ECMAScript 6 (ES6), donc nous pouvons créer des constantes.

Il existe une convention – largement acceptée par les développeurs – qui consiste à déclarer les constantes en majuscule. Au travers de mes lectures, j'ai remarqué que les développeurs Node.js n'adhèrent pas tous à cette convention, mais j'ai choisi de l'adopter dans ce support de cours, pour vous aider à mieux distinguer les constantes des variables.

Vous noterez que nous avons également créé une constante SERVER, ce qui nous permet de dissocier le code de définition du serveur, du code de lancement (via la méthode « listen »).

Nous détaillerons l'usage de la fonction « require » de manière plus approfondie dans un

prochain chapitre, mais je vous propose de nous arrêter un instant sur cette fonction, pour dégrossir un peu le sujet :

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

Cette fonction « require » est utilisée pour importer différents types de modules. Il peut s'agir :
- de modules fournis en standard par Node.js, comme le module « http » que nous venons d'utiliser. On parle dans ce cas de « core module » ou encore de « core package » (c'est la même chose).
- de modules importés via le gestionnaire de packages NPM (nous verrons cela plus tard)
- de modules écrits par nous (nous verrons également cela un peu plus loin)

Au travers de cette constante HTTP, nous sommes en mesure de faire appel à l'une des fonctions fournies par le module « http », à savoir la fonction « createServer » :

```
var server = HTTP.createServer(function(req, res) {
    var message = "coucou c'est encore moi !!!";
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.write(message);
    res.end();
});
```

La fonction « createServer » est normalisée de manière à recevoir 2 paramètres qui sont :
- au travers du paramètre « req », nous avons la requête HTTP entrante (nous verrons comment l'utiliser dans un prochain exemple)
- au travers du paramètre « res », nous avons la réponse HTTP sortante, générée dans notre exemple au format « text/plain », et qui contiendra dans son corps le message « coucou c'est encore moi !!! ». Ce sont les méthodes « writeHead », « write » qui vont nous permettre de préparer cette réponse et c'est la méthode « end » qui va avoir pour effet de la renvoyer vers le client (ici notre navigateur internet).

Vous noterez que jusqu'ici nous n'avons fait que préparer le serveur. Ce dernier n'est réellement mis en œuvre qu'à partir de la ligne suivante :

```
server.listen(PORT);
```

C'est en effet à partir de là que le serveur démarre, se met à surveiller les requêtes HTTP entrantes sur le port qu'on lui a indiqué, et renvoie des réponses HTTP aux clients qui le sollicitent.

Jusqu'ici le client de notre serveur web, c'est notre navigateur internet. Mais ce pourrait tout aussi bien être un botnet (robot) ou toute autre application, comme par exemple un client Curl. Pour le démontrer, relancez votre script serveur dans un terminal (s'il n'est pas déjà actif) :

```
node web2.js
```

… et lancez la commande « curl » suivante dans **un autre terminal.**
Si tout va bien, vous devriez voir apparaître le message suivant :

```
coucou c'est encore moi !!!
```

Curl reçoit donc bien la réponse transmise par notre serveur web maison.

Avant de conclure ce chapitre, je voudrais apporter une précision sur la méthode « listen » ci-dessous :

```
server.listen(PORT);
```

Si on ne transmet qu'un seul paramètre à la méthode « listen », ce paramètre correspond au numéro de port, et le serveur surveillera ce port sur l'adresse IP 127.0.0.1 correspondant au serveur local (localhost). Mais on peut aussi souhaiter travailler avec une adresse IP différente, pour cela il suffit de la préciser en second paramètre, comme dans l'exemple suivant :

```
server.listen(8000, '190.10.10.3');
```

Voilà, ça fait beaucoup de notions nouvelles dans ce chapitre d'introduction, même si je suis resté volontairement assez évasif sur certains sujets, pour ne pas risquer de vous noyer dans des détails.

Avant de conclure ce chapitre, je crois utile de vous montrer une variante du script précédent. Dans cette variante, j'ai transféré le code l'essentiel du code de gestion du serveur dans une fonction que j'ai appelée « httpserver ». Cela ne change rien au fonctionnement du script, mais je pense que cela améliore la lisibilité de l'exemple. Je vous laisse le soin de comparer le code ci-dessous avec la version précédente, pour vous habituer aux 2 formes d'écriture :

```
const HTTP = require('http');
const PORT = 8080;
var httpserver = function(req, res) {
    var message = "coucou c'est encore moi !!!";
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.write(message);
    res.end();
};
const SERVER = HTTP.createServer(httpserver).listen(PORT);
console.log('Serveur tourne sur localhost:'+PORT);
```

Dans le chapitre qui suit, nous allons poursuivre notre découverte de Node.js, toujours de manière progressive, en construisant des serveurs webs plus intéressants.

2.5 Serveur web amélioré

Je vous propose de créer maintenant un serveur web renvoyant une page HTML correctement structurée. Voici donc un nouvel exemple avec le script web3.js :

```
const HTTP = require('http');
const PORT = 8080;

const SERVER = HTTP.createServer(function(req, res) {
    var message = `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>NodeJS exemple 1</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <div>TODO write content</div>
    </body>
</html>`;
    res.writeHead(200, {
        'Content-Type': 'text/html; charset=utf-8'
    });
    res.write(message);
    res.end();
});

SERVER.listen(PORT);

console.log('Serveur tourne sur localhost:'+PORT);
```

Dans cet exemple, j'ai utilisé une nouveauté de ES6, à savoir la possibilité de créer une variable « message » de type « littéral de modèle ». C'est très pratique pour créer des chaînes de caractères multilignes, et cela se déclare en utilisant l'apostrophe inverse `. Ces littéraux de modèle se rapprochent un peu de la syntaxe « heredoc » de PHP, mais ils sont plus puissants et permettent dans de nombreux cas de se passer complètement de moteurs de templating. Nous allons voir un exemple d'utilisation plus intéressant dans le chapitre suivant.

2.6 Requêtes Get et paramètres

Si vous avez étudié le cours « PHP premier pas » qui se trouve dans ce dépôt :
https://github.com/gregja/PhpCorner

… alors vous savez que l'on interagit avec les pages HTML au travers de requêtes HTTP de type GET ou POST, et que ces requêtes peuvent contenir des paramètres que l'on souhaitera récupérer pour gérer certaines actions côté serveur.

Supposons que nous souhaitions transmettre à notre serveur 2 paramètres (gamer1 et gamer2) contenant chacun une valeur différente. Cette transmission pourrait se faire via Curl dans une requête HTTP de type GET très simple comme celle-ci :

```
curl 'http://localhost:8080?gamer1=titi&gamer2=grosminet'
```

Attention : avec Curl, quand on veut transmettre plus d'un paramètre dans une requête, il est impératif d'encapsuler la requête entre quotes simples, comme dans l'exemple ci-dessus (sinon seul le premier paramètre est pris en compte).

Alors, comment faire pour récupérer les paramètres « gamer1 » et « gamer2 » côté serveur ?

Vous trouverez page suivante une solution, que j'ai implémentée dans le script « web4_get.js ».

```
const HTTP = require('http');
const URL = require('url');  // ajout du core module url
const PORT = 8080;

const SERVER = HTTP.createServer(function(req, res) {
    var page = URL.parse(req.url).pathname;
    if (page === '/') {
        res.writeHead(200, {
            'Content-Type': 'text/html; charset=utf-8'
        });
        // extraction des paramètres de req
        var params = URL.parse(req.url, true).query;

        // construction de l'objet data contenant les données
        //   utilisées par le template
        var data = {};
        if ('gamer1' in params) {
            data.gamer1 = params.gamer1;
        } else {
            data.gamer1 = 'inconnu';
        }
        if ('gamer2' in params) {
            data.gamer2 = params.gamer2;
        } else {
            data.gamer2 = 'inconnu';
        }
        // template HTML
        var message = `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>NodeJS exemple 1</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <div>Bonjour ${data.gamer1} et ${data.gamer2} </div>
    </body>
</html>`;
        res.write(message);
    } else {
        res.writeHead(404, {
            'Content-Type': 'text/html; charset=utf-8'
        });
        res.write('Error 404 : page not found');
    }
    res.end();
});

SERVER.listen(PORT);

console.log('Serveur tourne sur localhost:'+PORT);
```

Après avoir lancé ce script avec la commande « node », pensez à le tester avec Curl, mais aussi et surtout dans votre navigateur avec l'URL suivante :

```
http://localhost:8080/?gamer1=titi&gamer2=grosminet
```

Bon, je reconnais qu'il y aurait à redire concernant la modularité et la lisibilité du code. On pourrait en effet le découper différemment en déportant certaines parties dans des fonctions distinctes, cela pourrait améliorer la lisibilité de l'ensemble. Mais j'ai préféré vous proposer cette version un peu « brute » pour ne pas mélanger trop de sujets.

On voit dans cet exemple que les « littéraux de modèles » (en anglais : « template litterals ») offrent beaucoup de souplesse pour fusionner des données avec des templates. C'est ce que nous faisons ici en fusionnant le template contenu dans la variable « message » avec des données de l'objet « data », comme dans l'extrait suivant :

```
     <div>Bonjour ${data.gamer1} et ${data.gamer2} </div>
```

Ces littéraux de modèles sont tellement puissants que l'on peut écrire des choses de ce genre (exemples empruntés à la doc Mozilla) :

- En ES2015 avec des littéraux de modèle et sans imbrication :

```
const classes = `header ${ isLargeScreen() ? '' :
    (item.isCollapsed ? 'icon-expander' : 'icon-collapser') }`;
```

- En ES2015 avec des littéraux de modèle imbriqués :

```
const classes = `header ${ isLargeScreen() ? '' :
 `icon-${(item.isCollapsed ? 'expander' : 'collapser')}` }`;
```

On peut imbriquer à l'intérieur de littéraux de modèle, des boucles et des structures conditionnelles. Par exemple, dans un prochain chapitre, pour générer un champ de formulaire de type « select » tel que celui ci :

```
<select id="id_col_rech" name="col_rech" class="form-control">
   <option value="1">code France</option>
   <option value="2">code ISO</option>
   <option value="3" selected>Libellé</option>
</select>
```

… nous utiliserons le code suivant :

```
var prm_col_rech = 3; // valeur par défaut du champ de saisie

// liste des options du champ de saisie
var list_col_rech = [];
list_col_rech.push({valeur:'1', libelle: 'code France'});
list_col_rech.push({valeur:'2', libelle:'code ISO'});
list_col_rech.push({valeur:'3', libelle:'Libellé'});

// utilisation de la fonction map() et de littéraux de modèles imbriqués
var opt_col_rech = `${list_col_rech.map((item, i) =>
           `<option value="${item.valeur}" ${item.valeur == prm_col_rech ?
               'selected': '' }>${item.libelle}</option>`).join('\n')}`;

// intégration du html des options dans le code HTML du select final
var html = `<select id="id_col_rech" name="col_rech" class="form-control">
    ${opt_col_rech}
    </select>`;
```

Bon, je reconnais que ça pique un peu les yeux, mais c'est puissant. J'ai mis la fonction « map » en rouge pour vous aider à la localiser, c'est elle qui va « balayer » le tableau des options contenu dans la variable « opt_col_rech ». J'ai mis aussi en vert la structure conditionnelle destinée à générer l'attribut « selected » sur la bonne ligne (celle qui a la valeur « 3 » dans notre exemple).

Avec un moteur de templating comme Handlebars.js, on aurait pu écrire la même chose avec une syntaxe un peu plus lisible. Mais l'avantage des littéraux de modèle d'ES6, c'est que l'on n'a pas besoin de charger de module supplémentaire, on fait appel ici à une fonction native, qui est compilée dans l'interpréteur Javascript de V8.

A noter que l'on pourrait imbriquer des littéraux de modèle de manière plus complexe encore, mais je recommande de ne pas le faire. Soyez sympas, pensez aux collègues qui vont reprendre votre code en maintenance, à moins bien sûr que vous n'ayez envie de vous faire détester.

Pour de plus amples précisions sur les littéraux de modèle :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Litt%C3%A9raux_gabarits
https://www.keithcirkel.co.uk/es6-template-literals/

2.7 Formulaires et requêtes Post

Pour les formulaires HTML destinés à la mise à jour de données, on utilise de préférence la méthode POST, qui a pour effet de déclencher des requêtes HTTP de type POST. On peut aussi utiliser la méthode GET, mais on la réservera plutôt à de simples formulaires de recherche. On peut en profiter pour rappeler que les requêtes HTTP de type POST offrent quelques avantages sur les requêtes HTTP de type GET, telles que :
- les paramètres des requête HTTP de type POST - paramètres qui correspondent aux champs de saisie du formulaire - ne sont pas directement accessibles dans l'URL de la requête (et sont donc plus difficiles à modifier pour un utilisateur lambda)
- les requêtes de type POST peuvent encapsuler un plus grand nombre de paramètres que les requêtes de type GET (qui sont limitées en taille)
- les requêtes de type GET ont tendance à perdre en route les caractères saisis dans des alphabets exotiques, alors que les requêtes de type POST n'ont pas de souci avec ça

Ces rappels étant faits, je vous propose d'implémenter un exemple de formulaire très simple utilisant la méthode POST et permettant la saisie d'un nom et d'un prénom. La validation du formulaire déclenche l'affichage des données saisies dans une seconde page générée par le même script. Pour l'instant, on n'implémente pas de contrôle de saisie (du genre « zone obligatoire »).

Le code source étant relativement long, je vais le détailler ici bloc par bloc, et vous retrouverez le code source dans son intégralité en annexe (chapitre 5.3).

Voici tout d'abord notre jeu de constantes, avec une nouveauté, avec l'importation du core module « querystring » que nous utiliserons pour extraire les paramètres issus du formulaire. Vous noterez que j'ai ajouté une constante HOST définissant l'adresse IP du serveur (ce n'était pas indispensable ici) :

```
const HTTP = require('http');
const URL = require('url');
const QUERYSTRING = require('querystring');
const PORT = 8080;
const HOST = '127.0.0.1';
```

Pour l'affichage du formulaire, je vous propose d'intégrer son code dans une fonction, ceci afin d'alléger un peu le code de gestion du serveur. Nous allons donc créer les 3 fonctions suivantes :
- testform() : renvoie le code HTML du formulaire
- errorInfo() : pour l'envoi au client des éventuelles erreurs (notamment 404)
- template() : renvoie le code HTML de la page HTML

Pour la création de ces 3 fonctions, je vous propose d'utiliser une nouveauté d'ES6, à savoir les fonctions fléchées (ou « arrow functions »). Si on prend l'exemple de la fonction testform(), ça donne ceci :

```
/**
 * Fonction dédiée à la génération du formulaire
 * @returns {String}
 */
var testform = () => `
<form action="message" method="post">
<label>Nom :<input name="nom"></label><br><br>
<label>Prénom :<input name="prenom"></label><br><br>
<input type="submit" name="Valider">
</form>
`;
```

La syntaxe des fonctions fléchées au premier abord, mais une fois qu'on connaît le principe, elle se révèle assez pratique. Nous aurions pu écrire la même chose en utilisant la syntaxe classique, un peu plus verbeuse, et cela aurait donné ceci :

```
/**
 * Fonction dédiée à la génération du formulaire
 * @returns {String}
 */
var testform = function() {
   return `
<form action="message" method="post">
<label>Nom :<input name="nom"></label><br><br>
<label>Prénom :<input name="prenom"></label><br><br>
<input type="submit" name="Valider">
</form>
`;
}
```

Pour la fonction errorInfo(), c'est sensiblement la même chose, si ce n'est que cette nouvelle fonction reçoit 2 paramètres en entrées, avec les variables « code » et « res ») :

```
var errorInfo = (code, res) => {
    //res.statusCode = code;
    res.writeHead(code, {
        'Content-Type': 'text/html; charset=utf-8'
    });
    res.write(`Error ${code} : page not found`);
    res.end();
};
```

Pour la fonction template(), il y a une petite subtilité, car j'utilise ici une autre nouveauté d'ES6, à savoir la possibilité de définir une valeur par défaut pour le paramètre « divcontent » (c'est le même principe qu'en PHP par exemple).

Voici le code de la fonction template() :

```
var template = (divcontent='') => `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>NodeJS exemple 1</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <div>${divcontent}</div>
    </body>
</html>`;
```

Pour la suite, nous allons déclarer notre serveur avec la méthode createServer() que vous connaissez déjà, et nous allons déterminer si la requête entrante est de type GET ou POST pour ajuster le comportement du serveur selon le contexte :
- si la requête entrante est de type GET, alors nous souhaitons afficher une page HTML contenant un formulaire
- si la requête entrante est de type POST, alors nous souhaitons récupérer les paramètres nom et prénom transmis via le formulaire, et nous souhaitons afficher un message de type « bonjour <nom> <prénom> »

```
var server = HTTP.createServer().listen(PORT, HOST);

server.on('request', function(req, res) {

    if (req.method === 'GET') {
        if (req.url === '/') {
            res.writeHead(200, {'Content-Type': 'text/html'});
            var message = template(testform());
            res.write(message);
            res.end();
        } else {
            errorInfo(404, res);
        }
    }

    if (req.method === 'POST') {
        var datas = {};
        var post_data = '';
        if (req.url === '/message') {

            // Réception des données provenant du formulaire
            req.on('data', function (data) {
                post_data += data;
            });

            // Fin de la réception, affichage de la page finale
            req.on('end', function () {
                datas = QUERYSTRING.parse(post_data);
                res.writeHead(200, {'Content-Type': 'text/html'});
                var message = template(`Bonjour ${datas.prenom} ${datas.nom} `);
                res.write(message);
                res.end();
            });
        } else {
```

```
            errorInfo(404, res);
        }
    }

});

console.log('Serveur tourne sur '+HOST+':'+PORT);
```

La partie la plus délicate se situe dans la partie « POST » avec la récupération des données du formulaire qui se fait en 2 temps, via les méthodes « req.on(data) » et « req.on(end) ». Ce mécanisme de récupération des données peut sembler surdimensionné, mais il est standardisé dans Node.js pour couvrir différents besoins.

Je vous encourage à ajouter des appels à la fonction console.log() sur les variables post_data et datas, afin de mieux comprendre les mécanismes en œuvre dans ce script.

2.8 Un client de formulaire en Node.js

Je vous propose de créer un script Node.js qui se comportera comme un client web et soumettra le formulaire de saisie à la place d'un utilisateur humain. J'ai appelé mon script « web5_client.js », voici son code :

```
const HTTP = require('http');
const QUERYSTRING = require('querystring');

// définition des paramètres du formulaire
var postData = QUERYSTRING.stringify({
    nom: 'Caan',
    prenom: 'Jerry'
});

// paramètres de la requête HTTP de type POST
var options = {
    hostname: 'localhost',
    port: 8080,
    method: 'POST',
    path: '/message',
    headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
        'Content-Length': postData.length
    }
};

var req = HTTP.request(options, function (res) {
    console.log('STATUS:', res.statusCode);
    console.log('HEADERS:', JSON.stringify(res.headers));

    res.setEncoding('utf8');

    res.on('data', function (chunk) {
        console.log('BODY:', chunk);
    });

    res.on('end', function () {
        console.log('No more data in response.');
    });
});

req.on('error', function (e) {
    console.log('Problem with request:', e.message);
});

req.write(postData);
req.end();
```

Saisissez dans un terminal la commande « node web5_client.js ». Si vous n'avez pas commis d'erreur, vous devriez obtenir l'affichage suivant :

```
gregja@gregja-UX303UB:~/Documents/nodejsworks/app01$ node web5_client.js
STATUS: 200
HEADERS: {"content-type":"text/html","date":"Mon, 08 Jan 2018 00:21:50 GMT","connection":"close","transfer-encoding":"chunked"}
BODY: <!DOCTYPE html>
<html lang="fr">
    <head>
        <title>NodeJS exemple 1</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <div>Bonjour Jerry Caan </div>
    </body>
</html>
No more data in response.
```

Vous voyez que la construction d'un client avec Node.js est à peine plus compliquée que la construction d'un serveur. Vous pourrez réutiliser ce principe si vous avez besoin de faire appel à des API fonctionnant selon l'architecture REST (sujet que nous aborderons ultérieurement).

3 Approfondissements

3.1 Projets Node.js et NPM

Jusqu'ici, nous avons utilisé Node.js de manière très libre, sans nous préoccuper d'initialiser un véritable projet Node.js. Il est temps de nous pencher sur la question.

Pour initialiser un projet, ouvrir un terminal, se placer sur le répertoire destiné à contenir le projet (dans mon cas c'est « app01 »), et saisir la commande suivante :

```
npm init -y
```

NPM vient de créer dans votre répertoire un fichier « package.json », dans lequel il a inséré les informations suivantes :

```
{
  "name": "app01",
  "version": "1.0.0",
  "description": "",
  "main": "hello.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Ce fichier sera utilisé par NPM pour référencer tous les packages que vous souhaiterez utiliser par la suite au sein de votre projet.

La commande « npm init -y » est un raccourci pour la commande « npm init --yes ».

A quoi sert ce paramètre « --yes » ? A rendre NPM moins bavard, puisqu'il va ainsi créer le fichier pakage.json sans vous poser de question, et en mettant des valeurs par défaut dans les différentes rubriques du fichier package.json.

Je vous invite à essayer la même commande sans le paramètre « --yes » :

```
npm init
```

Dans ce cas, NPM vous demandera quelles valeurs vous souhaitez affecter à chaque rubrique du fichier package.json. Comme nous sommes en mode découverte, et que nous créons notre première application qui est une application de test, la plupart des questions posées par NPM nous intéressent peu pour l'instant.

Je disais à l'instant que le fichier package.json servait à référencer les différents packages (ou modules) utilisés par notre projet. Si je souhaite utiliser – dans mon projet Node.js - le package dédié aux bases de données MySQL et MariaDB, je dois importer le package « mysql2 ». Pour ce faire, je peux commencer par vérifier que le package que je recherche existe bien avec la commande suivante :

```
npm search mysql2
```

Vous allez constater qu'il existe pas mal de packages liés au package mysql2.

```
NAME                      | DESCRIPTION          | AUTHOR           | DATE
mysql2                    | fast mysql driver.…  | =sushantdhiman…  | 2017-11-19
sql-template-strings      | ES6 tagged template… | =felixfbecker    | 2016-09-17
promise-mysql2            | A promise wrapper…   | =maifuquan       | 2017-11-17
dtd2mysql                 | Command line tool…   | =linusnorton     | 2018-01-03
panthera-mysql-adapter    | MySQL database…      | =pantherajs      | 2016-12-07
naomi-mysql               | MySQL connector for… | =jmike           | 2016-06-07
mysql-lite                | this package is a…   | =kaizhu          | 2016-12-28
namshi-node-mysql         | Small wrapper for…   | =joejean…        | 2017-08-24
express-mysql2-session-pr | Lets you use MySQL2… | =liminal18       | 2017-11-25
omise-adapter             |                      |                  |
node-mysql2-wrapper       | Shortcuts and…       | =colinmathews    | 2016-08-04
mysequel                  | A best-practices…    | =chipersoft      | 2017-08-19
jaypha-mysql-ext          | Some convenience…    | =jaypha          | 2017-05-02
mysql-procedures-2-json   | Get a Mysql's…       | =sethstalley     | 2017-02-11
mysql2-model              | Fork of…             | =andersoo        | 2017-12-07
json2mysql                | A tool for…          | =chopperlee      | 2017-07-07
@ag1/mysql_wrapper        | ```json…             | =kpping          | 2018-01-04
squiggle                  | Query builder for…   | =hugowetterberg  | 2015-01-08
nbatis                    | The node.js version… | =vyspace         | 2017-09-15
mysql2json                | Reads the schema of… | =schtoeffel      | 2014-06-04
orm-2.1.3                 | NodeJS…              | =ddo             | 2014-03-07
```

Attention : la première fois que vous lancez un « npm search », c'est un peu long, c'est normal, car NPM est en train de mettre à jour sa base de données interne. Ce sera plus rapide pour les prochains appels.

Je peux aussi effectuer un test d'installation, pour m'assurer que le package qui m'intéresse n'entre pas en conflit avec d'autres packages :

```
npm install-test mysql2
```

A ce stade, vous allez peut être voir apparaître le message d'erreur suivant :

```
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN app01@1.0.0 No description
npm WARN app01@1.0.0 No repository field.

+ node-mariadb@0.1.1
added 11 packages in 3.76s

> app01@1.0.0 test /home/gregja/Documents/nodejsworks/app01
> echo "Error: no test specified" && exit 1

Error: no test specified
npm ERR! Test failed.  See above for more details.
```

J'avoue que j'ai été surpris la première fois que j'ai vu ce message, car je ne l'avais pas rencontré sur des versions antérieures de Node.js. Je suppose que c'est une nouveauté de Node.js en version 9 (et peut être aussi en version 8, mais je ne l'ai pas vérifié). D'après les éléments trouvés dans les forums, il semble que ce message soit dû au fait qu'aucun outil de test n'est configuré sur le projet. Pour y remédier, vous pouvez demander l'installation du package « mocha » qui est un standard pour la gestion des tests :

```
npm install mocha --save-dev
```

Une fois l'installation terminée, modifiez le contenu du fichier package.json, en indiquant « mocha » sur la ligne « test », comme ceci :

```
  "scripts": {
    "test": "mocha"
  },
```

Cette modification permettra de lancer le package « mocha » automatiquement quand vous lancerez la commande :

```
npm test
```

Sauvegardez le fichier package.json modifié, puis relancez la procédure de test d'installation pour le package « mysql2 » :

```
npm install-test mysql2
```

Si vous n'obtenez plus de message d'erreur, alors vous pouvez procéder à l'installation du package :

```
npm install mysql2
```

Vous pouvez aussi installer un package de manière globale, afin qu'il soit disponible pour l'ensemble des projets Node.js hébergés sur votre machine. Pour cela, vous devez ajouter le paramètre -g, comme ceci :

```
npm install -g mysql2
```

Le paramètre -g est intéressant si vous savez que vous utilisez systématiquement le même module dans tous vos projets (ce serait dommage dans ce cas de l'importer sur chaque projet). Mais attention, le package ainsi déclaré n'est pas présent physiquement dans votre projet, il est stocké dans Node.js. Donc

si vous partagez le code de votre projet avec d'autres développeurs, il faut qu'ils déclarent eux aussi le même package en mode global.

Au fait, les packages qui sont importés dans votre projet en mode local (c'est à dire sans le paramètre -g), sont stockés dans le sous-répertoire « node_modules » qui se trouve à la racine de votre projet. Jetez-y un coup d'oeil, mais surtout ne touchez à rien.

Vous pouvez désinstaller un package :

```
npm uninstall mysql2
```

Et si vous voulez désinstaller un package installé précédemment avec le paramètre -g :

```
npm uninstall -g mysql2
```

Je recommanderais la plus grande prudence dans l'utilisation des packages. Importez dans vos projets le strict nécessaire et pas plus. L'histoire de Node.js est émaillée de nombreux problèmes provoqués par des packages obsolètes ou supprimés, ayant mis en difficulté des projets dépendants de ces packages.

3.2 Modules et Require

3.2.1 Créer ses propres modules

Dans le prochain chapitre, nous aborderons l'utilisation de MySQL et MariaDB dans Node.js.

Nous serons dès lors amenés à créer un script de connexion à la base de données. Il serait dommage de devoir déclarer ce même code de connexion à chaque fois qu'on en a besoin. Mieux vaudrait le déclarer une seule fois et l'inclure là où on en a besoin, à la manière de PHP avec les fonctions « require » ou « include ».

Ca tombe bien, car la fonction « require » que nous avons déjà utilisée dans quelques exemples, sait aussi importer nos propres scripts personnalisés. Nous allons voir comment cela fonctionne au travers d'un exemple simple (et nous réutiliserons ce principe, dans le chapitre suivant, pour l'accès à la base de données).

Commencez par créer un script « tools.js » et insérez-y le code suivant :

```
module.exports = function(datas) {
  if (!Array.isArray(datas)) {
      console.log('parametre pas de type array, traitement interrompu');
      return false;
  }
  let somme = 0,
    i = 0,
    l = datas.length;
    while (i < l) {
        somme += datas[i++];
    }
    return somme;
};
```

Nous venons de créer une fonction anonyme qui sera exportée via la propriété « exports » de l'objet « module » fourni en standard par Node.js.

Ce code ne présente pas de difficulté particulière, il s'agit d'une fonction qui reçoit un paramètre en entrée et vérifie qu'il s'agit bien d'un tableau. Si c'est le cas, elle exécute une boucle pour cumuler les différentes valeurs numériques contenues dans le tableau. On notera que je ne teste pas ici si les données contenues dans le tableau sont bien de type numérique, cela pourra faire l'objet d'une amélioration ultérieure (que je vous laisse le soin d'écrire).

Créez maintenant un second script et appelez « testtools.js ». Insérez dans ce script le code suivant :

```
const SOMME_TAB = require('./tools.js');
let values = [200, 1000, 550];

let total1 = SOMME_TAB( values );
console.log(total1);
```

```
let total2 = SOMME_TAB( 'xx' );
console.log(total2);
```

Exécutez maintenant le script « testtools.js » via la commande :

```
node testtools.js
```

Si tout s'est bien passé, vous devriez voir apparaître les données suivantes dans votre terminal :

```
1750
parametre pas de type array, traitement interrompu
false
```

Voilà, vous venez de créer votre propre composant réutilisable. C'est cool non ?

Mais vous rencontrerez forcément des cas pour lesquels vous souhaiteriez qu'un module contienne plusieurs méthodes. Voici comment procéder. Comme nous sommes en janvier (au moment où je rédige la première version de ce support), période où on souhaite traditionnellement la bonne année. Alors je vous propose de créer un module contenant un jeu de fonctions renvoyant « bonne année » en plusieurs langues. Attention, ce n'est pas de la grande programmation, c'est juste pour vous montrer le principe.

Commencez par créer un script « bonneannee.js » et collez-y le code suivant :

```
module.exports = {
  enAnglais() {
    return 'Happy New Year';
  },

  enFrancais() {
    return 'Bonne Année';
  },

  enItalien() {
    return 'Buon Anno';
  }
};
```

Créez maintenant un script « testbonneannee.js » et placez-y ceci :

```
const GOODYEAR = require('./bonneannee.js');

console.log(GOODYEAR.enAnglais());
console.log(GOODYEAR.enFrancais());
console.log(GOODYEAR.enItalien());
```

Exécutez ce dernier script dans votre console, vous devriez obtenir l'affichage suivant :

```
Happy New Year
Bonne Année
Buon Anno
```

3.2.2 Approfondir Module.exports et Require

L'objet « module.exports » propose plusieurs manières d'exporter les fonctionnalités d'un module. On peut notamment exporter :

- une fonction : `module.exports = function(ops) {...}`
- un objet : `module.exports = {...}`
- plusieurs fonctions : `module.exports.methodeA = function(ops) {...}`
- plusieurs objets : `module.exports.objetA = {...}`

Nous avons vu la fonction require() à plusieurs reprises et nous l'avons utilisée d'au moins 2 manières différentes. Il est temps de faire le point sur les possibilités de cette fonction :

- Import de « core modules/packages » : `const filesystem = require('fs')`
- Import de « npm modules/packages » : `const express = require('express')`
- Import de script perso : `const tools = require('./library/tools.js')`
- Import de fichier JSON : `const dbConfigs = require('./cfg/database.json')`
- Import de répertoires dans un projet : `const routes = require('./routes')`

Pour l'import de fichiers personnels, on peut :

- descendre dans le chemin courant : `const tools = require('./library/tools.js')`

- remonter dans le chemin courant : `const tools = require('../library/tools.js')`

- utiliser un chemin absolu :

```
const tools = require('/var/www/app/library/tools.js')
```

Attention : le chargement des fichiers contenus dans un répertoire ne fonctionnera que dans le cas où il n'existe pas de fichier avec l'extension « .js » portant le même nom que le répertoire considéré :

```
const routes = require('./routes')
```

3.2.3 Tour d'horizon des core modules

Voici la liste des principaux « core modules » :

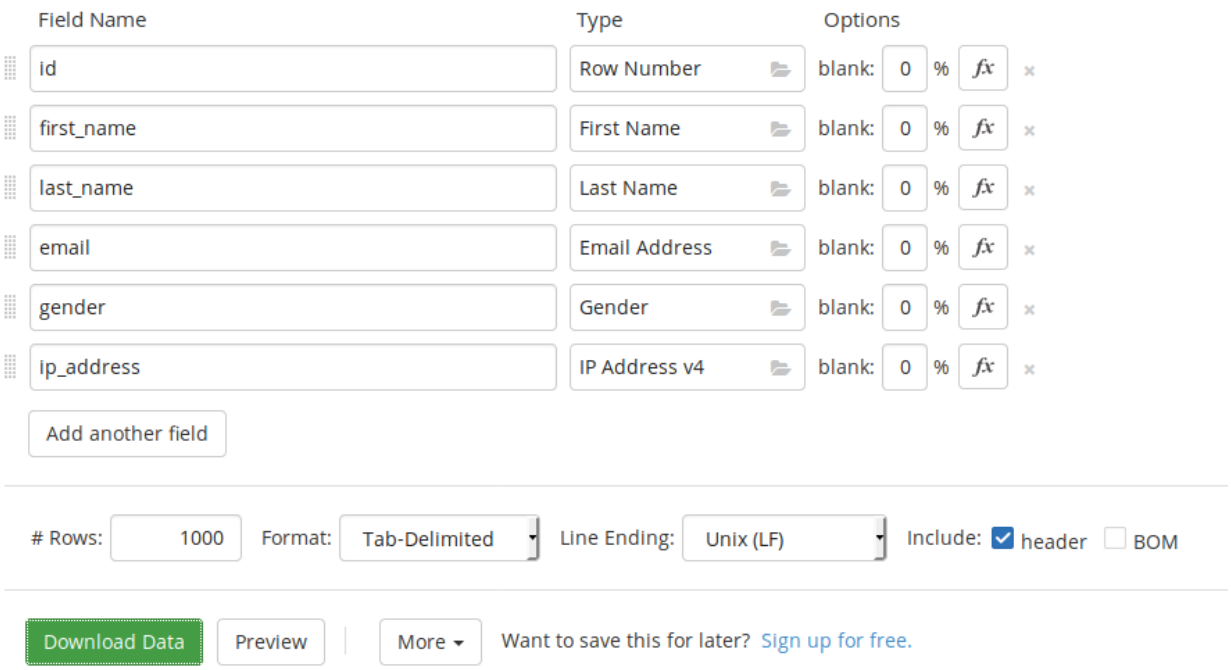
- `fs`: module pour travailler avec les fichiers et les répertoires
- `path`: module pour travailler avec des chemins d'accès « cross platform »
- `querystring`: module pour parser les HTTP Query Strings
- `net`: module réseau pouvant travailler avec différents protocoles
- `stream`: module pour travailler avec des « data streams »
- `events`: module pour implementer des « event emitters » (pattern Node observer)
- `child_process`: module pour gérer des processus externes
- `os`: module permettant d'accéder à des données systèmes
- `url`: module pour parser des URL
- `http`: module pour gérer des requêtes HTTP (client et serveur)
- `https`: module équivalent au module « http », mais pour le protocole HTTPS
- `util`: utilitaires divers
- `assert`: module de gestion des assertions (pour tests unitaires)
- `crypto`: module pour le cryptage/décryptage de données

3.3 Manipulation de fichiers

Etudier la manipulation de fichiers avec Node.js va nous donner l'occasion d'étudier son système d'E/S non bloquantes, de voir quels sont ses avantages mais aussi ses inconvénients.

3.3.1 Les fichiers texte

Si vous n'avez pas de fichier texte intéressant à manipuler sous la main, vous pouvez en créer un en vous servant du générateur de données proposé par le site Mockaroo.com. Vous pouvez vous en servir pour générer des fichiers textes simples, comme dans l'exemple ci-dessous où j'ai démandé un format « tab-delimited ». Vous pourrez vous en servir aussi dans le chapitre suivant, pour générer les fichiers CSV dont nous aurons besoin pour nos tests :



Je vous recommande de créer un premier fichier de petite taille (par exemple de 100 lignes), que vous appellerez « data_small.txt ». Créez également un second fichier en prenant le maximum de données que Mockaroo peut générer (1000 lignes) et en dupliquant ces données plusieurs fois avec d'avoir un volume significatif. Appelez ce second fichier « data_big.txt ».

Commençons par un premier script de lecture de fichier texte :

```
const FS = require('fs');
FS.readFile('data_small.txt', (err, data) => {
    if (err) {
        throw err;
    }
    console.log(data.toString());

    // data étant un objet, on le convertir en Chaîne, puis en tableau...
    let test = String(data).split('\n');
    // ... avant de récupérer le nombre de lignes
    console.log('Nombre de lignes :' + test.length);
});
```

Si vous n'avez pas fait d'erreur de syntaxe, vous devriez voir le contenu du fichier texte s'afficher dans votre console… avec en prime le nombre de lignes du fichier texte (le mien contient 102 lignes).

Si après avoir lu le contenu de votre premier fichier, vous voulez en créer une copie dans un second fichier, vous pouvez écrire ceci :

```
const FS = require('fs');
FS.readFile('data_small.txt', (err, data) => {
    if (err) {
        throw err;
    }
    console.log(data.toString());

    // data étant un objet, on le convertir en Chaîne, puis en tableau...
    let test = String(data).split('\n');
    // ... avant de récupérer le nombre de lignes
    console.log('Nombre de lignes :' + test.length);

    FS.writeFile('data_target.txt', data, (err) => {
        if (err) {
            throw err;
        }
        console.log('File saved!');
    });
});
```

C'est cool, ça marche, mais je ne vous recommande pas de faire ce genre de choses sur un gros fichier. Enfin si, soyons joueurs, testons ce même script sur notre gros fichier (data_big.txt). Le mien fait à peu près 1 million de lignes… Je crois les doigts et je lance l'exécution du script…

Ca marche mais il s'écoule quelques secondes avant que je ne voie s'afficher le nombre de lignes.

On notera qu'il est possible de charger des fichier en mode synchrone via la fonction « readFileSync ». Cette technique est à réserver au chargement de petits fichiers, sauf cas particulier (comme des reprises de données, qui sont généralement effectuées sur des environnements de développement et non de production). La fonction readFileSync est très simple d'utilisation :

```
var csv = fs.readFileSync('./mydata.csv');
```

Pour la lecture et l'écriture de gros fichiers on recommandera l'utilisation des fonctions dédiées à la manipulation de fichiers streams. La documentation relative à ce sujet se trouve ici :

https://nodejs.org/api/stream.html#stream_class_stream_readable

Voici un exemple :

```
const FS = require('fs');
var contents = '';

const RS = FS.createReadStream("data_big.txt");
RS.on('readable', function () {
    var buffer;
    var data = RS.read();
    if (data) {
        if (typeof data === 'string') {
            buffer = data;
        } else if (typeof data === 'object' && data instanceof Buffer) {
            buffer = data.toString('utf8');
        }
        if (buffer) {
            contents += buffer;
        }
    }
});
RS.on('end', function () {
    console.log("read in the file contents: ");
    console.log(contents);
});
```

La gestion des accès fichiers est un sujet qui nécessite une certaine pratique. Elle est moins intuitive ici que dans d'autres langages, du fait de la présence de ce mécanisme des E/S non bloquantes. De plus, la gestion des fichiers streams a subi quelques changements au fil des évolutions de Node.js, aussi pour une présentation assez exhaustive du sujet, je recommande la lecture des ces 2 articles de Scott Robinson :
http://stackabuse.com/read-files-with-node-js/
http://stackabuse.com/node-http-servers-for-static-file-serving/

3.3.2 Les fichiers CSV

Le package ya-csv est dédié à la lecture et à l'écriture de fichiers CSV.

Il est facile à utiliser et s'installe avec la commande :

```
npm install ya-csv
```

Voici un premier exemple de script dans lequel on crée un fichier CSV à partir d'un tableau.

```
const CSV = require('ya-csv');

var writer = CSV.createCsvFileWriter('data.csv');
var data = [];

// ligne 1 (4 fables)
data.push(['le corbeau et le renard', 'la cigale et la fourmi',
        'le loup et le chien', 'l\'ivrogne et sa femme']);

// ligne 2 (4 autres fables)
data.push(['le lièvre et la perdrix', 'le lion amoureux',
        "le loup et l'agneau", 'l\'ours et l\'amateur des jardins']);


data.forEach(function (record) {
    writer.writeRecord(record);
});
```

Le fichier CSV final généré par le script :

```
"le corbeau et le renard","la cigale et la fourmi","le loup et le chien","l'ivrogne et sa femme"
"le lièvre et la perdrix","le lion amoureux","le loup et l'agneau","l'ours et l'amateur des
jardins"
```

Autre exemple emprunté à la documentation de ce package :

```
const CSV = require('ya-csv');

var reader = CSV.createCsvFileReader('data.csv', {

    'separator': ',',
    'quote': '"',
    'escape': '"',
    'comment': ''
});

var writer = new CSV.CsvWriter(process.stdout);
reader.addListener('data', function(data) {
    // Affichage de la 1ère fable de chaque ligne
    writer.writeRecord([ data[0] ]);
});
```

Le package ya-csv propose également des fonctions dédiées à la manipulation de données en streaming. Voici un petit exemple adapté à la récupération dans un gros fichier :

```
const CSV = require('ya-csv');

var reader = CSV.createCsvFileReader('data.csv');

var data = [];

reader.on('data', function (record) {
    data.push(record);
}).on('end', function () {
    console.log(data);
});
```

Un autre exemple, dans lequel un serveur web lit un gros fichier CSV et renvoie son contenu dans la réponse HTTP :

```
const CSV = require('ya-csv');
const HTTP = require('http');

HTTP.createServer(function (req, res) {

    res.writeHead(200, {
        'Content-Type': 'text/plain; charset=utf-8'
    });

    res.write('[');
    CSV.createCsvFileReader('data.csv')
            .on('data', function (data) {
                res.write(((this.parsingStatus.rows > 0) ? ',' : '') +
                        JSON.stringify(data));
            }).on('end', function () {
        res.end(']');
    });
}).listen(8080);
```

Le package ya-csv est très fonctionnel, relativement bien documenté, et il peut rendre de nombreux services :

https://www.npmjs.com/package/ya-csv

3.4 Connexion à MySQL et MariaDB

3.4.1 Les bases

Pour le sujet qui suit, je pars du principe que vous avez déjà installé MySQL ou MariaDB sur votre PC. Si ce n'est pas le cas et si vous vous interrogez sur la manière de procéder, je vous invite à vous reporter au support de cours qui se trouve dans ce dépôt :
https://github.com/gregja/SQLCorner/

Si vous avez déjà une base MySQL ou MariaDB installée, assurez-vous qu'elle est bien démarrée avant de poursuivre la lecture de ce qui suit.

La base de données n'est pas suffisante, il nous faut aussi installer un package Node.js dédié à l'interfaçage de Node.js avec cette base de données. Peut être n'avez-vous pas encore installé le package « mysql2 » que nous avions évoqué pendant notre introduction aux packages. Si c'est le cas, je vous invite à installer ce package via la commande suivante :

```
npm install mysql2 --save
```

Vous aurez peut être remarqué qu'il existe plusieurs packages pour MySQL et MariaDB, et peut être vous interrogez-vous sur la raison pour laquelle j'ai retenu « mysql2 » plutôt qu'un autre package. Je reviendrai sur cette épineuse question dans un prochain chapitre. Pour l'instant je préfère me concentrer sur le principe de fonctionnement de ce package « mysql2 ».

Le script de connexion étant un peu long, je l'ai placé sur la page suivante pour ne pas le couper.

Voici le script - développé sous la forme d'un module - permettant de se connecter à MySQL et à MariaDB.

Vous veillerez à personnaliser les paramètres de connexion « password » et « database » :

```
const MYSQL = require('mysql2');

module.exports = {
    init: function () {

        var conn = MYSQL.createConnection({
            host: "localhost",
            user: "root",
            password: "XXXXXXX",
            database: "DDDDDDD",
            charset: "UTF8_GENERAL_CI"
        });

        conn.connect(function (err) {
            if (err) {
                console.error('ERREUR SUR CONNEXION: ' + err.stack);
                return;
            }

            console.log('Connected as id ' + conn.threadId);

        });

        return conn;
    }

};
```

Après quelques recherches dans la documentation, il apparaît que l'appel de la fonction « conn.connect() » est facultatif, car le premier appel de la fonction conn.query() déclenche une connexion implicite. Mais il est néanmoins recommandé d'utiliser la fonction conn.connect() afin de pouvoir mieux monitorer l'anomalie de connexion si elle se présente (pour cause de mot de passe invalide par exemple).

Je profite de l'occasion pour rappeler que, avec MySQL, il est vivement recommandé de paramétrer le client SQL en UTF-8 (d'où la présence du paramètre « charset »).

Il serait judicieux de pouvoir renvoyer aux scripts « consommateurs » de ce module une information permettant de savoir si l'objet « conn » est bien une connexion valide. Cette amélioration pourra être intégrée à une version ultérieure du module de connexion.

Créez maintenant un script de test, appelez-le comme vous voulez, et placez-y le code suivant :

```
const dbmodule = require('./dbconnex');

conn = dbmodule.init();

conn.query('SELECT 1 + 1 AS solution', function (err, results, fields) {
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
        return;
    }
    console.log('La solution est : ', results[0].solution);
});

conn.end();
```

Exécutez ce script dans une console, vous devriez voir apparaître le message suivant :

```
Connected as id : xx
La solution est : 2
```

Ca marche ! Vous disposez maintenant d'un module personnalisé pour la connexion à MySQL et à MariaDB. Vous allez pouvoir l'utiliser dans les scripts des prochains chapitres.

3.4.2 Manipulation d'objets SQL

Je vous propose d'étudier un exemple de script ayant pour action de créer et d'alimenter une table SQL. Ce n'est sans doute pas une technique que vous utiliserez tous les jours, mais cela pourrait être utile si vous avez besoin de déployer une base de données (ou de la migrer dans une nouvelle version), au moyen d'un script facilement réutilisable et automatisable.

Pour les besoins de l'exemple, j'ai réutilisé le module de connexion que nous avons créé au chapitre précédent. Et j'ai emprunté le CREATE TABLE et la requête INSERT à un exemple proposé par W3Schools : https://www.w3schools.com/nodejs/nodejs_mysql_create_table.asp
A noter que l'exemple proposé par W3Schools ne fonctionne pas avec Node.js en version 9, du fait de petites modifications intervenues sur le module « mysql2 », et qui impactent la manière de monitorer les erreurs.

L'exemple ci-dessous fonctionne lui sans problème, je vous invite à l'essayer :

```
const dbmodule = require('./dbconnex');

conn = dbmodule.init();

/* Create a table named "customers": */
var sql = "CREATE TABLE customers (name VARCHAR(255), address VARCHAR(255))";
conn.query(sql, function (err, result) {
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
        return;
    }
    console.log("Table created");
});

console.log("Connected!");
var sql = "INSERT INTO customers (name, address) VALUES ?";
var values = [
    ['John', 'Highway 71'],
    ['Peter', 'Lowstreet 4'],
    ['Amy', 'Apple st 652'],
    ['Hannah', 'Mountain 21'],
    ['Michael', 'Valley 345'],
    ['Sandy', 'Ocean blvd 2']
];
conn.query(sql, [values], function (err, result) {
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
        return;
    }
    console.log("Number of records inserted: " + result.affectedRows);
});

conn.end();
```

Vous avez remarqué de quelle manière se fait l'INSERT, avec le tableau « values » ? Difficile de faire plus simple, pas vrai ?

Exécutez le script dans une console, vous devriez voir apparaître les messages suivants :

```
Table created
Number of records inserted : 6
```

Vous noterez que nous avons obtenu le nombre de lignes insérées en utilisant la propriété « result.affectedRows ». Cette même propriété est alimentée aussi pour les requête de type DELETE.
Cet objet « result » est très intéressant car il nous permet de récupérer les informations ci-dessous après exécution de la requête :

```
{
  fieldCount: 0,
  affectedRows: 6,
  insertId: 0,
  serverStatus: 2,
  warningCount: 0,
  message: '\'Records:6  Duplicated: 0  Warnings: 0',
  protocol41: true,
  changedRows: 0
}
```

La propriété « affectedRows » comptabilise le nombre de lignes impactées, qu'il s'agisse de mise à jour, insertion ou suppression.
La propriété « changedRows » comptabilise les mises à jour réalisées par une requête de type UPDATE. Mais attention, ne sont comptabilisées que les mises à jour effectives (si certaines lignes n'ont pas été modifiées car elles contenaient déjà les données indiquées dans l'UPDATE, alors elle ne seront pas comptabilisées).
Si nous avions inséré des lignes dans une table possédant un identifiant en incrémentation automatique (ce n'est pas le cas dans notre exemple), nous aurions pu récupérer la valeur du dernier identifiant au moyen de la propriété « insertId ». Ce genre d'information pourra être utile dans d'autres circonstances.

Le procédé d'insertion que nous venons de voir est certes pratique, mais il présente cependant des limites. Ce serait en effet pratique de pouvoir injecter des données provenant d'un tableau d'objets, comme celui-ci :

```
var sql = "INSERT INTO customers (name, address) VALUES ?";
var values = [
    {name:'John', address:'Highway 71'},
    {name:'Peter', address:'Lowstreet 4'},
    {name:'Amy', address:'Apple st 652'},
    {name:'Hannah', address:'Mountain 21'},
    {name:'Michael', address:'Valley 345'},
    {name:'Sandy', address:'Ocean blvd 2'}
];
conn.execute(sql, [values], function (err, result, fields) {
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
```

```
        return;
    }
    console.log("Number of records inserted: " + result.affectedRows);
});
```

Malheureusement cela ne fonctionne pas. Nous devrons donc recourir à une autre technique, que nous étudierons au travers du chapitre suivant.

3.4.3 Quel package utiliser pour MySQL et MariaDB ?

Vous aurez sans doute remarqué qu'il existe un package « mysql » et un package « mysql2 » :

https://www.npmjs.com/package/mysql
https://www.npmjs.com/package/mysql2

Il existe aussi d'autres packages faisant référence à MySQL ou à MariaDB, mais ils ne bénéficient pas d'un support aussi actif que les 2 packages ci-dessus (certains projets semblent même carrément à l'arrêt).

Le package « mysql » est pour ainsi dire le package officiel pour utiliser MySQL et MariaDB dans Node.js. Il est aussi connu sous le nom de « mysqljs », son dépôt Github est d'ailleurs « mysqljs/mysql ». Il est toujours maintenu, mais il souffre de certaines limitations comme nous allons le voir dans un instant.

Le package « mysql2 » est développé par une équipe concurrente. Il offre quelques fonctionnalités supplémentaires par rapport au package « mysql », et les développeurs de « mysql2 » ont eu la bonne idée de conserver les mêmes appels de fonctions que ceux existant dans « mysql ». On peut donc passer de « mysql » à « mysql2 » sans problème.

A noter que la documentation du package « mysql » est sur certains sujets plus complète que celle de « mysql2 » aussi pour des questions d'ordre général, n'hésitez pas à la consulter.

Mais quels sont les avantages de « mysql2 » qui m'incitent à passer sur ce package ? La documentation de « mysql2 » indique ceci :

MySQL2 is mostly API compatible with mysqljs and supports majority of features. MySQL2 also offers these additional features

- *Faster / Better Performance*
- *Prepared Statements*
- *MySQL Binary Log Protocol*
- *MySQL Server*
- *Extended support for Encoding and Collation*
- *Promise Wrapper*
- *Compression*
- *SSL and Authentication Switch*
- *Custom Streams*

- *Pooling*

Ainsi le package « mysql2 » offre quelques fonctionnalités très alléchantes, et en particulier les « prepared statements ». Il s'agit de requêtes SQL préparées selon un mécanisme qui offre une excellente sécurité contre les attaques par injection SQL (et aussi de très bonnes performances si on sait l'utiliser correctement).

La technique du « prepared statement » proposée par le package « mysql2 » est décrite ici : https://www.npmjs.com/package/mysql2#using-prepared-statements

Voici le code de l'exemple proposé dans la documentation :

```
// get the client
const mysql = require('mysql2');

// create the connection to database
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  database: 'test'
});

// execute will internally call prepare and query
connection.execute(
  'SELECT * FROM `table` WHERE `name` = ? AND `age` > ?',
  ['Rick C-137', 53],
  function(err, results, fields) {
    console.log(results); // => lignes renvoyées par la requête
    console.log(fields); // => métadonnées relatives au contenu de results
    // If you execute same statement again, it will be picked form a LRU cache
    // which will save query preparation time and give better performance
  }
);
```

Les informations essentielles à mon avis sont celles indiquées dans les commentaires suivants :

```
// execute will internally call prepare and query
// If you execute same statement again, it will be picked form a LRU cache
// which will save query preparation time and give better performance
```

Dans d'autres langages, comme par exemple PHP avec la librairie PDO, il est possible de créer un statement grâce à la méthode « prepare ». Ce « statement » pourra être utilisé ensuite pour effectuer plusieurs « execute » si nécessaire. J'explique cette technique dans le cours PHP (cf. https://github.com/gregja/PhpCorner), elle permet d'obtenir d'excellentes performances dans le cas où l'on souhaite injecter un grand nombre de lignes dans une table. On désigne souvent cette technique par le terme de « bulk insert ».

Le fait de ne pas pouvoir déclarer de « prepared statement » avec le package « mysql » nous encourage à utiliser de préférence le package concurrent « mysql2 ». L'équipe de développement du package

« mysql » a été interpellée à plusieurs reprises sur ce problème, peut être parviendra-t-elle à le résoudre dans une version future du package :
https://github.com/sidorares/node-mysql2/issues/234

Cette présentation étant faite, nous allons voir dans les chapitres suivants des exemples de requêtes d'insertion, mise à jour et suppression. J'ai privilégié dans ces exemples la technique du « prepared statement », pour ses avantages indéniables (sécurité et performances).

3.4.4 INSERT avec prepared statement

Voici un exemple de script constitué de 2 étapes :
1 – le vidage de la table SQL des « customers »
2 – le remplissage de la table à partir d'un tableau d'objets, via une requête SQL préparée

Le code :

```
const dbmodule = require('./dbconnex');
const conn = dbmodule.init();

var sql = "DELETE FROM customers";
conn.query(sql, function (err, result) {
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
        return;
    }
    console.log("Number of records deleted: " + result.affectedRows);
});
console.log("Connected!");
var sql = "INSERT INTO customers (name, address) VALUES (?, ?)";

var datas = [
    {name:'John', address:'Highway 71'},
    {name:'Peter', address:'Lowstreet 4'},
    {name:'Amy', address:'Apple st 652'},
    {name:'Hannah', address:'Mountain 21'},
    {name:'Michael', address:'Valley 345'},
    {name:'Sandy', address:'Ocean blvd 2'},
    {name:'Betty', address:'Green Grass 1'},
    {name:'Richard', address:'Sky st 331'},
    {name:'Susan', address:'One way 98'},
    {name:'Vicky', address:'Yellow Garden 2'},
    {name:'Ben', address:'Park Lane 38'},
    {name:'William', address:'Central st 954'},
    {name:'Chuck', address:'Main Road 989'},
    {name:'Viola', address:'Sideway 1633'}
];
```

```
let i = 0, max = datas.length;
while (i < max) {
    let item = datas[i++];

    conn.execute(sql, [item.name, item.address], function (err, result) {
        if (err) {
            console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
            return;
        }
        console.log("Number of records inserted: " + result.affectedRows);
    });
}

conn.end();
```

3.4.5 DELETE avec prepared statement

Voici un exemple relativement simple, avec suppression de 2 clients, à partir d'un jeu de données contenu dans un tableau. Je vous laisse le soin de l'étudier :

```
const dbmodule = require('./dbconnex');

const conn = dbmodule.init();

var sql = "DELETE FROM customers WHERE name = ? AND address = ?";

// tableau des clients à supprimer
var datas = [
    {name:'John', address:'Highway 71'},
    {name:'Peter', address:'Lowstreet 4'}
];

let i = 0, max = datas.length;
while (i < max) {
    let item = datas[i++];

    conn.execute(sql, [item.name, item.address], function (err, result) {
        if (err) {
            console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
            return;
        }
        console.log("Number of records deleted: " + result.affectedRows);
    });
}

conn.end();
```

3.4.6 UPDATE avec prepared statement

Et pour finir voici un autre exemple relativement simple, avec la mise à jour des adresses de 2 clients, toujours à partir de données stockées dans un tableau. Je vous laisse aussi le soin de l'étudier :

```
const dbmodule = require('./dbconnex');

const conn = dbmodule.init();

var sql = "UPDATE customers SET address = ? WHERE name = ?";

// tableau des nouvelles adresses
var datas = [
    {name:'Sandy', address:'Railroad St'},
    {name:'Betty', address:'Brooklin'}
];

let i = 0, max = datas.length;
while (i < max) {
    let item = datas[i++];

    // attention à l'ordre des paramètres par rapport aux ? de la requête
    conn.execute(sql, [item.address, item.name], function (err, result) {
        if (err) {
            console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
            return;
        }
        console.log("Number of records concerned: " + result.affectedRows);
        console.log("Number of records really updated: " +
                        result.changedRows);
    });
}

conn.end();
```

J'avais présenté dans un précédent chapitre, la signification des propriétés de l'objet « result », mais je crois utile de les rappeler ici :
- La propriété « affectedRows » comptabilise le nombre de lignes impactées, qu'il s'agisse de mise à jour, insertion ou suppression.
- La propriété « changedRows » comptabilise les mises à jour réalisées par une requête de type UPDATE. Mais attention, ne sont comptabilisées que les mises à jour effectives (si certaines lignes n'ont pas été modifiées car elles contenaient déjà les données indiquées dans l'UPDATE, alors elle ne seront pas comptabilisées).

Si vous lancez plusieurs fois le script ci-dessus, vous pourrez observer le contenu de ces 2 propriétés, au travers des appels à la fonction « console.log() ».

3.4.7 SELECT avec prepared statement

On ne peut pas clore ce sujet sans parler du SELECT, et de la manière de l'exécuter dans un « prepared statement ». C'est relativement simple, comme vous allez le voir :

```
const dbmodule = require('./dbconnex');

const conn = dbmodule.init();

var sql = "SELECT * FROM customers WHERE name = ? OR address = ?";

var data1 = 'Sandy';
var data2 = 'Apple st 652';

// attention à l'ordre des paramètres par rapport aux ? de la requête
conn.execute(sql, [data1, data2], function (err, rows, result) {
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
        return;
    }

    for (var i in rows) {
        console.log( rows[i].name, ' => ', rows[i].address );
    }

});

conn.end();
```

La boucle « for » à la fin du script est importante, puisque c'est elle qui va nous permettre de récupérer le jeu de données (en anglais : « dataset » ou encore « resultset ») produit par la requête SELECT.

Dans le cas de notre script, nous allons obtenir le résultat suivant :

```
Amy => Apple st 652
Sandy => Railroad St
```

Il existe une alternative à l'écriture précédente qui permet d'aboutir au même résultat.
Je la trouve intéressante, d'autant qu'elle est compatible avec la fonction « execute » et donc avec les prepared statements :

```
const dbmodule = require('./dbconnex');

const conn = dbmodule.init();

var sql = "SELECT * FROM customers WHERE name = ? OR address = ?";

var data1 = 'Sandy';
var data2 = 'Apple st 652';

var query = conn.execute(sql, [data1, data2]);

query.on('error', function(err) {
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
        return;
    }
});

query.on('fields', function(fields) {
    // affichage des infos SQL (à activer en cas de besoin)
    // console.log(fields);
});

query.on('result', function(row) {
    // Affichage des données provenant du SELECT
    console.log(row.name, ' => ', row.address);
});

conn.end();
```

Vous préférerez sans doute un des deux styles d'écriture. L'approche ci-dessus a le mérite de mieux structurer les choses.

3.4.8 Procédures stockées

Une manière efficace et intelligente d'écrire du code « legacy » performant, c'est de l'encapsuler dans des procédures stockées. J'avais traité ce sujet dans le support de cours dédié à SQL :

https://github.com/gregja/SQLCorner/

... et je vous invite à vous y reporter pour de plus amples précisions sur ce sujet.

Je dois dire que ce chapitre sur les procédures stockées m'aura donné du fil à retordre, car ce sujet est très mal documenté par les développeurs des packages « mysql » et « mysql2 ». De plus, les quelques exemples trouvés sur internet étaient souvent trop basiques et ne couvraient pas la question des paramètres de type OUT (sortie) et INOUT (entrée/sortie). J'ai dû pas mal défricher pour déterminer ce qui fonctionne, et ce qui ne fonctionne pas.

Nous allons commencer par un cas simple qui fonctionne à tous les coups : la procédure stockée avec des paramètres de type IN (entrée). On se penchera ensuite sur les cas plus épineux...

Je vous propose donc de commencer par la procédure stockée ci-dessous, que je vous invite à créer dans votre client SQL préféré (phpMyAdmin ou autre) :

```
DELIMITER $$
CREATE PROCEDURE my_proc_test(IN prm1 INT, IN prm2 INT)
BEGIN
  DECLARE x INT DEFAULT 0;
  DECLARE y INT DEFAULT 0;
  SET x = 15 + prm1;
  SET y = 10 + prm2;
  SELECT x, y, x-y as result
  UNION
  SELECT x+1 as x, y-1 as y, (x+1)-(y-1) as result;
END$$
```

Si vous exécutez cette procédure dans votre client SQL via un CALL :

call `my_proc_test(2, 4)`

... Vous devriez obtenir ceci :

x	y	result
17	14	3
18	13	5

Vous noterez que je n'ai pas défini de paramètre de sortie de la procédure, pourtant MySQL renvoie par défaut le contenu du dernier SELECT effectué à l'intérieur de la procédure. Ce n'est pas le comportement adopté par d'autres SGBD (comme PostgreSQL et DB2), mais puisque cela fonctionne ainsi dans MySQL (et dans MariaDB), cela nous évite de trop complexifier le code de la procédure.

Passons maintenant à l'écriture du script Node.js qui va nous permettre de lancer cette procédure.

```
const dbmodule = require('./dbconnex');

const conn = dbmodule.init();

var sql = "CALL my_proc_test (? , ?);";

var data1 = 2;
var data2 = 4;

// attention à l'ordre des paramètres par rapport aux ? de la requête
conn.execute(sql, [data1, data2], function (err, rows, result) {
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
        return;
    }

    for (var i in rows[0]) {
        let item = rows[0][i];
        console.log( item.x, item.y, item.result );
    }
});

conn.end();
```

Exécutez ce script, vous devriez obtenir le même résultat que tout à l'heure, à savoir :

x	y	result
17	14	3
18	13	5

Franchement, c'est plutôt encourageant tout ça. Mais notre procédure stockée n'accepte que dans des paramètres de type IN (entrée). Ce serait bien de pouvoir utiliser des paramètres de type OUT (sortie) et INOUT (entrée/sortie). En PHP, notamment avec la librairie PDO, on utilise une technique appelée « bind

param » pour gérer les paramètres de type OUT et INOUT. Or, ni la documentation du package « mysql », ni celle du package « mysql2 » ne mentionnent cette possibilité.

Il existe néanmoins une possibilité, que j'ai trouvée expliquée sur Github, dans les « issues » du package « mysql » et qui fonctionne aussi avec le package « mysql2 » (j'ai testé pour vous).

L'explication fournie par Andrey Sidorov, un des développeurs du package « mysql » se trouve ici :

https://github.com/mysqljs/mysql/issues/682

Je vous copie-colle ci-dessous l'exemple de code fourni par Andry Sidorov, je l'ai légèrement remanié pour en améliorer la lisibilité :

```
var mysql = require('mysql2');  //=> fonctionne avec mysql et mysql2
var opts = {
    host: "localhost",
    user: "root",
    password: "XXXXXXX",
    database: "DDDDDDD",
    multipleStatements: true
};

var sql_cre_proc = `DROP PROCEDURE IF EXISTS pp;
 CREATE PROCEDURE pp (INOUT i INT, IN j INT)
 BEGIN
   SET i = 10 + j;
 END;`;

var sql_exe_proc = `SET @test = 1;
 CALL pp(@test, 123);
 SELECT @test as inout_i`;

var pool = mysql.createPool(opts);
pool.getConnection(function (err, conn) {
    // création de la procédure de test
    conn.query(sql_cre_proc);

    // test de la procédure
    conn.query(sql_exe_proc, function (err, rows) {
        if (err) {
            console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
            return;
        }
        for (var i in rows[2]) {
            console.log(rows[2][i].inout_i);
        }

    });
});
```

Voilà pour cette technique, qui nécessite de passer par un pool de connexion et d'utiliser les fonctions « createPool » et « getConnection ». Mis à part, le principe reste assez proche de ce qu'on avait vu précédemment, mais on notera qu'il est impossible d'utiliser la méthode « execute » propre aux « prepared statement ». Les procédures étant naturellement protégées contre les attaques par injection SQL, ce problème n'en est peut être pas un. A vous de voir si vous préférez utiliser la technique ci-dessus, ou la technique précédente (qui permet aussi de récupérer des données en sortie, mais sans passer par les paramètres OUT et INOUT).

3.4.9 Pagination et clause LIMIT

On ne le dira jamais assez, mais balancer dans une page web des centaines de lignes de tableaux HTML (générés en général à partir de jeux de données provenant de SQL), et déléguer au navigateur le soin de gérer la pagination de toutes ces données au moyen d'un plugin quelconque… c'est une très mauvaise manière de travailler car :

- on surcharge les serveurs bases de données et les serveurs webs avec des données inutiles dont les utilisateurs n'ont que faire,

- on surconsomme de la bande passante sur les réseaux sur lesquels les données transitent

- on surconsomme de la CPU et on sature la mémoire du client avec des données inutiles que ce dernier doit pourtant traiter, afin de proposer à l'internaute un semblant de pagination. Et c'est particulièrement dommageable si le client est un navigateur embarqué sur un smartphone

- l'expérience de navigation de l'internaute est dégradée par les mauvaises performances, si c'est un consommateur potentiel, on risque de le perdre

Ce n'est pourtant pas compliqué de mettre en place une pagination s'appuyant sur la clause LIMIT de MySQL, comme dans les exemples suivants :

- affichage de 10 lignes à partir de la toute première du jeu de données (ligne zéro)

```
SELECT * FROM matable WHERE … LIMIT 0, 10
```

- technique équivalente utilisant la clause OFFSET :

```
SELECT * FROM matable WHERE … LIMIT 10 OFFSET 0
```

ATTENTION : la pagination avec la clause LIMIT peut rendre beaucoup de services, mais elle ne suffit pas à rendre une requête SQL plus rapide, si cette dernière est particulièrement complexe (avec beaucoup de jointure ou de sous-requêtes), ou faiblement optimisée (indexs manquants). La durée d'exécution de la requête risque d'être trop longue pour les utilisateurs, clause LIMIT ou pas. Nous aborderons ce sujet dans un chapitre ultérieur (TODO : sujet à traiter ultérieurement)

Nous verrons une mise en application de ce principe dans un prochain chapitre (TODO: indiquer la référence au chapitre dès qu'il sera prêt).

3.4.10 Monitoring

Nous n'avons pas regardé ce qui se passe quand une erreur SQL se produit.

Dans les exemples de code que je vous ai présenté jusqu'ici, je me suis contenté pour la gestion des erreurs, du code minimaliste suivant :

```
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                "  (" + err.message + ")");
        return;
    }
```

Nous envoyons donc dans la log des propriété (code et message) de l'objet « err ». Mais nous pouvons récupérer deux autres informations intéressantes qui sont :

- le numéro d'erreur propre à MySQL
- le SQL State, un code à 5 chiffres presque conforme à la norme SQL (« presque », car tous les codes définis dans la norme SQL ne sont pas implémentés dans MySQL).
On peut consulter la liste des SQL State de MySQL sur la page suivante :
https://dev.mysql.com/doc/refman/5.5/en/error-messages-server.html

Je vous propose de reprendre un script d'un chapitre précédent, et de modifier légèrement le mécanisme d'affichage des erreurs. Nous allons également déclencher une erreur volontaire en introduisant un nom de table erronée dans la requête SQL :

```
const dbmodule = require('./dbconnex');

const conn = dbmodule.init();

var sql = "SELECT * FROM Xustomers WHERE name = ? OR address = ?";

var data1 = 'Sandy';
var data2 = 'Apple st 652';

var query = conn.execute(sql, [data1, data2]);

query.on('error', function(err) {
    console.log('SQL code erreur :', err.code);
    console.log('SQL num. erreur :', err.errno);
    console.log('SQL State :', err.sqlState);
    console.log('SQL Msg :', err.sqlMessage);
    // throw err;
});

query.on('fields', function(fields) {
```

```
    // affichage des infos SQL (à activer en cas de besoin)
    // console.log(fields);
});

query.on('result', function(row) {
    // Affichage des données provenant du SELECT
    console.log(row.name, ' => ', row.address);
});

conn.end();
```

Les données récupérées dans la console sont les suivantes :

```
SQL code erreur : ER_NO_SUCH_TABLE
SQL num. erreur : 1146
SQL State : 42S02
SQL Msg : Table 'test.Xustomers' doesn't exist
```

Ces informations sont conformes aux spécifications de la norme SQL, vous les retrouverez sur la plupart des bases de données.

Cela peut être une très bonne chose que d'envoyer ces informations dans la console à chaque fois qu'une erreur SQL se produit. Mais ce serait encore mieux de les archiver dans un fichier de log, pour en faciliter la consultation ultérieure. C'est pourquoi il pourra être intéressant de concevoir une fonction générique pour le monitoring des erreurs SQL, fonction qui recevra en entrée l'objet référençant l'erreur, et en extraira les informations essentielles, pour les envoyer dans un fichier de log.

3.5 Express

3.5.1 Présentation et installation

Le package Express est un véritable framework, qui simplifie le développement d'applications webs et décharge le développeur d'un certain nombre de tâches.

Nous avons fait beaucoup d'essais dans notre répertoire de travail. A ce stade, il est sans doute judicieux de laisser de côté ce répertoire, et d'en créer un nouveau. Placez-vous dans ce nouveau répertoire et saisissez la commande :

```
npm init -y
```

... puis la commande

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

Vous vous souvenez sans doute que nous avons créé précédemment des serveurs webs dont le code source ressemblait grosso modo à ceci :

```
const HTTP = require("http");

function process_request(req, res) {
    res.end("Hello World");
}

var s = HTTP.createServer(process_request);
s.listen(8080);
```

Un serveur Express, eh bien cela va ressembler plutôt à ça :

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

var app = EXPRESS();
app.get('/', function (req, res, next) {
    res.end('hello world');
});
app.listen(8080);
```

Saisissez ce code dans un script que vous appellerez « server_01.js », puis lancez son exécution :

```
node server_01.js
```

Allez sur votre navigateur préféré et saissisez l'url suivante :

http://localhost:8080

Vous voilà sur un nouvel « hello world », cette fois à la « sauce Express ».

3.5.2 Les routes en théorie

Express est bâti sur le module « connect », un module qui embarque un certain nombre de fonctionnalités qui vont nous être utiles au quotidien.

Notre serveur web minimaliste embarque pour l'instant une seule couche de logique applicative, avec l'appel de la fonction « get() ».

Notre serveur ne sait pour le moment gérer que des requêtes HTTP de type GET, et une seule route (ou chemin) matérialisé par le slash (/)

Express embarque un système de routage, matérialisé par le chemin indiqué sur chaque fonction app.get().

Le format général d'une URL de routage est le suivant :

```
app. method ( url_pattern , optional_functions ,
              request_handler_function );
```

Nous avons vu la méthode « get », le principe est le même avec la méthode POST :

```
app.post("/forms/update_product", function (req, res) { ... });
```

Express supporte aussi les requêtes HTTP déclarées avec les méthodes DELETE et PUT.

On peut aussi utiliser la méthode « all » si on souhaite qu'une même requête puisse être appelée avec n'importe quelle méthode :

```
app.all("/products/info", function (req, res) { ... });
```

Le premier paramètre de la méthode (get, post, delete, put, all), est une expression régulière qui peut prendre différentes formes, comme par exemple :

```
/product(s)?/list
```

… qui a pour effet d'accepter les deux syntaxes suivantes :

```
/products/list
```
ou
```
/product/list
```

… ou encore :

/products/*

… qui détecte tout ce qui commence par « products ».

Une des caractéristiques essentielles du routage, c'est la capacité à utiliser des paramètres dans la requête, matérialisés par « : », comme par exemple :

```
app.get("/products/:productid", function (req, res) {
   var prod_id = parseInt(req.params.productid);
   res.end(`Produit : ${prod_id}`);
});
```

On peut dès lors envoyer des requêtes telles que :

```
http://localhost:8080/products/10
http://localhost:8080/products/230
```

On peut aussi envoyer transmettre plusieurs paramètres dans une même requête :

```
app.get("/products/:productid/category/:categid", function (req, res) {
   var prod_id = parseInt(req.params.productid);
   var cat_id = parseInt(req.params.categid);
   res.end(`Produit : ${prod_id} ; Categorie : ${cat_id}`);
});
```

3.5.3 Les routes en pratique

Nous allons voir un exemple de script implémentant 3 types de routes :

- une route sans paramètre

- une route permettant de sélectionner un produit

- une route permettant de sélectionner un produit et une catégorie

Voici le code du script :

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

app.get('/', function (req, res, next) {
    res.end('hello world');
    console.log('arret / route 1');
    next();
});

app.get("/products/:productid", function (req, res, next) {
    var prod_id = parseInt(req.params.productid);
    res.end(`Produit : ${prod_id}`);
    console.log('arret / route 2');
});

app.get("/products/:productid/category/:categid", function (req, res, next) {
    var prod_id = parseInt(req.params.productid);
    var cat_id = parseInt(req.params.categid);
    res.end(`Produit : ${prod_id} ; Categorie : ${cat_id}`);
    console.log('arret / route 3');
});

app.listen(8080);
```

Pour l'instant, notre serveur Express renvoie du HTML un peu pourri, mais nous verrons cela plus tard.

Ce qui est important ici, c'est que vous compreniez bien comment sont définis les paramètres à l'intérieur de chaque route. C'est la raison pour laquelle je les ai mis en gras dans le code source ci-dessus.

Vous voyez que l'objet « req.params » est à peu près équivalent – d'un point de vue fonctionnel - au tableau associatif `$_GET` du langage PHP.

Pour bien comprendre ce qui se passe, lancez ce script avec la commande « node », et testez-le en lançant différentes requêtes dans votre navigateur, telles que :

```
http://localhost:8080/products/20/category/10
```

```
http://localhost:8080/products/20
```

```
http://localhost:8080/
```

… tout en faisant ces tests, observez en parallèle les messages que vous obtenez dans votre console.

Dans votre script en cours, à la suite des routes existantes, je vous invite à ajouter la route suivante :

```
app.get("/apropos", function (req, res, next) {
    res.sendfile('./apropos.html');
    console.log('arret / page "à propos" ');
});
```

Vous voyez que nous utilisons ici une méthode « sendfile » qui a pour particularité de charger et d'envoyer un fichier HTML statique, ce qui est pratique pour les pages qui n'ont pas de contenu dynamique.

Mais pour que cela fonctionne, il nous faut créer le fichier « apropos.html » dans le même répertoire que le script. Pour gagner du temps, je vous propose de copier-coller le bout de code HTML suivant :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>A propos</title>
        <meta charset="UTF-8">
    </head>
    <body>
        <div>Contenu de la page "à propos"</div>
    </body>
</html>
```

Relancez votre script serveur avec la commande « node », et saisissez l'url suivante dans votre navigateur :

```
http://localhost:8080/apropos
```

Dans un précédent chapitre, nous avions créé une petite fonction « template() », pour générer une page HTML bien structurée. Je vous propose une petite variante de cette fonction, que voici :

```
var template = (title='no title', divcontent='') => `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>${title}</title>
        <meta charset="UTF-8">
    </head>
    <body>
        <div>${divcontent}</div>
    </body>
</html>`;
```

Le seul changement par rapport à la version précédente, c'est l'ajout d'un paramètre « title » en entrée de la fonction, que nous insérons dans la balise « title » du template HTML.

Il ne nous reste plus qu'à intégrer cette fonction dans notre script serveur. Je vous propose de faire cet exercice par vous même, et de consulter ensuite la solution proposée en annexe (cf. chapitre 5.4).

3.5.4 Erreur 404

La documentation d'Express est bien faite, et en plus elle est en français.

Cette page sur les différentes techniques de routage est très intéressante, et je vous en recommande la lecture pour approfondir le sujet :

http://expressjs.com/fr/guide/routing.html

Curieusement, il y a une info que je n'ai pas trouvée sur cette page, c'est comment configurer Express pour déclencher une erreur 404 si aucune des routes prédéfinies ne correspond au contexte. J'ai trouvé une solution dans un forum, que je vous propose ci-dessous, car elle semble bien fonctionner dans mon cas. Il faut ajouter à la fin des routes (juste avant le « app.listen() »), le bloc suivant :

```
app.use(function(req, res, next){
    res.status(404);
    res.end(template(
            'Erreur 404',
            'Désolé, page non trouvée'
    ));
});
```

Après avoir ajouté cette dernière route dans votre script, je vous invite à faire des tests en saisissant des url valides et des url invalides.

Attention, si vous implémentez le code ci-dessus, je vous recommande de ne pas utiliser de route avec « astérisque », telle que celle ci-dessous :

```
app.get("/products/*", function (req, res, next) {
    // do what you want here
});
```

… car une telle route pourrait empêcher l'accès au bloc de gestion de l'erreur 404. Je ne rentre pas dans les détails, mais je vous invite à tester par vous même, pour vous faire une idée du problème.

3.5.5 Routes et formulaire

Dans le chapitre 2.7 nous avions vu comment créer un formulaire (sans Express).

Je vous propose de repartir de cet exemple, et de le réécrire pour Express. Nous allons donc retrouver notre formulaire avec ses deux champs de saisie « nom » et « prénom. Et nous allons de surcroît améliorer ce formulaire en implémentant des contrôles simplifiés de type « nom obligatoire » et « prénom obligatoire ». Tant que l'utilisateur n'aura pas renseigné les 2 champs de saisie, il restera bloqué sur le formulaire. La valeur du champ déjà saisi sera conservée pour éviter à l'utilisateur de la resaisir. Dès qu'il aura correctement rempli les 2 champs de saisie et validé le formulaire, ce dernier disparaîtra au profit d'un message du genre :

« Merci <prénom> <nom>, nous avons bien pris en compte votre demande ».

Nous allons construire notre script brique par brique, et vous retrouverez le code source complet en annexe (au chapitre 5.4).

On attaque page suivante avec la fonction qui va nous servir à afficher le formulaire. Je l'ai un peu modifiée par rapport à la version étudiée au chapitre 2.7.

```
/**
 * Fonction dédiée à la génération du formulaire
 * @returns {String}
 */
var testform = function(values, errors=[]) {
    var error_list = '';
    if (errors.length > 0) {
        error_list = '<fieldset><legend>Liste des erreurs</legend>\n';
        for (var i=0, imax=errors.length; i<imax ; i++) {
            error_list += errors[i] + '<br>\n';
        }
        error_list += '</fieldset><br>\n';
    }

    return `${error_list}<form action="/testform" method="post">
    <label>Nom :<input name="nom" value="${values.nom}"></label><br><br>
    <label>Prénom :<input name="prenom" value="${values.prenom}"></label><br><br>
    <input type="submit" name="Valider">
    </form>
    `;
};
```

Dans la fonction testform() ci-dessus, j'ai abandonné la syntaxe « arrow function » au profit d'une déclaration de fonction de type ES5. On retrouve donc le mot clé « function », les accolades et le mot clé « return ».

Mais les plus gros changement sont ceux que j'ai indiqués en gras, à commencer par les paramètres en entrée « values » et « errors » :

- le paramètre « values » n'a pas de valeur par défaut. Il s'agit d'un objet qui sera utilisé pour alimenter les attributs « value » des champs de saisie. Dans le code appelant, on veillera à ce que l'objet « values » contienne les propriétés « nom » et « prenom », initialisées à blanc ou à la valeur saisie précédemment. Mais ça on en reparlera dans un instant.

- le paramètre « errors » est un tableau, si on omet de le transmettre lors de l'appel, il sera vide par défaut. Ce tableau contient la liste des erreurs, il va nous permettre, s'il contient des éléments, de construire un petit « fieldset » HTML présentant cette liste d'erreurs (une ligne par poste de tableau). La construction du code HTML est ici très classique, une petite boucle, un peu concaténation et le tour est joué. Vous noterez que ce « fieldset » est stocké dans la variable « error_list » et que cette dernière est insérée dans la chaîne de caractères renvoyée par le « return ».

A ce stade, peut être avez-vous du mal à vous représenter la manière dont le formulaire va se présenter, alors je vous propose quelques copies d'écran sur la page qui suit.

Le formulaire au premier affichage (oui, je sais, il est moche) :

Nom :

Prénom :

Envoyer

Oups, j'ai oublié de saisir le prénom et j'ai validé le formulaire :

Liste des erreurs
Prénom obligatoire

Nom : aaaa

Prénom :

Envoyer

Re-oups, j'ai fait l'inverse (mais quel boulet ce mec!!) :

Liste des erreurs
Nom obligatoire

Nom :

Prénom : bbb

Envoyer

Cette fois j'ai bien saisi le nom et le prénom, du coup le formulaire disparaît au profit du joli message suivant :

Merci Grégory Jarrige, votre demande
a bien été prise en compte

Ouf, l'honneur est sauf ;)

Bon, maintenant que vous connaissez mieux l'histoire, il ne reste plus qu'à implémenter le reste.

On va reprendre notre fonction « template() » vue dans un chapitre précédent, en ajoutant simplement le titre de la page en premier paramètre :

```
var template = (title='no title', divcontent='') => `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>${title}</title>
        <meta charset="UTF-8">
    </head>
    <body>
        <div>${divcontent}</div>
    </body>
</html>`;
```

On peut se prévoir une route par défaut pour la racine du site, pour la beauté du geste :

```
app.get('/', function (req, res, next) {
    res.end(template('page hello world', 'hello world'));
    console.log('arret / route 1');
});
```

Passons au plat de résistance, avec une route en mode GET, pour afficher le formulaire quand l'url est « /tesform » :

```
app.get("/testform", function (req, res, next) {
    var datas = {};
    datas.nom = '';
    datas.prenom = '';
    res.end(template(
            'Formulaire',
            testform(datas)
        ));

    console.log('arret / route 2');
});
```

… rien de compliqué ici, on avait déjà vu tout ça dans le chapitre précédent.

Passons maintenant à la route « /testform », mais cette fois ci en mode POST. Cette route sera utilisée chaque fois que nous validerons notre formualaire, car j'ai indiqué « /testform » dans l'attribut « action » du formulaire HTML (si vous ne me croyez pas, jetez un œil au code de la fonction testform().

Le squelette de départ pour le POST est identique à celui utilisé pour le GET :

```
app.post("/testform", function (req, res, next) {

});
```

… mais il y a une petite vacherie, car pour récupérer les valeurs saisies dans le formulaire, on ne va pas utiliser la méthode utilisée pour le GET, qui consistait – je le rappelle - à s'appuyer sur l'objet « params », comme ceci :

```
app.get("/products/:productid", function (req, res, next) {
    var prod_id = parseInt(req.params.productid);
    res.end(`Produit : ${prod_id}`);
    console.log('arret / route 2');
});
```

… on va recourir à une autre propriété associée à l'objet « req » qui s'appelle « body ». Mais pour pouvoir disposer de cet objet, nous devons installer un module complémentaire à Express, qui s'appelle « body-parser ». L'installation se fait très simplement via un terminal :

```
npm install body-parser --save
```

Une fois l'installation terminée, nous devons ajouter un « require » au début de notre script :

```
const BODYPARSER = require('body-parser');
//app.use(BODYPARSER.json()); // support json encoded bodies
app.use(BODYPARSER.urlencoded({ extended: true })); // support encoded bodies
```

Vous noterez que nous avons également ajouté 2 lignes en dessous du « require » :

- la première est en commentaire car elle ne sera pas utile ici, elle nous sera utile dans d'autres contexte, par exemple pour récupérer des données au format JSON transmises pour une API

- la seconde ligne est indispensable car c'est grâce à elle que nous allons être en mesure de récupérer les informations provenant du formulaire, qui sont contenues dans le corps de la requête entrante.

Voici le code relatif à la gestion de la requête de type POST :

```
app.post("/testform", function (req, res, next) {
    var datas = {};
    //console.log(req.body);  //=> à activer si vous êtes curieux
    // si présence de blancs, on les supprime avec la fonction trim()
    datas.nom = req.body.nom ? req.body.nom.trim() : '';
    datas.prenom = req.body.prenom ? req.body.prenom.trim() : '';

    // génération du tableau des erreurs
    var erreurs = [];
    if (datas.nom === '') {
        erreurs.push('Nom obligatoire');
    }
    if (datas.prenom === '') {
        erreurs.push('Prénom obligatoire');
    }

    // si présence d'erreurs alors on réaffiche le formulaire
    if (erreurs.length > 0) {
        res.end(template(
                'Formulaire',
                testform(datas, erreurs)
            ));
    } else {
        res.end(template(
                'Merci',
                `Merci ${datas.prenom} ${datas.nom}, votre demande a bien été
prise en compte`
            ));
    }
    console.log('arret / route 3');
});
```

Quelques remarques s'imposent :

- j'ai intégré la création d'un objet « datas » dans lequel je vais stocker le contenu des propriétés « nom » et « prenom » de l'objet « req.body ». J'en ai profité pour passer un « coup » de fonction « trim() », histoire de supprimer les blancs que l'utilisateur aurait pu laisser au début ou à la fin de ses saisies.

- J'ai intégré la gestion des zones obligatoires, avec l'alimentation du tableau des erreurs, tableau qui sera transmis, à la fonction « testform() » à la suite de l'objet « datas »

On peut pour finir ajouter un bout de code pour gérer les erreurs 404, et le tour est joué :

```
app.use(function(req, res, next){
    res.status(404);
    res.end(template(
            'Erreur 404',
            'Désolé, page non trouvée'
    ));
});

app.listen(8080);
```

Je rappelle que vous avez le code source en annexe.

Prenez le temps qu'il faut pour bien comprendre cet exemple, refaites le plusieurs fois si besoin. A partir du moment où vous vous sentirez à l'aise avec ce que nous venons de voir, vous pourrez développer tout ce que vous voudrez avec Node.js et Express.

3.5.6 Intégration de Bootstrap

Nos pages en HTML brut sont vraiment moches. Il est temps que l'on intègre un peu de CSS à notre projet, histoire de donner du « peps » à nos pages. Autant y aller franco et installer Bootstrap !

Comme je n'ai pas encore eu le temps de m'intéresser à Bootstrap 4, je vous propose d'installer plutôt Bootstrap 3. Sachant que Bootstrap a besoin de jQuery pour fonctionner, nous allons donc lancer les 2 commandes suivantes :

```
npm install jquery --save
npm install bootstrap@3 --save
```

Nous pouvons tester le bon fonctionnement de Bootstrap sur le script développé au chapitre précédent. Par précaution, faites-en une copie (sauf si vous versionnez votre code bien sûr), et appliquez les modifications indiquées en gras ci-dessous, au début de votre script :

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

app.use('/', EXPRESS.static(__dirname + '/www'));
   // redirection pour la racine du site
app.use('/js', EXPRESS.static(__dirname + '/node_modules/bootstrap/dist/js'));
   // redirection pour bootstrap JS
app.use('/js', EXPRESS.static(__dirname + '/node_modules/jquery/dist'));
   // redirection pour JS jQuery
app.use('/css', EXPRESS.static(__dirname + '/node_modules/bootstrap/dist/css'));
   // redirection pour CSS bootstrap
```

Dans la fonction « template() », appliquez les quelques modifications en gras ci-dessous :

```
var template = (title='no title', divcontent='') => `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>${title}</title>
        <meta charset="UTF-8">
        <link rel="stylesheet" href="/css/bootstrap.min.css">
        <script src="/js/jquery.min.js"></script>
        <script src="/js/bootstrap.min.js"></script>
    </head>
    <body>
        <div class="container">
            <div class="jumbotron">${divcontent}</div>
        </div>
    </body>
</html>`;
```

Je rappelle que la classe CSS « container » va avoir pour effet de centrer le contenu de la « div » à l'intérieur de la page, et que la classe CSS « jumbotron » a pour effet d'appliquer un fond gris clair au contenu de la « div » concernée.

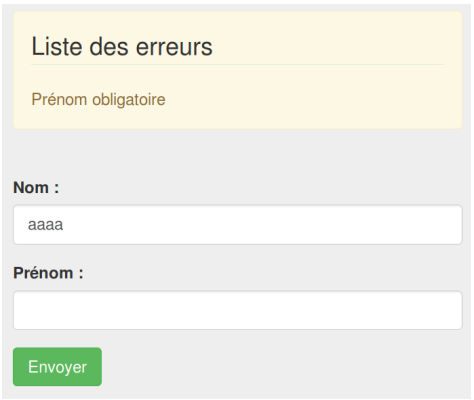
Les 3 lignes en gras ajoutées dans la partie « head » de la page vont avoir pour effet de charger les fichiers CSS et JS nécessaires à Bootstrap pour fonctionner.

En complément, nous allons modifier une partie de la fonction « testform() », pour intégrer dans les champs de saisie les classes CSS de Bootstrap. Ces classes s'appliquent à la fois au « fieldset », que j'ai encapsulé dans une « div », histoire d'appliquer les classes « alert et alert-warning » de Bootstrap.
Dans la partie spécifique au formulaire, il y a beaucoup de changement, avec l'ajout de « div » et de classes CSS, alors je crois qu'un copier-coller s'impose :

```
var testform = function(values, errors=[]) {
    var error_list = '';
    if (errors.length > 0) {
        error_list = '<div class="alert alert-warning">\n';
        error_list += '<fieldset><legend>Liste des erreurs</legend>\n';
        for (var i=0, imax=errors.length; i<imax ; i++) {
            error_list += errors[i] + '<br>\n';
        }
        error_list += '</fieldset></div><br>\n';
    }

    return `${error_list}<form action="/testform" method="post">
    <div class="form-group">
        <label for="idnom">Nom :</label>
        <input type="text" class="form-control" id="idnom" name="nom"
               value="${values.nom}" >
    </div>
    <div class="form-group">
        <label for="idprenom">Prénom :</label>
        <input type="text" class="form-control" id="idprenom" name="prenom"
               value="${values.prenom}" >
    </div>
    <div class="form-group">
        <input type="submit" class="btn btn-success" name="Valider">
    </div>
    </form>
    `;
};
```

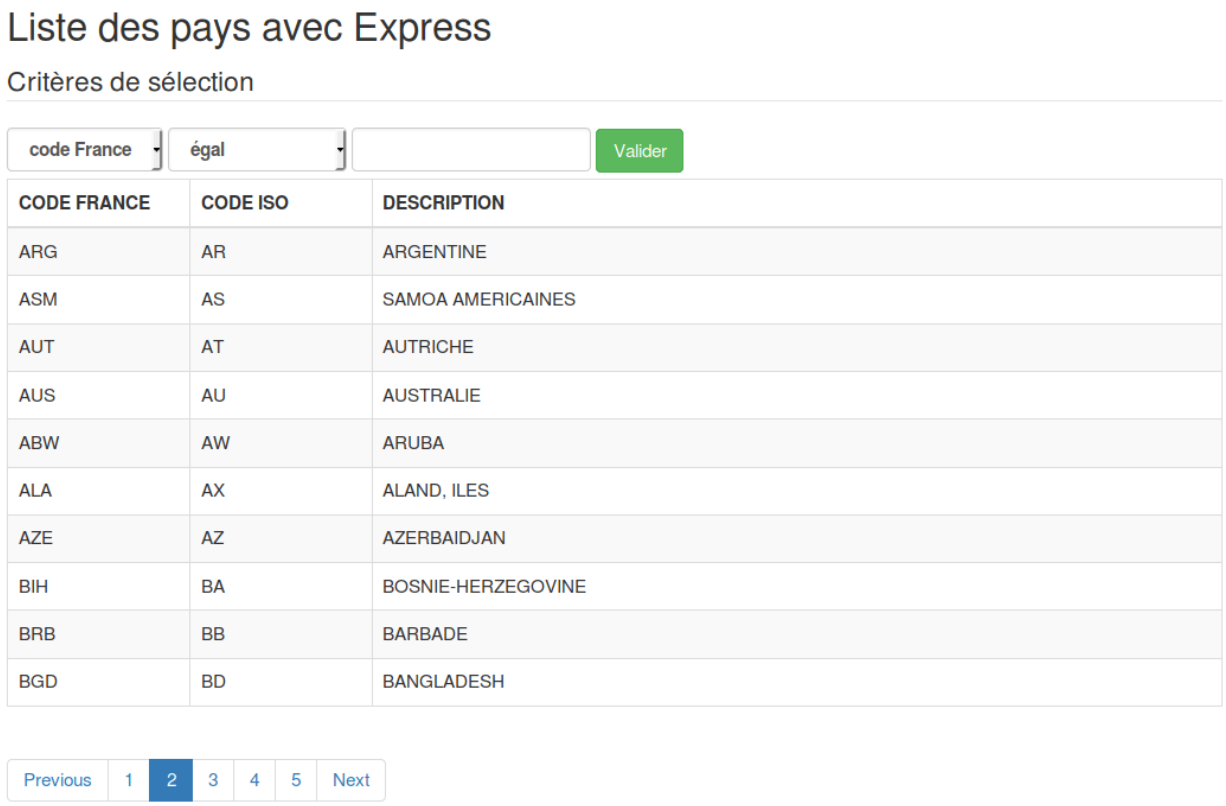
Une fois le serveur redémarré, l'effet est immédiat, puisque notre formulaire apparaît maintenant sous cette forme :



3.6 Miniprojet avec Express, MariaDB, Pagination et Formulaire

Nous avons étudié beaucoup de choses concernant Express, et nous avons aussi fait beaucoup d'expérimentations autour de MariaDB. Il est temps d'essayer de combiner ces deux briques, aussi je vous propose de créer une petite liste avec un formulaire de recherche. Le code que nous allons développer pourra servir de base pour le développement ultérieur de modules de type CRUD (acronyme de « Create Retrieve Update Delete »).

Mais pour que vous compreniez bien de quoi je suis en train de parler, voici une copie d'écran qui présente le résultat final de ce que nous allons développer dans ce chapitre :



Nous avons ici plusieurs éléments, avec en particulier :

- un formulaire de recherche placé au dessus d'une liste

- une liste présentée sous forme de tableau HTML

- une barre de pagination placée sous la liste, qui permet d'afficher les données page par page. Nous avons choisi arbitrairement d'afficher 10 lignes maxi par page.

Pour développer la page que je viens de vous présenter, nous devons créer quelques composants, avec en priorité :

- un composant pour la génération de la barre de pagination

- un composant pour la génération d'un template HTML (qui servira de squelette principal pour notre page web)

Ce que je viens de lister, c'était le minimum syndical pour commencer à travailler. Mais nous serons amenés à développer d'autres composants - moins vitaux - qui contribueront à la lisibilité du code. Je pense notamment à:

- un composant pour la génération du formulaire de recherche

- un composant (ou peut être plusieurs), pour la génération du tableau HTML

Nous allons préparer ces différents composants, pour pouvoir les utiliser un peu comme les briques d'un jeu de construction.

Avant de poursuivre, je vous propose de créer un sous-répertoire « library » dans le répertoire que nous allons utiliser pour notre projet. Nous regrouperons dans ce sous-répertoire les composants « maison » que nous allons développer tout au long de ce chapitre.

N'hésitez pas aussi à vous créer un sous-répertoire « tests », histoire de pouvoir y regrouper les scripts de test que vous pourriez souhaiter développer tout au long de ce chapitre.

3.6.1 Barre de pagination

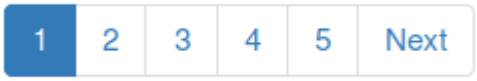
Nous avons donc besoin d'un composant permettant de générer une barre de pagination. Cela n'a l'air de rien, mais c'est un mécanisme essentiel pour la gestion d'une liste digne de ce nom.

C'est un composant intéressant à développer (d'un point de vue de la logique algorithmique), mais cela prend du temps… Heureusement pour nous, il existe un package « pagination » qui fait plutôt bien le travail, il s'appelle « pagination » et il s'installe très facilement :

```
npm install pagination --save
```

Les exemples fournis dans la documentation du composant sont intéressants, mais c'est celui qui est adapté à Bootstrap qui m'intéresse plus particulièrement. Je l'ai testé pour vous, cela donne ceci :

- lors du tout premier affichage



- après avoir sélectionné la page « 2 » :

C'est tout à fait ce qu'il nous faut pour la mise en place d'une liste avec un affichage par page. Allez hop ! Adopté !

L'exemple spécifique à Bootstrap fourni dans la documentation du composant, a nécessité quelques aménagements, car je souhaitais l'encapsuler dans un module « maison », histoire de pouvoir le réutiliser plus facilement dans ce projet ou dans d'autres. Après quelques essais, j'ai abouti au script ci-dessous. J'ai encapsulé ce script dans un fichier que j'ai appelé « pagination_bootstrap.js », et je l'ai stocké dans un sous-répertoire de mon projet que j'ai appelé « library » :

```
module.exports = function (params) {

    var pagination = require('pagination');

    var boostrapPaginator = new pagination.TemplatePaginator({
        prelink: params.prelink, current: params.current,
        rowsPerPage: params.rowsPerPage,
        totalResult: params.totalResult, slashSeparator: params.slashSeparator,
        template: function (result) {
            var i, len, prelink;
            var html = '<div><ul class="pagination">';
            if (result.pageCount < 2) {
                html += '</ul></div>';
                return html;
            }
            prelink = this.preparePreLink(result.prelink);
            if (result.previous) {
                html += '<li><a href="' + prelink + result.previous + '">' +
                         this.options.translator('PREVIOUS') + '</a></li>';
            }
            if (result.range.length) {
                for (i = 0, len = result.range.length; i < len; i++) {
                    if (result.range[i] === result.current) {
                        html += '<li class="active"><a href="' + prelink +
                                result.range[i] + '">' + result.range[i] + '</a></li>';
                    } else {
                        html += '<li><a href="' + prelink + result.range[i] + '">' +
                                result.range[i] + '</a></li>';
                    }
                }
            }
            if (result.next) {
                html += '<li><a href="' + prelink + result.next +
                        '" class="paginator-next">' + this.options.translator('NEXT') +
                        '</a></li>';
            }
            html += '</ul></div>';
            return html;
        }
    });
    return boostrapPaginator.render();
};
```

Si vous êtes pressé de voir ce que ce composant génère, vous pouvez créer un petit script de test comme celui-ci, et le lancer avec la commande « node » :

```
var pagination_tpl = require('./../library/pagination_bootstrap');

var datapage = {
    prelink: '/liste', current: 2, rowsPerPage: 10,
    totalResult: 55, slashSeparator: true
};

var test = pagination_tpl(datapage);

console.log(test);
```

Voici le code HTML produit par notre script de test :

```
<div><ul class="pagination"><li><a href="/liste/page/1">Previous</a></li><li><a
href="/liste/page/1">1</a></li><li class="active"><a href="/liste/page/2">2</a><
/li><li><a href="/liste/page/3">3</a></li><li><a href="/liste/page/4">4</a></li>
<li><a href="/liste/page/5">5</a></li><li><a href="/liste/page/3" class="paginat
or-next">Next</a></li></ul></div>
```

Pour avoir eu l'occasion d'implémenter des composants de pagination notamment en PHP (cf. le cours « PHP premier pas » déjà évoqué précédemment), je peux vous dire que ce package « pagination » est plutôt efficace.

Attention : vous noterez que pour le require, j'ai utilisé 2 petits points dans le chemin d'accès au fichier, comme ceci :

```
var pagination_tpl = require('./../library/pagination_bootstrap');
```

La présence de ces 2 petits points est due au fait que mon script de test se trouve dans le sous-répertoire « tests », et que mon sous-répertoire « library » se trouve au même niveau hiérarchique que le sous-répertoire « tests ». En utilisant les « .. », je demande donc à Node de remonter d'un cran dans la hiérarchie (par rapport au répertoire « tests » dans lequel mon script de test se trouve), pour aller « pointer sur le répertoire « library ».

Si vous êtes familiarisé avec la logique des fonctions « require » et « include » de PHP, alors vous êtes déjà familiarisé avec ce mode de fonctionnement. Dans le cas contraire, il me semblait opportun d'en parler ici.

3.6.2 Template HTML

Dans un précédent exemple, nous avons créé une fonction « template » destinée à générer le squelette HTML d'une page. Mais cette fonction était intégrée « en dur » dans notre script. Je vous propose de la transférer dans un composant indépendant, que j'ai appelé « template_bootstrap.js » et que j'ai stocké dans le sous-répertoire « library ». Voici le code :

```
module.exports = function template (title='no title', divcontent='') {

return  `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>${title}</title>
        <meta charset="UTF-8">
        <link rel="stylesheet" href="/css/bootstrap.min.css">
        <script src="/js/jquery.min.js"></script>
        <script src="/js/bootstrap.min.js"></script>
    </head>
    <body>
        <div class="container">
            <div id="body-page">${divcontent}</div>
        </div>
    </body>
</html>`;

};
```

Je vous laisse le soin d'écrire un petit script de test - si vous le jugez utile - avant de passer au chapitre suivant.

A noter que la classe CSS « container » permet de centrer le contenu de la « div » dans la page.

J'ai placé un identifiant « body-page » sur la « div » qui contiendra l'essentiel de la page. Cela peut être pratique de disposer de cet identifiant pour développer certaines interactions dans le navigateur (par exemple en mode « AJAX »), mais ce n'est pas une obligation. Vous pouvez remplacer cet identifiant par celui de votre choix, ou carrément le supprimer.

3.6.3 Entête de tableau HTML

Pour générer le code HTML du tableau contenant la liste des pays, nous pouvons utiliser un code tel que celui-ci :

```
var template_tableau = `<table class="table table-striped table-bordered"
summary=" Liste des pays ">
  <thead>
  <tr>
    <th>CODE FRANCE</th>
    <th>CODE ISO</th>
    <th>DESCRIPTION</th>
  </tr>
  </thead>
  <tbody>`;
```

Nous avons ici une variable qui contient un code HTML « en dur ». Ce serait pas mal pour un premier test, mais on peut quand même faire mieux que ça. Je vous propose de créer un composant encapsulant une fonction, fonction qui recevra en entrée les paramètres suivants :

- l'intitulé destiné à l'attribut « summary » de la balise « table »,

- un tableau contenant l'intitulé des différentes colonnes à placer dans les balises « th »

Cette fonction renverra en sortie le code HTML « pré-mâché » correspondant à notre entête de tableau HTML.

En partant de l'exemple de composant développé au chapitre précédent (pour le template de la page), je vous propose de développer votre propre composant de génération d'entête de tableau HTML. Prenez le temps d'y réfléchir, de faire des essais. Quand vous aurez obtenu un résultat satisfaisant, vous pouvez tourner la page et étudier la solution que je vous propose.

Si vous êtes bloqué, que vous n'y arrivez pas, n'ayez pas de honte à regarder la solution. Si vous êtes dans ce cas, c'est que vous manquez sans doute encore d'expérience, et que vous avez besoin de « former » votre regard, en étudiant des exemples de code déjà écrits. Du coup l'exemple présenté page suivante va contribuer à « former » votre regard. Je vous encourage dans ce cas, à saisir le code manuellement, plutôt qu'à le copier-coller, car ce faisant, vous allez affiner votre compréhension du code. Sans que vous vous en rendiez compte, des connexions synaptiques vont se mettre en place. Vous ne verrez pas le résultat tout de suite, mais je vous garantis que ce travail sera payant sur le long terme.

Voici donc un composant destiné à générer l'entête de notre tableau HTML.

Ce n'est pas la seule manière de faire, il y en a bien d'autres, mais celle-ci est tout à fait honorable. J'ai placé ce code dans le script « library/template_table_head.js » :

```
module.exports = function template (summary='', colons=[]) {

    var template_cols = '';

    for (var i=0, imax = colons.length ; i < imax ; i++) {
        template_cols += `<th>${colons[i]}</th>`;
    }

    return `
<table class="table table-striped table-bordered" summary="${summary}">
      <thead>
      <tr>
        ${template_cols}
      </tr>
      </thead>
      <tbody>`;

};
```

Pour l'utilisation de ce composant au sein de notre script final, nous pourrons par exemple écrire ceci :

```
var template_table = require('./library/template_table_head.js') ;
...
var liste_cols = ['CODE FRANCE', 'CODE ISO', 'DESCRIPTION'];
var tempo = template_table('Liste des pays', liste_cols);
```

Je vous laisse le soin d'écrire un petit script de test plus abouti - si vous le jugez utile - avant de passer au chapitre suivant.

3.6.4 Formulaire de recherche

Pour la génération du formulaire de recherche, je vous propose de la jouer « minimaliste », avec une simple fonction renvoyant le code HTML défini en dur.

Pour mes premiers tests, j'avais créé à la va-vite le script ci-dessous, que j'ai appelé « library/search_form_countries.js » :

```
module.exports = function template () {

    return `
<h2>Liste des pays avec Express</h2>
<form id="listpays" method="get">
<fieldset>
<legend>Crit&egrave;res de s&eacute;lection</legend>
<label><select id="id_col_rech" name="col_rech" class="form-control">
<option value="1" selected >code France</option>
<option value="2"  >code ISO</option>
<option value="3"  >libellé</option>
</select>
</label> <label><select id="id_typ_rech" name="typ_rech" class="form-
control">
<option value="1" selected >égal</option>
<option value="2"  >contient</option>
<option value="3"  >commence par</option>
</select>
</label> <label><input type="text" name="lib_rech" id="id_lib_rech"
value="" class="form-control"   />
</label> <input type="submit" value="Valider" class="btn btn-success"
/></form><br/>\n`;

};
```

C'était du code HTML « en dur », limité en terme d'usage, car il ne permettait pas d'adapter les valeurs par défaut des champs de saisie, en fonction de la dernière saisie de l'utilisateur. Donc l'utilisateur aurait obligé de resaisir ses critères de sélection après chaque réaffichage du formulaire. C'était néanmoins pratique pour un premier test, d'autant que j'avais repompé ce formulaire sur une vieille application PHP. Cela me permettait de mettre en place rapidement le formulaire suivant :

Mais je ne vais pas vous demander d'implémenter cette version figée. Nous allons implémenter une version améliorée, avec un formulaire capable de rappeler à l'internaute les paramètres de

sa dernière sélection. Pour cela, il faut que la fonction soit en mesure de recevoir les valeurs correspondant aux champs de saisie « col_rech », « typ_rech » et « lib_rech ».

Voici le code du module« library/search_form_countries.js » :

```
module.exports = function template (prm_col_rech='1', prm_typ_rech='1',
  prm_lib_rech='') {

    if (prm_col_rech === undefined) {
        prm_col_rech = '1';
    }
    if (prm_typ_rech === undefined) {
        prm_typ_rech = '1';
    }
    if (prm_lib_rech === undefined) {
        prm_lib_rech = '';
    }

    var list_col_rech = [];
    list_col_rech.push({valeur:'1', libelle: 'code France'});
    list_col_rech.push({valeur:'2', libelle:'code ISO'});
    list_col_rech.push({valeur:'3', libelle:'Libellé'});

    var list_typ_rech = [];
    list_typ_rech.push({valeur:'1', libelle: 'égal'});
    list_typ_rech.push({valeur:'2', libelle: 'contient'});
    list_typ_rech.push({valeur:'3', libelle: 'commence par'});

    /*
     * L'objectif est de générer un code HTML tel que celui-ci
     *  <select id="id_col_rech" name="col_rech" class="form-control">
     *     <option value="1">code France</option>
     *     <option value="2">code ISO</option>
     *     <option value="3" selected>Libellé</option>
     *  </select>
     */
    var opt_col_rech = `${list_col_rech.map((item, i) =>
              `<option value="${item.valeur}" ${item.valeur == prm_col_rech ?
                  'selected': '' }>${item.libelle}</option>`).join('\n')}`;

    var opt_typ_rech = `${list_typ_rech.map((item, i) =>
              `<option value="${item.valeur}" ${item.valeur == prm_typ_rech ?
                  'selected': '' }>${item.libelle}</option>`).join('\n')}`;

    return `
<h2>Liste des pays avec Express</h2>
<form id="listpays" method="get">
<fieldset>
<legend>Crit&egrave;res de s&eacute;lection</legend>
<label><select id="id_col_rech" name="col_rech" class="form-control">
${opt_col_rech}
</select>
</label> <label><select id="id_typ_rech" name="typ_rech" class="form-control">
${opt_typ_rech}
</select>
</label> <label><input type="text" name="lib_rech" id="id_lib_rech" value="$
{prm_lib_rech}" class="form-control"   />
</label> <input type="submit" value="Valider" class="btn btn-success"
/></form><br/>\n`;
```

```
};
```

Dans cette fonction, nous avons utilisé intensivement les littéraux de modèle que nous avions présenté à la fin du chapitre 2.6. Si vous avez un peu de mal avec cette syntaxe, je vous encourage à écrire un petit script de test pour étudier le code HTML produit par cette fonction.

3.6.5 La pagination du point de vue de SQL

J'ai parlé jusqu'ici du composant « pagination », qui va permettre de représenter graphiquement une barre de pagination dans la page. Mais je n'ai pas expliqué ce qui se passe dans les coulisses. En effet, du côté du serveur, pour pouvoir réaliser une pagination correcte, j'ai besoin de connaître 2 informations :
1 – j'ai besoin de connaître le nombre total de lignes concernées par ma sélection (qu'il s'agisse d'afficher tous les pays, ou un panel de pays)
2 – j'ai besoin de savoir sur quelle page je me trouve à un moment donné (la page 1, la page 2, etc.), pour déterminer quelles lignes SQL je dois afficher

Donc côté SQL, pour chaque changement de page, nous devons exécuter les 2 requêtes SQL suivantes :

- pour l'affichage de tous les pays, on aura ceci :

```
SQL1 : SELECT count(*) as comptage FROM countries;
SQL2 : SELECT codinter, codfra, countryname FROM countries;
```

- pour l'affichage des pays dont le nom contient « FR », on aura cela :

```
SQL1 : SELECT count(*) as comptage FROM countries
       WHERE countryname LIKE '%F%';
SQL2 : SELECT codinter, codfra, countryname FROM countries
       WHERE countryname LIKE '%F%';
```

Et puisque nous devons effectuer une pagination avec la requête « SQL2 », nous devrons en plus lui ajouter une clause LIMIT en précisant à partir de quelle ligne nous commençons la pagination, et combien de lignes nous souhaitons afficher par la suite. En supposant que nous souhaitions afficher la 20ème ligne et les 10 lignes qui suivant, cela donnerai ceci sur l'affichage de tous les pays :

```
SQL2 : SELECT codinter, codfra, countryname FROM countries LIMIT 20, 10;
```

… et cela sur l'affichage ave le filtre « FR » appliqué sur le nom :

```
SQL2 : SELECT codinter, codfra, countryname FROM countries
       WHERE countryname LIKE '%F%' LIMIT 20, 10;
```

Donc à chaque fois que nous allons effectuer une demande de pagination, nous allons récupérer le numéro de page courant, calculer à quel numéro de ligne SQL commence cette page. Si on suppose que le numéro de page courant est stocké dans la variable « current_page », que le nombre maximum de ligne par page est stocké dans la variable « max_rows_per_page » (que l'on pourrait d'ailleurs transformer en constante), on aboutit à un code tel que celui-ci :

```
// calcul du numéro de ligne SQL correspondant au début
//   de la page courante (pour intégration dans la clause LIMIT)
var num_ligne = (current_page - 1) * max_rows_per_page ;

// ajout de la clause LIMIT à la requête SQL d'affichage
sql2 += ` limit ${num_ligne}, ${max_rows_per_page}`;
```

Je pense que ces explications vous aideront à mieux comprendre le code que nous allons mettre en place par la suite. Mais si ça ne vous semble pas clair à ce stade, cela devrait se clarifier en le mettant en œuvre réellement.

3.6.6 Assemblage des briques (1ère version)

Dans ce chapitre, nous allons travailler sur la mise en place du script principal regroupant les principaux composants (développés dans les chapitres précédents) et définissant les routes supportées par notre serveur de pages webs.

Je vous donne tout d'abord la structure générale du script que nous allons écrire. Il va se composer des parties suivantes :

- inclusion (via require) du package Express

- déclaration des chemins statiques pour les fichiers JS et CSS intégrés dans la page (c'est ce que nous avons vu dans le chapitre 3.5.6, « intégration de Bootstrap »)

- inclusion du package personnalisé « dbconnex » (vu au chapitre 3.4.1)

- inclusion des packages personnalisés suivants (vus aux chapitres 3.5.7.1 à 3.5.7.4) : *pagination_bootstrap*, *template_html*, *template_table_head*, *search_form_countries*

- déclaration d'une route « / » standard (qui renvoie ce que vous voulez, par exemple « hello world »)

- déclaration d'une route « /liste » affichant la première page de la liste des pays

- déclaration d'une route «/liste/page/:page » affichant la même liste de pays, mais en fonction de la page spécifiée

- déclaration d'une route générique traitant les erreurs 404 (ressouce non trouvée)

A la lecture de cette liste, on pressent que 2 routes vont partager tout ou partie du même code, je pense bien évidemment aux routes « /liste » et «/liste/page/:page ». C'est la raison pour laquelle je vais écrire une fonction générique qui sera utilisée par les 2 routes. J'ai appelé cette fonction « countriesList », les 2 routes concernées seront donc définies de la façon suivante :

```
app.get("/liste", countriesList);

app.get("/liste/page/:page", countriesList);
```

Nous verrons dans un instant ce qui se passe dans la fonction « countriesList ». Mais pour le moment nous allons rédiger le code du script principal. C'est cool, je me rends compte qu'il tient sur une seule page (à condition de ne pas y mettre la fonction countriesList), alors je vous invite à tourner la page pour l'étudier tranquillement.

Un petit détail qui a son importance : pour l'instant le fonctionnement de recherche ne fonctionne pas. Ce qui fonctionne bien en revanche, dans cette première version, c'est l'affichage page par page, en cliquant sur la barre de pagination.

Voici le code la première version :

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

// préparation des redirections pour les routes statiques
app.use('/', EXPRESS.static(__dirname + '/www'));
app.use('/js', EXPRESS.static(__dirname + '/node_modules/bootstrap/dist/js'));
app.use('/js', EXPRESS.static(__dirname + '/node_modules/jquery/dist'));
app.use('/css', EXPRESS.static(__dirname +
    '/node_modules/bootstrap/dist/css'));

// chargement du package personnalisé pour la connexion à MariaDB
const DBMODULE = require('./dbconnex');
// initialisation du connecteur à MariaDB
const conn = DBMODULE.init();

var pagination_tpl = require('./library/pagination_bootstrap.js');
var template_page = require('./library/template_html.js') ;
var template_table = require('./library/template_table_head.js') ;
var template_search_form = require('./library/search_form_countries.js') ;

var countriesList = function (req, res, next) {
    console.log('arret / route 2');
    // ATTENTION, fonction ici incomplète, version complète page suivante
}

app.get('/', function (req, res, next) {
    console.log('arret / route 1');
});

app.get("/liste", countriesList);

app.get("/liste/page/:page", countriesList);

app.use(function(req, res, next){
    res.status(404);
    res.end(template_page(
            'Erreur 404',
            'Désolé, page non trouvée'
    ));
});

app.listen(8080);
```

Voici le code source de la fonction « countriesList » :

```
var countriesList = function (req, res, next) {
        console.log('arret / route 2');

    // détermination du numéro de page courant
    var current_page = 1;
    if (req.params.page) {
        console.log(req);
        current_page = parseInt(req.params.page);
    }

    // définition du nombre de lignes maxi par page (pour la pagination)
    var max_rows_per_page = 10;

    // préparation des requêtes SQL de comptage et d'affichage
    var sql1 = "SELECT count(*) as comptage FROM countries";
    var sql2 = "SELECT codinter, codfra, countryname "+
               " FROM countries order by codfra";

    // Exécution de la requête de comptage
    conn.execute(sql1, [], function (err, rows, result) {
        //if (err) throw err;
        if (err) {
            console.log("ERREUR SUR REQUETE : " + err.code +
                    " (" + err.message + ")");
            return;
        }

        // récupération du nombre de lignes total renvoyé par la requête SQL
        var nb_lig_total = 0;
        if (rows[0]) {
            nb_lig_total = rows[0].comptage ;
        }

        if (nb_lig_total === 0) {
            // TODO : rendu de la liste vide à améliorer ultérieurement
            res.end(template_page(
                'Formulaire',
                'liste vide' + '<br>\n'
            ));

        } else {

            // calcul du numéro de ligne SQL correspondant au début
            //   de la page courante
            var num_ligne = (current_page - 1) *  max_rows_per_page ;

            // ajout de la clause LIMIT à la requête SQL d'affichage
            sql2 += ` limit ${num_ligne}, ${max_rows_per_page}`;
```

```
            // Execution de la requête SQL d'affichage des pays
            conn.execute(sql2, [], function (err, rows, result) {
                if (err) {
                    console.log("ERREUR SUR REQUETE : " + err.code +
                            " (" + err.message + ")");
                    return;
                }

                // tableau contenant l'intitulé des colonnes du tableau HTML
                var liste_cols = ['CODE FRANCE', 'CODE ISO', 'DESCRIPTION'];

                // génération entête de page (incluant formulaire de recherche
                //   et entête de tableau HTML)
                var tmp_liste = template_search_form()  +
                                template_table('Liste des pays', liste_cols);

                // construction des lignes détail du tableau HTML
                for (var i in rows) {
                    var row = rows[i];
                    tmp_liste += `<tr>
    <td width="15%">${row.codinter}</td>
    <td width="15%">${row.codfra}</td>
    <td width="70%">${row.countryname}</td>
  </tr>\n`;
                }
                tmp_liste += '</tbody>\n</table>\n';

                // Objet définiisant les parammètres de pagination
                var datapage = {
                    prelink: '/liste', current: current_page,
                    rowsPerPage: max_rows_per_page,
                    totalResult: nb_lig_total, slashSeparator: true
                };

                // ajout de la barre de pagination au code HTML
                tmp_liste += pagination_tpl(datapage);

                // rendu final de la page
                res.end(template_page(
                    'Formulaire',
                    tmp_liste
                ));

            });
        }
    });
};
```

Si vous n'avez pas commis d'erreur dans votre code, la pagination devrait fonctionner, et vous pouvez vous amuser à faire défiler la liste des pays.

Mais comme je vous le disais précédemment, le formulaire de recherche est pour le moment inopérant. Nous devons trouver un moyen de l'intégrer dans notre traitement. C'est l'objet du chapitre qui suit.

3.6.7 Assemblage des briques (2ère version avec le formulaire)

Si à ce stade vous saisissez des paramètres de recherche dans le formulaire, vous allez constater un phénomène étrange.

Par exemple, si je saisis dans le formulaire les paramètres suivants :

… je vais retrouver ces paramètres dans l'URL de la page sous la forme suivante :

`http://localhost:8080/liste/page/1`**`?col_rech=2&typ_rech=2&lib_rech=FR`**

Si nous sommes sur le tout premier affichage d'une page, l'URL sera plutôt celle ci (c'est la même chose sans le paramètre « page ») :

`http://localhost:8080/liste/`**`?col_rech=2&typ_rech=2&lib_rech=FR`**

Cela ne correspond pas vraiment à la manière dont les routes fonctionnent dans Express. Mais c'est normal de retrouver les champs de saisie du formulaire (col_rech, typ_rech et lib_rech) sous cette forme, dans l'URL envoyée au serveur (et récupérée par la suite dans le navigateur).

En effet, notre formulaire est un formulaire HTML tout à fait classique, qui fonctionne sans l'aide d'aucun code Javascript. Nous ne sommes pas ici dans un mode de fonctionnement de type AJAX. Nous aurions pu le développer en mode AJAX ce formulaire (et vous pourrez le faire dans une autre version), mais le fonctionnement actuel est intéressant car il me garantit que l'application fonctionne, même si quelque chose se passe mal côté navigateur, comme par exemple un plantage dans le code Javascript, ou si l'interpréteur Javascript est désactivé dans le navigateur de l'internaute (cela peut arriver dans certaines sociétés appliquant des règles de sécurité drastiques).

N'empêche que notre formulaire avec ses critères de sélection est inopérant. Nous devons trouver un moyen de mettre le formulaire dans la boucle. Heureusement, il y a un moyen simple

de récupérer les paramètres qui se trouvent à droite du point d'interrogation dans la requête HTTP suivante :

```
http://localhost:8080/liste/?col_rech=2&typ_rech=2&lib_rech=FR
```

... le moyen en question consiste à exploiter le contenu de l'objet « req.params ». Dans le cas de notre formulaire de recherche, cet objet contient les propriétés suivantes :
- req.params.col_rech : qui contient la valeur « 2 »
- req.params.typ_rech : qui contient aussi la valeur « 2 »
- req.params.lib_rech : qui contient la valeur « FR »

Fort de ces informations, nous sommes en mesure de générer dynamiquement des requêtes SQL de ce type :

```
SQL1 : SELECT count(*) as comptage FROM countries WHERE codinter LIKE ?;
SQL2 : SELECT codinter, codfra, countryname FROM countries
       WHERE codinter LIKE ?;
```

Lors de l'exécution de ces requêtes, nous transmettrons un tableau de cette forme : ['%FR%']
... car je rappelle que nous faisons tout notre possible pour éviter tout risque d'injection SQL. Il est donc hors de question de concaténer le «%FR » pour aboutir à des horreurs de ce genre :

```
SQL1 : SELECT count(*) as comptage FROM countries WHERE codinter LIKE  '%FR%';
SQL2 : SELECT codinter, codfra, countryname FROM countries
       WHERE codinter LIKE  '%FR%';
```

En revanche, ce que nous pouvons ajouter par concaténation, sur la requête SQL2, c'est la clause LIMIT, comme dans l'exemple suivant :

```
SELECT codinter, codfra, countryname FROM countries
WHERE codinter LIKE ? Limit 0, 10
```

Il y a un autre point important à prendre en considération. Nous avons vu que nous sommes en mesure de récupérer, via l'objet « req.params » les paramètres transmis par le formulaire. Nous sommes donc en mesure d'afficher la première page de données correspondant à la sélection. Mais le composant « pagination » ne nous permet pas de transmettre ces paramètres sous la même forme, afin que l'on puisse les réexploiter sur la seconde page. Il est en revanche possible de transmettre ces informations sous un forme un peu différente.

En effet, le composant pagination génère un code HTML composé d'une série de balises « a » qui se présentent sous la forme de boutons cliquables, par la magie du CSS de Bootstrap. Voici un exemple de balise « a » généré par le composant « pagination :

```
<a href="/liste/page/2">2</a>
```

On pourra difficilement, à moins de modifier le package « pagination », générer des liens de ce type là :

```
<a href="/liste/page/2/?col_rech=2&typ_rech=2&lib_rech=F">2</a>
```

En revanche, il est possible de générer assez facilement des liens sous cette forme :

```
<a href="/col_rech/2/typ_rech/2/lib_rech/F/liste/page/2">2</a>
```

… car de cette manière, nous nous intégrons dans le système des routes proposé par Express, et ça devient cool. Mais cela a 2 impacts sur notre code :

Premièrement, nous devons maintenant gérer 2 cas de figure :
- soit les paramètres du formulaire proviennent de l'objet « req.query » (c'est le cas lors de la validation du formulaire)
- soit les paramètres du formulaire proviennent de l'objet « req.params » (c'est le cas lors de l'utilisation de la barre de pagination)

Deuxièment, nous sommes dans l'obligation de doubler les routes pour prendre en compte cette nouvelle situation, ce qui nous donne à l'arrivée ceci :

```
// Préparation des routes, avec et sans critères de sélection, avec et
//  sans pagination, soit 4 possibilités
app.get("/liste", countriesList);

app.get("/liste/page/:page", countriesList);

app.get("/col_rech/:col_rech/typ_rech/:typ_rech/lib_rech/:lib_rech/liste",
countriesList);

app.get("/col_rech/:col_rech/typ_rech/:typ_rech/lib_rech/:lib_rech/liste/page/
:page", countriesList);
```

Eh oui, cela fait plus de boulot, mais je vous garantis que ça fonctionne, et que c'est très performant, quand c'est correctement implémenté.

L'essentiel des modifications concerne la fonction « countriesList » (encore elle), car nous devons intégrer pas mal de changements. Dans la version du code qui va suivre, j'ai opté pour une écriture assez linéaire, peu modulaire, mais accompagnée de beaucoup de commentaires, afin que vous puissiez bien comprendre chaque étape.
Ce code pourra être assez largement refactorisé, certaines parties pourront assez facilement être transformées en composants réutilisables. Mais c'est un travail que vous pourrez faire dans un second temps, quand vous aurez bien compris la logique de fonctionnement de ce qui va suivre.

Bonne lecture, bon courage, on se retrouve dans 4 pages ;).

```
var countriesList = function (req, res, next) {

    // Variable qui servira de container pour les critères de
    //  sélection du formulaire, à intégrer dans la route pour assurer
    //  un bon fonctionnement du composant de pagination
    var current_prelink = '';

    // Détermination du numéro de page courant
    //  (indispensable pour le composant de pagination et pour déterminer
    //  les valeurs de la clause LIMIT dans la requête SQL stockée dans "sql2".
    var current_page = 1;
    if (req.params.page) {
        current_page = parseInt(req.params.page);
    }

    // On fixe le nombre de lignes maxi qu'une page peut afficher
    var max_rows_per_page = 10;

    // Préparation des requêtes de comptage et d'affichage
    var sql1 = "SELECT count(*) as comptage FROM countries";
    var sql2 = "SELECT codinter, codfra, countryname FROM countries";

    // Préparation du tableau qui contiendra d'éventuels critères de recherche
    //  à transmettre à SQL
    var sql_params = [];

    // On teste si des critères de recherche ont été saisis dans le formulaire
    //  si oui, on les trouve soit dans "req.query" (lors de la validation du
    //  formulaire, ou dans req.params, pour les paginations
    if ('col_rech' in req.query || req.params.col_rech) {
        var col_rech = '';
        var typ_rech = '';
        var lib_rech = '';

        // Le paramètre "col_rech" est présent dans req.query lors de la
        //  validation du formulaire, ensuite sa valeur est intégrée dans la
        //  route et est récupérable via l'objet "req.params". Et c'est
        //  pareil pour les paramètres "typ_rech" et "lib_rech".
        // Quand l'internaute demande une nouvelle sélection, on force le
        //  repositionnement sur la page 1 (sinon la panation peut être faussée).
        if ('col_rech' in req.query) {
            // pour 1er affichage => repositionnement obligatoire sur page 1
            current_page = 1;
            col_rech = req.query['col_rech'];
            typ_rech = req.query['typ_rech'];
            lib_rech = req.query['lib_rech'];
        } else {
            col_rech = req.params.col_rech;
            typ_rech = req.params.typ_rech;
            lib_rech = req.params.lib_rech;
        }

        // Intégration des paramètres du formulaire dans la route standard
        current_prelink += '/col_rech/' + col_rech;
        current_prelink += '/typ_rech/' + typ_rech;
        current_prelink += '/lib_rech/' + lib_rech;

        // Préparation de la clause WHERE qui sera ajoutée aux requêtes SQL
        var where = ' WHERE ';
```

```
    // clause WHERE modulée selon critère indiqué dans le formulaire
    if (col_rech === '1') {
        // recherche sur code France
        where += 'codfra';
    } else {
        if (col_rech === '2') {
            // recherche sur code international (ISO)
            where += 'codinter';
        } else {
            // recherche sur libellé
            where += 'countryname';
        }
    }

    // recherche SQL de type "égal", "contient" ou "commence par" ?
    if (typ_rech === '1') {
        // recherche de type "égal"
        where += ' = ?';
        sql_params.push(lib_rech);
    } else {
        if (typ_rech === '2') {
            // recherche de type "contient"
            where += ' LIKE ?';
            sql_params.push('%'+lib_rech+'%');
        } else {
            // recherche de type "commence par"
            where += ' LIKE ?';
            sql_params.push(lib_rech+'%');
        }
    }

    // ajout de la clause WHERE à la fin des 2 requêtes SQL
    sql1 += where ;
    sql2 += where ;

    // quelques mouchards pour contrôle, à supprimer ultérieurement (TODO)
    console.log(sql1);
    console.log(sql2);
    console.log(sql_params);
}

// Exécution de la requête de comptage
conn.execute(sql1, sql_params, function (err, rows, result) {
    //if (err) throw err;
    if (err) {
        console.log("ERREUR SUR REQUETE : " + err.code +
                " (" + err.message + ")");
        return;
    }

    // récupération du nombre de lignes total
    var nb_lig_total = 0;
    if (rows[0]) {
        nb_lig_total = rows[0].comptage ;
    }

    // tableau contenant l'intitulé des colonnes du tableau HTML
    var liste_cols = ['CODE ISO', 'CODE FRANCE', 'DESCRIPTION'];
```

```
        // génération entête de page (incluant formulaire de recherche
        //   et entête de tableau HTML
        var tmp_liste = template_search_form(col_rech, typ_rech, lib_rech) +
                template_table('Liste des pays', liste_cols);

        // mouchard à supprimer ultérieurement (TODO)
        console.log('nombre de lignes total renvoyé par SQL1 : '+nb_lig_total);

        if (nb_lig_total === 0) {
            // Affichage du tableau HTML avec une ligne vide
            tmp_liste += '<tr><td colspan="3">Pas de données</td></tr>\n';
            tmp_liste += '</tbody>\n</table>\n';

            res.end(template_page(
                'Formulaire',
                tmp_liste
            ));
        } else {
            // calcul du numéro de ligne SQL correspondant au début
            //   de la page courante (pour intégration dans la clause LIMIT)
            var num_ligne = (current_page - 1) *  max_rows_per_page ;

            // ajout de la clause LIMIT à la requête SQL d'affichage
            sql2 += ` limit ${num_ligne}, ${max_rows_per_page}`;

            // mouchard à supprimer ultérieurement (TODO)
            console.log('requête SQL2 avec clause LIMIT : ' + sql2);

            // Execution de la requête SQL numéro 2 pour l'affichage des pays
            conn.execute(sql2, sql_params, function (err, rows, result) {
                if (err) {
                    console.log("ERREUR SUR REQUETE : " + err.code +
                            " (" + err.message + ")");
                    return;
                }

                // construction des lignes détail du tableau HTML
                for (var i in rows) {
                    tmp_liste += `<tr>
  <td width="15%">${rows[i].codinter}</td>
  <td width="15%">${rows[i].codfra}</td>
  <td width="70%">${rows[i].countryname}</td>
</tr>\n`;
                }

                // ajout du pied de tableau HTML
                tmp_liste += '</tbody>\n</table>\n';

                // Objet définiisant les parammètres de pagination
                var datapage = {
                    prelink: current_prelink + '/liste', current: current_page,
                    rowsPerPage: max_rows_per_page,
                    totalResult: nb_lig_total, slashSeparator: true
                };

                // ajout de la barre de pagination
                tmp_liste += pagination_tpl(datapage);

                // rendu final du tableau HTML
                res.end(template_page(
                    'Formulaire',
```

```
                    tmp_liste
                ));

            });
        }
    });
};
```

Ah, vous êtes encore là !!! C'est super.

Fort de cette expérience dans le monde de la pagination, vous comprenez sans doute pourquoi certains développeurs préfèrent déporter la logique de pagination dans le navigateur, en utilisant des composants Javascript « clés en main », comme par exemple Datatables (www.datatables.net) ou jqGrid (www.trirand.com). Ces composants sont très bien mais quand ils sont utilisés de manière standard – comme le font les 3/4 des développeurs – on aboutit à des aberrations de ce genre :
- lancement d'une requête SQL remontant plusieurs milliers de ligne vers le langage serveur (PHP, Nodejs ou autre…),
- création d'une page HTML contenant un tableau HTML contenant plusieurs milliers de lignes,
- transfert de ce tableau HTML vers le client (le navigateur de l'internaute, qui est peut être sur un smarphone)
- parsing dans le navigateur (via Datatables ou jqGrid) du tableau HTML et génération d'une pagination dynamique

Résultat des courses, on crée des goulets d'étranglement à tous les étages :
- au niveau de la base de données qui est sur-sollicitée,
- au niveau des échanges entre la base de données et le langage serveur (PHP ou autre),
- au niveau du langage serveur qui se trouve obligé de générer des quantités astronomiques de code HTML (souvent en pure perte),
- au niveau de la bande passante, qui est surchargée de code HTML inutile
- au niveau du navigateur de l'internaute qui, s'il est localisé sur un smarphone, va peiner (avec sa mémoire et son processeur limités) à traiter un tel volume de données

Quand j'évoque l'idée de « données générées en pure perte », je veux dire par là que très souvent l'internaute n'a pas besoin de toutes les données qui lui sont envoyées, et que 7 fois sur 10, il n'utilisera même pas la pagination qui lui est proposée.

Le mode d'utilisation que je viens de décrire – que ce soit pour Datatables ou jqGrid - est sans doute très pratique pour un prototype d'application, il n'est tout simplement pas viable sur une application utilisée en production, par plusieurs milliers de personnes (voire plus) chaque jour.

Bien sûr, je grossis un peu le trait, heureusement il y a des développeurs plus malins qui utilisent les composants que je viens de citer en mode AJAX, déclenchant une nouvelle requête sur le

serveur chaque fois que l'internaute clique sur un bouton de la barre de pagination. On limite ainsi le volume des échanges client-serveur. Mais combien de développeurs ont pris véritablement le temps de s'intéresser à ce mode de fonctionnement, alors que le mode que j'ai décrit juste avant est tellement plus simple et rapide à mettre en œuvre ? Je crains malheureusement que ce soit un très petit pourcentage.

En tout cas, en ce qui vous concerne, vous savez maintenant comment fonctionne une pagination. C'est extrêmement enrichissant de travailler sur ce sujet, car cela nous fait toucher à différentes techniques, avec notamment :
- la génération d'une barre de navigation HTML (ici via un composant qui fait le gros du travail),
- la génération de 2 requêtes SQL (une pour le comptage, une pour l'affichage des données)
- la mise en forme des données en HTML (ici en s'appuyant sur les nouvelles possibilités de templating intégrées dans la norme ES6)
- la gestion des données provenant du formulaire et la gestion de leur durée de vie au travers du composant de pagination

Pour vérifier que vous vous sentez à l'aise avec les techniques que nous avons évoquées dans ce chapitre, je vous invite à créer un mini-projet sur le même principe en procédant ainsi :
- utilisez le site mockaroo.com pour générer une table SQL contenant un millier de lignes (à moins que vous n'ayiez déjà sous la main une table intéressante sur laquelle vous préférez travailler),
- injectez cette table dans une base MySQL (ou MariaDB) de votre environnement de travail,
- reprenez le code de ce chapitre et utilisez-le pour créer une page similaire à celle que nous venons de développer
- adaptez le formulaire de recherche à la structure de la table SQL que vous avez choisi d'utiliser

Une fois que vous êtes certain que tout fonctionne, réfléchissez à la manière dont vous pouvez refactoriser le code. En comparant le code que nous avons étudié ensemble au code que vous avez vous même écrit, identifiez les portions de code qui peuvent être mutualisées, et déportez ces portions de code dans des modules Node.js « maison » (que vous placerez par exemple dans le répertoire « library »). Intégrez les changements dans votre module et dans celui que nous avons étudié dans ce chapitre. Testez tout soigneusement. Pensez à versionner votre code (avec Git de préférence) tout au long de ces opérations (si vous n'êtes pas très à l'aise avec Git, essayez d'en profiter pour vous y mettre sérieusement).

A noter que vous retrouverez en annexe le code source complet du script que nous venons de développer (cf. chapitre 5.7).

3.7 Dataviz

3.7.1 D3.js

Le sujet du chapitre précédent était un peu lourd, ce serait bien d'aborder un sujet plus fun et léger. Aussi je vous propose d'aborder D3.js.
D3.js est un framework JS open source dédié à la dataviz. Il est développé par Mike Bostock, et on peut dire que, dans sa catégorie, D3 est une référence.

Par curiosité, lançons en ligne de commande un « node search d3 »…

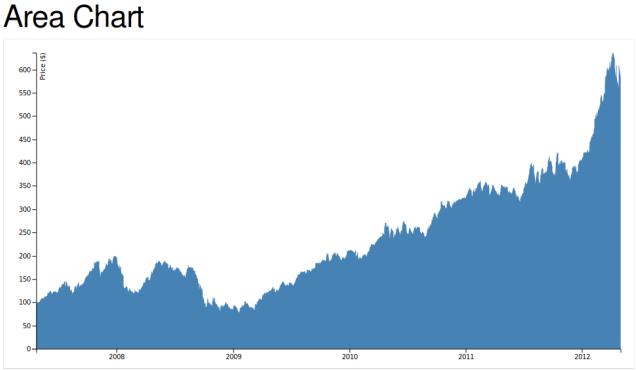
```
NAME                  | DESCRIPTION          | AUTHOR         | DATE       | VERSION  | KEYWORDS
d3                    | Data-Driven…         | =mbostock…     | 2017-12-26 | 4.12.2   | dom visualization svg animation canvas
d3-shape              | Graphical…           | =mbostock      | 2017-05-16 | 1.2.0    | d3 d3-module graphics visualization canvas
d3-array              | Array manipulation,… | =mbostock      | 2017-09-23 | 1.2.1    | d3 d3-module histogram bisect shuffle stati
d3-selection          | Data-driven DOM…     | =mbostock      | 2017-11-21 | 1.2.0    | d3 d3-module dom selection data-join
d3-queue              | Evaluate…            | =mbostock      | 2017-05-09 | 3.0.7    | d3 d3-module asynchronous async queue
d3-interpolate        | Interpolate…         | =mbostock      | 2017-11-21 | 1.1.6    | d3 d3-module interpolate interpolation colo
d3-format             | Format numbers for…  | =mbostock      | 2018-01-09 | 1.2.2    | d3 d3-module format localization
d3-color              | Color spaces! RGB,…  | =mbostock      | 2017-03-10 | 1.0.3    | d3 d3-module color rgb hsl lab hcl lch cube
d3-time-format        | A JavaScript time…   | =mbostock      | 2017-11-21 | 2.1.1    | d3 d3-module time format strftime strptime
d3-scale              | Encodings that map…  | =mbostock      | 2017-11-21 | 1.0.7    | d3 d3-module scale visualization
d3-collection         | Handy data…          | =mbostock      | 2017-06-21 | 1.0.4    | d3 d3-module nest data map set object colle
d3-time               | A calculator for…    | =mbostock      | 2017-11-21 | 1.0.8    | d3 d3-module time interval calendar
d3-ease               | Easing functions…    | =mbostock      | 2017-03-10 | 1.0.3    | d3 d3-module ease easing animation transiti
d3-hierarchy          | Layout algorithms…   | =mbostock      | 2017-06-09 | 1.1.5    | d3 d3-module layout tree treemap hierarchy
d3-dsv                | A parser and…        | =mbostock      | 2017-11-21 | 1.0.8    | d3 d3-module dsv csv tsv
d3-geo                | Shapes and…          | =mbostock      | 2017-12-08 | 1.9.1    | d3 d3-module geo maps cartography
d3-transition         | Animated…            | =mbostock      | 2017-11-21 | 1.1.1    | d3 d3-module dom transition animation
d3-path               | Serialize Canvas…    | =mbostock      | 2017-03-10 | 1.0.5    | d3 d3-module canvas path svg graphics Canva
d3-timer              | An efficient queue…  | =mbostock      | 2017-09-03 | 1.0.7    | d3 d3-module timer transition animation req
d3-axis               | Displays automatic…  | =mbostock      | 2017-06-08 | 1.0.8    | d3 d3-module axis scale visualization
```

Ah ouais, quand même !!! Il y a du monde !
C'est vrai que D3 est très modulaire, on peut charger uniquement les composants de D3 dont on a besoin.

Le site officiel de D3 présente une palette d'exemples impressionnante :
https://d3js.org/

Pour nous familiariser avec D3, et surtout avec son intégration dans Node.js, nous allons tester l'intégration d'un graphe simple, pris dans la page « examples » du site officiel de D3. Je vous propose de prendre le graphe suivant (qui se trouve dans la partie « Basic Charts ») :

https://bl.ocks.org/mbostock/3883195



Nous devons tout d'abord intégrer D3 dans notre projet :

```
npm install d3 --save
```

Ensuite, nous allons créer un template simplifié, et le stocker dans le fichier « library/template_html_d3.js ».

```
module.exports = function template (title='no title', divcontent='') {

return  `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>${title}</title>
        <meta charset="UTF-8">
        <script src="/js/d3.min.js"></script>
    </head>
    <body>
        <div class="container">
            <div id="body-page">${divcontent}</div>
        </div>
    </body>
</html>`;

};
```

Nous allons également créer un répertoire « data » dans notre projet, et y stocker un fichier text « data.tsv ». Copiez-collez dans ce fichier au moins une partie du contenu du fichier donné en exemple sur le site de D3 :

data.tsv

date	close
24-Apr-07	93.24
25-Apr-07	95.35
26-Apr-07	98.84
27-Apr-07	99.92
30-Apr-07	99.80
1-May-07	99.47
2-May-07	100.39
3-May-07	100.40
4-May-07	100.81
7-May-07	103.92
8-May-07	105.06
9-May-07	106.88
10-May-07	107.34
11-May-07	108.74
14-May-07	109.36
15-May-07	107.52
16-May-07	107.34
17-May-07	109.44
18-May-07	110.02
21-May-07	111.98

Nous allons prendre l'essentiel du code générant le graphe qui nous intéresse, et le copier-coller dans un script que je vous propose d'appeler « library/D3_graphe1.js ». Le code se trouve sur la page suivante.

```
/*
 * Exemple de graphe pris dans les exemples de D3 :
 * https://bl.ocks.org/mbostock/3883195
 */
var gengraph = function () {

    var svg = d3.select("svg"),
            margin = {top: 20, right: 20, bottom: 30, left: 50},
            width = +svg.attr("width") - margin.left - margin.right,
            height = +svg.attr("height") - margin.top - margin.bottom,
            g = svg.append("g").attr("transform", "translate(" + margin.left + "," +
                 margin.top + ")");

    var parseTime = d3.timeParse("%d-%b-%y");

    var x = d3.scaleTime().rangeRound([0, width]);

    var y = d3.scaleLinear().rangeRound([height, 0]);

    var area = d3.area()
            .x(function (d) {
                return x(d.date);
            })
            .y1(function (d) {
                return y(d.close);
            });

    d3.tsv("data/data_lite.tsv", function (d) {
        d.date = parseTime(d.date);
        d.close = +d.close;
        return d;
    }, function (error, data) {
        if (error)
            throw error;

        x.domain(d3.extent(data, function (d) {
            return d.date;
        }));
        y.domain([0, d3.max(data, function (d) {
                return d.close;
            })]);
        area.y0(y(0));

        g.append("path")
                .datum(data)
                .attr("fill", "steelblue")
                .attr("d", area);

        g.append("g")
                .attr("transform", "translate(0," + height + ")")
                .call(d3.axisBottom(x));

        g.append("g")
                .call(d3.axisLeft(y))
                .append("text")
                .attr("fill", "#000")
                .attr("transform", "rotate(-90)")
                .attr("y", 6)
                .attr("dy", "0.71em")
                .attr("text-anchor", "end")
                .text("Price ($)");
```

```
    });

};

window.addEventListener("load", function (event) {
    console.log("DOM chargé, lancement du graphe D3");
    gengraph();
});
```

Le code Javascript que nous venons de créer, nous allons devoir l'envoyer au navigateur, aussi nous nous contenterons de le charger via la fonction « readFileSync » du module « fs », et de le coller au sein de la page HTML envoyée au navigateur. L'utilisation de la fonction « readFileSync » se justifie ici par le fait qu'on ne veut surtout pas envoyer la page vers le navigateur si l'intégralité du fichier JS n'est pas en place. De plus, il s'agit d'un petit fichier, son chargement en mémoire va être très rapide, alors nous n'avons pas besoin d'utiliser un chargement asynchrone.

Voici le code du script principal :

```
const EXPRESS = require('express');
var app = EXPRESS();
const FS = require('fs');

// préparation des redirections pour les routes statiques
app.use('/', EXPRESS.static(__dirname + '/www'));
app.use('/js', EXPRESS.static(__dirname + '/node_modules/d3/build'));
app.use('/data', EXPRESS.static(__dirname + '/data'));

var template_page = require('./library/template_html_d3.js') ;

var d3_graphe_src = FS.readFileSync('./library/D3_graphe1.js');

var d3_for_client = `
<svg width="960" height="500"></svg>
<script>
${d3_graphe_src}
</script>\n`;

app.get('/', function (req, res, next) {
    console.log('arret / route 1');
    res.end(template_page('page hello world', d3_for_client));
});

app.listen(8080);
```

3.7.2 P5.js

Conçu par des développeurs de la Processing Foundation, P5.js est un portage en Javascript du framework de programmation Processing.

Pour ceux qui ne le connaissent pas, Processing est développé en Java, c'est un framework très dédié à la programmation de graphismes, animés ou pas. Conçu par des artistes pour les artistes, Processing est un framework puissant et polyvalent, que l'on peut utiliser aussi pour la Dataviz, et même pour du développement d'applications mobiles.

P5.js est une réécriture de Processing en Javascript. Il s'exécute dans un navigateur et s'appuie sur l'API Canvas du HTML5 pour générer des graphismes, sur le même principe que Processing.

Les scripts pour P5 et Processing sont appelés des sketchs, et comme P5 s'appuie sur Javascript, et que Processing s'appuie sur Java, il existe quelques différences dans la manière d'écrire un sketch sur les deux environnement. Début 2017, j'avais rédigé une petite étude sur les difficultés que l'on peut rencontrer dans le portage d'un sketch Processing vers P5.js. Cette étude est téléchargeable librement sur le dépôt suivant :

https://github.com/gregja/p5Migration

Si vous avez envie de vous initier à P5.js, vous pouvez aussi vous reporter au document « Tuto_P5_Premiers_Pas.pdf » qui se trouve dans le dépôt suivant :

https://github.com/gregja/JSCorner

Comme je sais que beaucoup de « creative coders » s'intéressent à P5.js ainsi qu'à Node.js, j'ai eu envie de leur consacrer ce chapitre en montrant comment on peut intégrer P5.js dans une application Node.js.

Il faut souligner que P5.js est un formidable outil pour la simulation, ce chapitre va vous le démontrer.

Pour la rédaction de ce chapitre, je me suis largement inspiré d'un exemple de sketch trouvé dans un article publié par Tim Jones, en 2010, sur le site Developerworks :
https://www.ibm.com/developerworks/library/os-datavis/

L'article de Tim Jones s'intitule :
« Data visualization with Processing, Part 1, An introduction to the language and environment »

Tim Jones a publié 2 autres articles, tout aussi intéressants dont je vous donne les références ci-dessous :

- « Data visualization with Processing, Part 2, Intermediate data visualization using interfaces, objects, images, and applications »
https://www.ibm.com/developerworks/library/os-datavis2/

- « Data visualization with Processing, Part 3, 2-D, 3-D, physics, and networking »
https://www.ibm.com/developerworks/library/os-datavis3/index.html

Je vais pour ma part me concentrer sur un chapitre du premier dossier de Tim Jones, consacré aux automates cellulaires. C'est un sujet qui intéressait beaucoup les développeurs dans les années 80, on trouvait de temps en temps des articles sur ce sujet dans la presse informatique. Ce n'est plus guère le cas aujourd'hui, et c'est bien dommage.

Dans le chapitre consacré aux automates cellulaires, Tim Jones nous propose d'implémenter le modèle de la forêt en feu (en anglais : « forest-fire model »). Tim nous explique ceci :

« Maintenant, regardons quelques simulations construites avec Processing. Le premier est un automate cellulaire 2-D qui implémente le « forest-fire model ». Ce modèle, de Chopard et Droz («modélisation des systèmes physiques par automates cellulaires») fournit un système simple illustrant la croissance des arbres dans une grille, et la propagation du feu résultant d'un coup de foudre. Cette simulation consiste en un ensemble de règles simples définies comme suit :
• Sur un site vide (brun), un arbre croît avec la probabilité « pGrowth ».
• Un arbre devient un arbre en feu (rouge), si au moins un de ses voisins est en train de brûler.
• Un arbre en feu (rouge) devient un site vide (brun).
• Un arbre sans voisins brûlants devient un arbre en feu avec la probabilité « pBurn ». Cela se produit, par exemple, à la suite d'un coup de foudre.

Ces règles sont codées dans la fonction update(), qui itère à travers l'espace 2-D pour déterminer les changements d'états selon les règles que nous venons d'énoncer.

Notez que l'espace 2D est en fait 3-D parce que vous maintenez deux copies de l'espace, une pour l'itération en cours, et une pour la dernière itération. On fait cela pour éviter de polluer l'espace entre chaque changement. On dispose donc de 2 espaces virtuels, un dédié à l'affichage, et un au calcul et à l'application des règles. Le swap entre les 2 espaces se fait à chaque changement de génération.

Pour l'essentiel, cette application utilise très peu de fonctions graphiques. Quelques couleurs sont définies pour l'espace, le trait est utilisé pour changer les couleurs et le point est utilisé pour dessiner un pixel.
En utilisant le modèle de Processing, la fonction draw() appelle la fonction update() pour appliquer les règles. L'espace est redessiné à un rythme proche de 60 images/seconde.

Très intéressé par l'article de Tim Jones, j'ai converti son sketch Processing en une version P5.js. La conversion n'était pas très compliquée, il y a juste une chose qui m'a embêtée, c'est la création du tableau multi-dimensionnel au début du sketch :

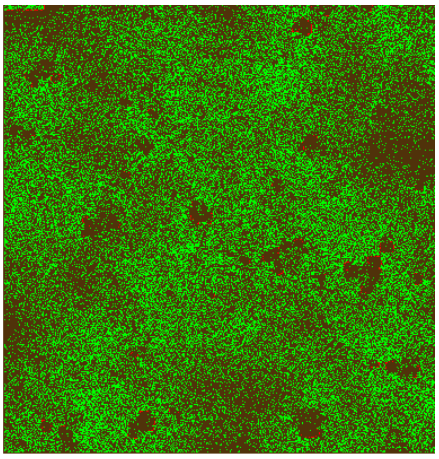
- en Java, on a ça :

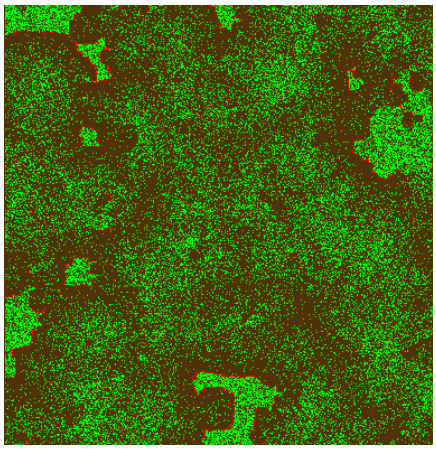
```
int[][][] pix = new int[2][400][400];
```

- pour la conversion en Javascript, et pour ne pas me prendre la tête, j'ai créé une petite fonction qui crée un tableau à 2 dimensions, fonction que j'ai appelé 2 fois. Cela donne ceci :

```
var matrix = function(numrows, numcols, initial) {
    var arr = [];
    for (var i = 0; i < numrows; ++i) {
        var columns = [];
        for (var j = 0; j < numcols; ++j) {
            columns[j] = initial;
        }
        arr[i] = columns;
    }
    return arr;
};
var pix = [];
pix.push(matrix(400, 400, 0));
pix.push(matrix(400, 400, 0));
```

Après mise au point du sketch, voici deux images générées, à 2 stades différents de l'évolution des feux de forêt :





Vous êtes intéressé ? Alors allons y pour quelques minutes d'intégration d'un feu de forêt dans Node.js.

Voici le sketch P5.js à créer dans un script « library/forest-fire-model.js » :

```
/**
 * Adaptation en P5.js d'un sketch proposé par Tim Jones dans le dossier
 * suivant :
 *
 *  "Data visualization with Processing, Part 1, An introduction
 *  to the language and environment"
 *
 * https://www.ibm.com/developerworks/library/os-datavis/
 */

/**
 * Fonction permettant de créer des tableaux multi-dimensionnels en JS
 */
var matrix = function(numrows, numcols, initial) {
    var arr = [];
    for (var i = 0; i < numrows; ++i) {
        var columns = [];
        for (var j = 0; j < numcols; ++j) {
            columns[j] = initial;
        }
        arr[i] = columns;
    }
    return arr;
};

// tableau des 2 espaces (de dessin et de calcul)
var pix = [];

var toDraw = 0;
var tree = 0;
var burningTree = 1;
var emptySite = 2;
var x_limit = 400;
var y_limit = 400;
var col_brown = null;  // brown
var col_red = null; // red;
var col_green = null; // green
var pGrowth = 0.01;
var pBurn = 0.00006;

function setup() {
    createCanvas(x_limit, y_limit);

    col_brown = color(80, 50, 10);
    col_red = color(255, 0, 0);
    col_green = color(0, 255, 0);

    frameRate(60);

    // Aire peuplée initialement d'arbres
    // Les 2 lignes ci-dessous équivalent en Java à :
    //     int[][][] pix = new int[2][400][400];
    pix.push(matrix(x_limit, y_limit, tree));
    pix.push(matrix(x_limit, y_limit, tree));
```

```
    /* Initialize to all empty sites */
    for (var x = 0; x < x_limit; x++) {
        for (var y = 0; y < y_limit; y++) {
            pix[toDraw][x][y] = emptySite;
        }
    }
}

function draw() {
    update();
    for (var x = 0; x < x_limit; x++) {
        for (var y = 0; y < y_limit; y++) {
            //console.log(toDraw, x, y);
            if (pix[toDraw][x][y] === tree)
                stroke(col_green);
            else if (pix[toDraw][x][y] === burningTree)
                stroke(col_red);
            else
                stroke(col_brown);
            point(x, y);
        }
    }
    toDraw = (toDraw === 0) ? 1 : 0;
}

var prob = function (p) {
    if (random(0, 1) < p) {
        return true;
    } else {
        return false;
    }
};

function update() {
    var x, y, dx, dy, cell, chg, burningTreeCount;
    var toCompute = (toDraw === 0) ? 1 : 0;
    for (x = 1, xmax = x_limit - 1; x < xmax; x++) {
        for (y = 1, ymax = y_limit - 1; y < ymax; y++) {
            cell = pix[toDraw][x][y];
            // Survey area for burning trees
            burningTreeCount = 0;
            for (dx = -1; dx < 2; dx++) {
                for (dy = -1; dy < 2; dy++) {
                    if ((dx === 0) && (dy === 0)) {
                        continue;
                    } else {
                        if (pix[toDraw][x + dx][y + dy] === burningTree) {
                            burningTreeCount++;
                        }
                    }
                }
                // Determine next state
                if (cell === burningTree) chg = emptySite;
                else if ((cell === emptySite) && (prob(pGrowth))) chg = tree;
                else if ((cell === tree) && (prob(pBurn))) chg = burningTree;
                else if ((cell === tree) && (burningTreeCount > 0)) chg = burningTree;
                else chg = cell;

                pix[toCompute][x][y] = chg;
            }
```

```
        }
    }
}
```

Il ne nous reste plus qu'à installer le package de P5.js dans Node.js :

```
npm install p5 --save
```

… et à créer le script serveur, en utilisant la même technique que nous avions vue dans le chapitre consacré à D3.js :

```
const EXPRESS = require('express');
var app = EXPRESS();
const FS = require('fs');

// préparation des redirections pour les routes statiques
app.use('/', EXPRESS.static(__dirname + '/www'));
app.use('/js', EXPRESS.static(__dirname + '/node_modules/p5/lib'));

var template_page = require('./library/template_html_p5.js') ;

var graphe_src = FS.readFileSync('./library/forest-fire-model.js');

var code_for_client = `
<script>
${graphe_src}
</script>\n`;

app.get('/', function (req, res, next) {
    res.end(template_page('Forest Fire Model', code_for_client));
});

app.listen(8080);
```

On notera pour finir que P5.js propose un large panel de fonctions, dont 4 peuvent être utiles pour charger des données en provenance d'un serveur :
loadJSON()
loadStrings()
loadTable() (pour le chargement de fichier CSV notamment)
loadXML()

Le site officiel de P5.js :
https://p5js.org/reference/

Le site officiel de Processing :
https://processing.org/reference/

3.8 Creative Coding

3.8.1 P5.js et norme OSC

En novembre 2017, j'ai participé à un hackaton organisé par l'Adami, et j'ai eu l'occasion de développer une petite application s'appuyant sur P5.js pour la partie graphique.

Dans cette application, je devais récupérer dans mon navigateur des données transmises selon la norme OSC, pour les retraduire en informations graphiques avec P5.js.

Pour ceux qui ne connaîtraient pas, la norme OSC (Open Sound Control) est l'héritière de la norme MIDI utilisée par les musiciens depuis 1981. La norme MIDI est encore utilisée aujourd'hui, mais elle tend à être remplacée progressivement par OSC, plus moderne et plus puissante :
https://fr.wikipedia.org/wiki/Open_Sound_Control

J'avais donc besoin de trouver un moyen de récupérer des données à la norme OSC dans mon sketch P5.js. Après quelques recherches sur le web, j'ai trouvé le projet « osc-web » qui semblait bien correspondre à mon besoin :
https://github.com/automata/osc-web

… mais la documentation du projet était un peu trop succincte à mon goût. Fort heureusement, en poursuivant mes recherches, je suis tombé sur un article de Jürgen Röhm, qui m'a été très utile pour mettre en route le projet « osc-web » et intégrer la réception des messages OSC dans mon sketch P5.js.
https://www.jroehm.com/2015/10/a-simple-guide-to-use-osc-in-the-browser/

Pour résumer brièvement le principe, le projet « osc-web » fonctionne comme un serveur Node.js, indépendant de l'application P5. Le serveur « osc-web » se démarre en lance la commande « node bridge.js », et il se met en mode réception des données OSC transmises depuis l'extérieur. Dans mon cas, les données OSC étaient transmises à mon PC via une liaison Wifi et un port UDP. Le serveur « osc-web » servait donc d'intermédiaire pour réceptionner sur ma machine les signaux OSC, et les retransmettre à une autre application tournant sur le même PC avec une autre instance de serveur Node.js (pour la partie graphique pilotée par P5.js).

Je n'ai malheureusement pas le temps de rentrer plus dans le détail maintenant. J'essaierai de la faire dans une prochaine version de ce support.

4 Conclusion

Dans sa version 1.2, ce support d'introduction à Node.js est suffisamment complet pour permettre à toute personne motivée de se lancer. C'est pourquoi je le mets en ligne sur mon dépôt Github à partir de cette version.

Mais Node.js est un sujet inépuisable, et il y a beaucoup de sujets que j'ai laissés en standbye, faute de temps. Je compte bien faire évoluer ce support rapidement, d'autant que le sujet est passionnant.

En à peine 9 ans d'existence, Node.js a rattrapé et même dépassé les capacités de la plupart des autres langages. Plus puissant, plus polyvalent que toutes les autres solutions existantes, Node.js est aujourd'hui utilisé comme socle par un nombre impressionnant de projets et frameworks. Ce nombre est difficilement quantifiable, mais ce qui est sûr c'est qu'il est en constante augmentation. Cela ne va pas sans poser des difficultés, car il est facile de se noyer face à l'offre foisonnante de projets, dont certains ne sont pas nécessairement pérennes.

Doté de ce formidable langage qu'est Javascript, et avec le concours de quelques uns des meilleurs développeurs de la planète, Node.js a investi avec succès des domaines aussi variés que le développement web, le développement desktop, le développement mobile, l'IOT (Internet des Objets), et même le NLP (Natural Language Processing).

Le principal domaine où Node.js est mal positionné – et sans doute pas positionnable à l'heure actuelle - c'est celui du Big Data. Pour ce sujet en particulier, des langages comme Python, R ou Scala sont certainement mieux placés pour répondre aux besoins. Mais sachant qu'on a déjà un portage de Scala en Javascript (le projet Scala.js), je ne serais pas surpris de voir apparaître de nouveaux projets Node.js venant bousculer la hiérarchie actuelle.

Avec son modèle objet particulier offrant aux objets la capacité de s'auto-modifier dynamiquement, je pense que Javascript a encore une carte à jouer dans le domaine du Machine Learning et de l'IA, même si pour le moment il semble quelque peu absent de ces domaines de recherche.

En tout cas, ce qui est certain, c'est qu'avec Node.js, on ne va pas s'ennuyer dans les mois et les années à venir.

5 Annexe

5.1 Bibliographie

Node.js ayant connu de nombreux changements, certains ouvrages peuvent être légèrement dépassés sur certains sujets. J'ai néanmoins choisi de conserver dans cette sélection des ouvrages dont certains datent de 2012, quand les explications données sur certains sujets difficiles sont particulièrement bonnes. Je recommande toutefois de privilégier les ouvrages les plus récents, sauf s'ils ne couvrent pas le sujet qui vous intéresse :

- Learning Node.js, de Marc Wandschneider, Addison-Wesley (2013)

- Node.js 8 The Right Way, de Jim R. Wilson, Pragmatic Programmers (2018)

- Node Cookbook, de David Mark Clements, Packt Publishing (2012)

Les éditions Diamond avaient publié, pendant l'été 2016, un hors série consacré à Node.js. Du fait de l'évolution très rapide de Node.js, certains articles de ce numéro sont un peu datés, mais il demeure une bonne introduction à des sujets tels que Browserify, Electron, la création d'exécutables, ou encore la sécurité :

https://proboutique.ed-diamond.com/les-guides/1060-gnulinux-magazine-hs-85.html

Deux livres en français à signaler, parus chez Eyrolles. Je ne les ai pas encore lus, mais ils semblent intéressants et méritent sans doute le détour :

Programmation avec Node.js, Express.js et MongoDB
de Eric Sarrion
Ed Eyrolles (2014)

Comprendre et développer un Chatbot
Messagerie instantanée, paiements en ligne, intelligence artificielle...
de Samuel Ronce
Ed Eyrolles
Collection : Blanche (07/12/2017)

TODO : liste à compléter

5.2 Liens utiles

Quelques liens intéressants à signaler, collectés avant et pendant la préparation de ce support :

https://nodejs.org/en/docs/guides/anatomy-of-an-http-transaction/

https://www.w3schools.com/nodejs/

https://openclassrooms.com/courses/des-applications-ultra-rapides-avec-node-js/tp-le-super-chat

https://nodejs.developpez.com/tutoriels/javascript/debuter-avec-node-js-partie-3/

TODO : liste à compléter

5.3 Exemple de script avec formulaire en méthode POST

Code source complet de l'exemple présenté au chapitre 2.7 :

```
const HTTP = require('http');
const URL = require('url');
const QUERYSTRING = require('querystring');
const PORT = 8080;
const HOST = '127.0.0.1';

/**
 * Fonction dédiée à la génération du formulaire
 * @returns {String}
 */
var testform = () => `
<form action="message" method="post">
<label>Nom :<input name="nom"></label><br><br>
<label>Prénom :<input name="prenom"></label><br><br>
<input type="submit" name="Valider">
</form>
`;

var errorInfo = (code, res) => {
    //res.statusCode = code;
    res.writeHead(code, {
        'Content-Type': 'text/html'
    });
    res.write(`Error ${code} : page not found`);
    res.end();
};

var template = (divcontent='') => `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>NodeJS exemple 1</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <div>${divcontent}</div>
    </body>
</html>`;

var server = HTTP.createServer().listen(PORT, HOST);

server.on('request', function(req, res) {

    if (req.method === 'GET') {
        if (req.url === '/') {
            res.writeHead(200, {'Content-Type': 'text/html'});
            //var params = URL.parse(req.url, true);
            var message = template(testform());
            res.write(message);
            res.end();
        } else {
            errorInfo(404, res);
```

```
        }
    }

    if (req.method === 'POST') {
        var datas = {};
        var post_data = '';
        if (req.url === '/message') {

            // Réception des données provenant du formulaire
            req.on('data', function (data) {
                post_data += data;
            });

            // Fin de la réception, affichage de la page finale
            req.on('end', function () {
                datas = QUERYSTRING.parse(post_data);
                res.writeHead(200, {'Content-Type': 'text/html'});
                // Console.log pour comparaison de post_data et datas (si besoin)
                //  console.log("Body end : " + post_data);
                //  console.log(datas);
                var message = template(`Bonjour ${datas.prenom} ${datas.nom} `);
                res.write(message);
                res.end();
            });
        } else {
            errorInfo(404, res);
        }
    }

});

console.log('Serveur tourne sur '+HOST+':'+PORT);
```

5.4 Exemple de serveur Express

Solution de l'exercice proposé au chapitre 3.5.3 :

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

var template = (title='no title', divcontent='') => `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>${title}</title>
        <meta charset="UTF-8">
    </head>
    <body>
        <div>${divcontent}</div>
    </body>
</html>`;

app.get('/', function (req, res, next) {
    res.end(template('page hello world', 'hello world'));
    console.log('arret / route 1');
});

app.get("/products/:productid", function (req, res, next) {
    var prod_id = parseInt(req.params.productid);
    res.end(template(
            'Produit',
            `Produit : ${prod_id}`
                    ));
    console.log('arret / route 2');
});

app.get("/products/:productid/category/:categid", function (req, res, next) {
    var prod_id = parseInt(req.params.productid);
    var cat_id = parseInt(req.params.categid);
    res.end(template(
            'Produit/Catégorie',
            `Produit : ${prod_id} ; Categorie : ${cat_id}`
                    ));
    console.log('arret / route 3');
});

app.get("/apropos", function (req, res, next) {
    res.sendfile('./apropos.html');
    console.log('arret / page "à propos" ');

});

app.listen(8080);
```

5.5 Exemple de serveur Express avec formulaire

Solution de l'exercice proposé au chapitre 3.5.5 :

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

const BODYPARSER = require('body-parser');
//app.use(BODYPARSER.json()); // support json encoded bodies
app.use(BODYPARSER.urlencoded({ extended: true })); // support encoded bodies

/**
 * Fonction dédiée à la génération du formulaire
 * @returns {String}
 */
var testform = function(values, errors=[]) {
    var error_list = '';
    if (errors.length > 0) {
        error_list = '<fieldset><legend>Liste des erreurs</legend>\n';
        for (var i=0, imax=errors.length; i<imax ; i++) {
            error_list += errors[i] + '<br>\n';
        }
        error_list += '</fieldset><br>\n';
    }

    return `${error_list}<form action="/testform" method="post">
    <label>Nom :<input name="nom" value="${values.nom}"></label><br><br>
    <label>Prénom :<input name="prenom" value="${values.prenom}"></label><br><br>
    <input type="submit" name="Valider">
    </form>
    `;
};

/**
 * Fonction "template" de page HTML
 * @param {type} title
 * @param {type} divcontent
 * @returns {String}
 */
var template = (title='no title', divcontent='') => `<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>${title}</title>
        <meta charset="UTF-8">
    </head>
    <body>
        <div>${divcontent}</div>
    </body>
</html>`;

app.get('/', function (req, res, next) {
    res.end(template('page hello world', 'hello world'));
    console.log('arret / route 1');
});
```

```
app.get("/testform", function (req, res, next) {
    var datas = {};
    datas.nom = '';
    datas.prenom = '';
    res.end(template(
            'Formulaire',
            testform(datas)
        ));

    console.log('arret / route 2');
});

app.post("/testform", function (req, res, next) {
    var datas = {};
    //console.log(req.body);
    datas.nom = req.body.nom ? req.body.nom.trim() : '';
    datas.prenom = req.body.prenom ? req.body.prenom.trim() : '';

    var erreurs = [];
    if (datas.nom === '') {
        erreurs.push('Nom obligatoire');
    }
    if (datas.prenom === '') {
        erreurs.push('Prénom obligatoire');
    }

    if (erreurs.length > 0) {
        res.end(template(
                'Formulaire',
                testform(datas, erreurs)
            ));
    } else {
        res.end(template(
                'Merci',
                 `Merci ${datas.prenom} ${datas.nom}, votre demande a bien été prise en
compte`
            ));
    }
    console.log('arret / route 3');
});

app.use(function(req, res, next){
    res.status(404);
    res.end(template(
            'Erreur 404',
            'Désolé, page non trouvée'
    ));
});

app.listen(8080);
```

5.6 Table SQL des pays pour liste avec pagination

La table SQL ci-dessous va nous être utile pour implémenter un exemple de liste avec Express.

```
CREATE TABLE `countries` (
  `id` int(6) NOT NULL auto_increment,
  `codinter` char(3) NOT NULL default '',
  `codfra` char(2) NOT NULL default '',
  `countryname` varchar(250) NOT NULL default '',
  PRIMARY KEY  (`id`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8 AUTO_INCREMENT=1 ;

INSERT INTO countries (codinter, codfra, countryname)
VALUES
 ('AFG ', 'AF ', 'AFGHANISTAN'),
 ('ZAF ', 'ZA ', 'AFRIQUE DU SUD'),
 ('ALA ', 'AX ', 'ALAND, ILES'),
 ('ALB ', 'AL ', 'ALBANIE'),
 ('DZA ', 'DZ ', 'ALGERIE '),
 ('DEU ', 'DE ', 'ALLEMAGNE'),
 ('AND ', 'AD ', 'ANDORRE'),
 ('AGO ', 'AO ', 'ANGOLA'),
 ('AIA ', 'AI ', 'ANGUILLA'),
 ('ATA ', 'AQ ', 'ANTARCTIQUE'),
 ('ATG ', 'AG ', 'ANTIGUA-ET-BARBUDA'),
 ('ANT ', 'AN ', 'ANTILLES NEERLANDAISES '),
 ('SAU ', 'SA ', 'ARABIE SAOUDITE'),
 ('ARG ', 'AR ', 'ARGENTINE'),
 ('ARM ', 'AM ', 'ARMENIE'),
 ('ABW ', 'AW ', 'ARUBA'),
 ('AUS ', 'AU ', 'AUSTRALIE'),
 ('AUT ', 'AT ', 'AUTRICHE'),
 ('AZE ', 'AZ ', 'AZERBAIDJAN'),
 ('BHS ', 'BS ', 'BAHAMAS'),
 ('BHR ', 'BH ', 'BAHREIN'),
 ('BGD ', 'BD ', 'BANGLADESH'),
 ('BRB ', 'BB ', 'BARBADE'),
 ('BLR ', 'BY ', 'BELARUS'),
 ('BEL ', 'BE ', 'BELGIQUE'),
 ('BLZ ', 'BZ ', 'BELIZE'),
 ('BEN ', 'BJ ', 'BENIN '),
 ('BMU ', 'BM ', 'BERMUDES'),
 ('BTN ', 'BT ', 'BHOUTAN'),
 ('BOL ', 'BO ', 'BOLIVIE'),
 ('BIH ', 'BA ', 'BOSNIE-HERZEGOVINE '),
 ('BWA ', 'BW ', 'BOTSWANA'),
 ('BVT ', 'BV ', 'BOUVET, ILE '),
 ('BRA ', 'BR ', 'BRESIL '),
 ('BRN ', 'BN ', 'BRUNEI DARUSSALAM '),
 ('BGR ', 'BG ', 'BULGARIE'),
 ('BFA ', 'BF ', 'BURKINA FASO'),
 ('BDI ', 'BI ', 'BURUNDI'),
 ('CYM ', 'KY ', 'CAIMANES, ILES '),
 ('KHM ', 'KH ', 'CAMBODGE'),
 ('CMR ', 'CM ', 'CAMEROUN'),
 ('CAN ', 'CA ', 'CANADA'),
 ('CPV ', 'CV ', 'CAP-VERT'),
 ('CAF ', 'CF ', 'CENTRAFRICAINE, REPUBLIQUE '),
 ('CHL ', 'CL ', 'CHILI'),
 ('CHN ', 'CN ', 'CHINE'),
 ('CXR ', 'CX ', 'CHRISTMAS, ILE '),
 ('CYP ', 'CY ', 'CHYPRE'),
 ('CCK ', 'CC ', 'COCOS (KEELING), ILES '),
 ('COL ', 'CO ', 'COLOMBIE'),
 ('COM ', 'KM ', 'COMORES'),
 ('COG ', 'CG ', 'CONGO'),
 ('COD ', 'CD ',
      'CONGO, LA REPUBLIQUE DEMOCRATIQUE DU '),
 ('COK ', 'CK ', 'COOK, ILES'),
 ('KOR ', 'KR ', 'COREE, REPUBLIQUE DE '),
 ('PRK ', 'KP ',
      'COREE, REPUBLIQUE POPULAIRE DEMOCRATIQUE DE '),
 ('CRI ', 'CR ', 'COSTA RICA'),
 ('CIV ', 'CI ', 'COTE D''IVOIRE '),
 ('HRV ', 'HR ', 'CROATIE'),
 ('CUB ', 'CU ', 'CUBA'),
```

```
('DNK ', 'DK ', 'DANEMARK'),
('DJI ', 'DJ ', 'DJIBOUTI'),
('DOM ', 'DO ', 'DOMINICAINE, REPUBLIQUE'),
('DMA ', 'DM ', 'DOMINIQUE'),
('EGY ', 'EG ', 'EGYPTE '),
('SLV ', 'SV ', 'EL SALVADOR'),
('ARE ', 'AE ', 'EMIRATS ARABES UNIS'),
('ECU ', 'EC ', 'EQUATEUR '),
('ERI ', 'ER ', 'ERYTHREE'),
('ESP ', 'ES ', 'ESPAGNE'),
('EST ', 'EE ', 'ESTONIE'),
('USA ', 'US ', 'Etats-Unis'),
('ETH ', 'ET ', 'ETHIOPIE'),
('FLK ', 'FK ', 'FALKLAND, ILES (MALVINAS) '),
('FRO ', 'FO ', 'FEROE, ILES '),
('FJI ', 'FJ ', 'FIDJI'),
('FIN ', 'FI ', 'FINLANDE'),
('FRA ', 'FR ', 'FRANCE'),
('GAB ', 'GA ', 'GABON'),
('GMB ', 'GM ', 'GAMBIE'),
('GEO ', 'GE ', 'GEORGIE'),
('SGS ', 'GS ',
     'GEORGIE DU SUD ET LES ILES SANDWICH DU SUD'),
('GHA ', 'GH ', 'GHANA'),
('GIB ', 'GI ', 'GIBRALTAR'),
('GRC ', 'GR ', 'GRECE '),
('GRD ', 'GD ', 'GRENADE'),
('GRL ', 'GL ', 'GROENLAND'),
('GLP ', 'GP ', 'GUADELOUPE'),
('GUM ', 'GU ', 'GUAM'),
('GTM ', 'GT ', 'GUATEMALA'),
('GGY ', 'GG ', 'GUERNESEY'),
('GIN ', 'GN ', 'GUINEE '),
('GNB ', 'GW ', 'GUINEE BISSAU '),
('GNQ ', 'GQ ', 'GUINEE EQUATORIALE '),
('GUY ', 'GY ', 'GUYANA'),
('GUF ', 'GF ', 'GUYANE FRANCAISE '),
('HTI ', 'HT ', 'HAITI '),
('HMD ', 'HM ',
     'HEARD, ILE ET MCDONALD, ILES'),
('HND ', 'HN ', 'HONDURAS'),
('HKG ', 'HK ', 'HONG KONG'),
('HUN ', 'HU ', 'HONGRIE'),
('IMN ', 'IM ', 'ILE DE MAN '),
('UMI ', 'UM ',
     'ILES MINEURES ELOIGNEES DES ETATS-UNIS '),
('VGB ', 'VG ',
     'ILES VIERGES BRITANNIQUES'),
('VIR ', 'VI ',
     'ILES VIERGES DES ETATS-UNIS '),
('IND ', 'IN ', 'INDE'),
('IDN ', 'ID ', 'INDONESIE '),
('IRN ', 'IR ',
     'IRAN, REPUBLIQUE ISLAMIQUE D'' '),
('IRQ ', 'IQ ', 'IRAQ'),
('IRL ', 'IE ', 'IRLANDE'),
('ISL ', 'IS ', 'ISLANDE'),
('ISR ', 'IL ', 'ISRAEL '),
('ITA ', 'IT ', 'ITALIE'),
('JAM ', 'JM ', 'JAMAIQUE '),
('JPN ', 'JP ', 'JAPON'),
('JEY ', 'JE ', 'JERSEY'),
('JOR ', 'JO ', 'JORDANIE'),
('KAZ ', 'KZ ', 'KAZAKHSTAN'),
('KEN ', 'KE ', 'KENYA'),
('KGZ ', 'KG ', 'KIRGHIZISTAN'),
('KIR ', 'KI ', 'KIRIBATI'),
('KWT ', 'KW ', 'KOWEIT '),
('LAO ', 'LA ',
     'LAOS, REPUBLIQUE DEMOCRATIQUE POPULAIRE'),
('LSO ', 'LS ', 'LESOTHO'),
('LVA ', 'LV ', 'LETTONIE'),
('LBN ', 'LB ', 'LIBAN'),
('LBR ', 'LR ', 'LIBERIA '),
('LBY ', 'LY ',
     'LIBYENNE, JAMAHIRIYA ARABE'),
('LIE ', 'LI ', 'LIECHTENSTEIN'),
('LTU ', 'LT ', 'LITUANIE'),
('LUX ', 'LU ', 'LUXEMBOURG'),
('MAC ', 'MO ', 'MACAO'),
('MKD ', 'MK ',
     'MACEDOINE, L''EX-REPUBLIQUE YOUGOSLAVE DE '),
('MDG ', 'MG ', 'MADAGASCAR'),
('MYS ', 'MY ', 'MALAISIE'),
('MWI ', 'MW ', 'MALAWI'),
('MDV ', 'MV ', 'MALDIVES'),
('MLI ', 'ML ', 'MALI'),
('MLT ', 'MT ', 'MALTE'),
('MNP ', 'MP ',
     'MARIANNES DU NORD, ILES'),
('MAR ', 'MA ', 'MAROC'),
```

```
('MHL ', 'MH ', 'MARSHALL, ILES'),
('MTQ ', 'MQ ', 'MARTINIQUE'),
('MUS ', 'MU ', 'MAURICE'),
('MRT ', 'MR ', 'MAURITANIE'),
('MYT ', 'YT ', 'MAYOTTE'),
('MEX ', 'MX ', 'MEXIQUE'),
('FSM ', 'FM ',
     'MICRONESIE, ETATS FEDERES DE '),
('MDA ', 'MD ', 'MOLDOVA'),
('MCO ', 'MC ', 'MONACO'),
('MNG ', 'MN ', 'MONGOLIE'),
('MNE ', 'ME ', 'MONTENEGRO'),
('MSR ', 'MS ', 'MONTSERRAT'),
('MOZ ', 'MZ ', 'MOZAMBIQUE'),
('MMR ', 'MM ', 'MYANMAR'),
('NAM ', 'NA ', 'NAMIBIE'),
('NRU ', 'NR ', 'NAURU'),
('NPL ', 'NP ', 'NEPAL '),
('NIC ', 'NI ', 'NICARAGUA'),
('NER ', 'NE ', 'NIGER'),
('NGA ', 'NG ', 'NIGERIA '),
('NIU ', 'NU ', 'NIUE '),
('NFK ', 'NF ', 'NORFOLK, ILE '),
('NOR ', 'NO ', 'NORVEGE '),
('NCL ', 'NC ', 'NOUVELLE-CALEDONIE'),
('NZL ', 'NZ ', 'NOUVELLE-ZELANDE '),
('IOT ', 'IO ',
     'OCEAN INDIEN, TERRITOIRE BRITANNIQUE DE L'' '),
('OMN ', 'OM ', 'OMAN'),
('UGA ', 'UG ', 'OUGANDA'),
('UZB ', 'UZ ', 'OUZBEKISTAN '),
('PAK ', 'PK ', 'PAKISTAN'),
('PLW ', 'PW ', 'PALAOS'),
('PSE ', 'PS ',
     'PALESTINIEN OCCUPE, TERRITOIRE'),
('PAN ', 'PA ', 'PANAMA'),
('PNG ', 'PG ',
     'PAPOUASIE-NOUVELLE-GUINEE '),
('PRY ', 'PY ', 'PARAGUAY'),
('NLD ', 'NL ', 'PAYS-BAS'),
('PER ', 'PE ', 'PEROU '),
('PHL ', 'PH ', 'PHILIPPINES'),
('PCN ', 'PN ', 'PITCAIRN'),
('POL ', 'PL ', 'POLOGNE'),
('PYF ', 'PF ',
     'POLYNESIE FRANCAISE '),
('PRI ', 'PR ', 'PORTO RICO'),
('PRT ', 'PT ', 'PORTUGAL'),
('QAT ', 'QA ', 'QATAR'),
('REU ', 'RE ', 'REUNION '),
('ROU ', 'RO ', 'ROUMANIE'),
('GBR ', 'GB ', 'ROYAUME-UNI'),
('RUS ', 'RU ',
     'RUSSIE, FEDERATION DE '),
('RWA ', 'RW ', 'RWANDA'),
('ESH ', 'EH ', 'SAHARA OCCIDENTAL'),
('BLM ', 'BL ', 'SAINT-BARTHELEMY'),
('KNA ', 'KN ', 'SAINT-KITTS-ET-NEVIS'),
('SMR ', 'SM ', 'SAINT-MARIN'),
('MAF ', 'MF ',
     'SAINT-MARTIN (PARTIE FRANCAISE) '),
('SPM ', 'PM ',
     'SAINT-PIERRE-ET-MIQUELON'),
('VAT ', 'VA ',
     'SAINT-SIEGE (ETAT DE LA CITE DU VATICAN)'),
('VCT ', 'VC ',
     'SAINT-VINCENT-ET-LES GRENADINES'),
('SHN ', 'SH ', 'SAINTE-HELENE'),
('LCA ', 'LC ', 'SAINTE-LUCIE'),
('SLB ', 'SB ', 'SALOMON, ILES '),
('WSM ', 'WS ', 'SAMOA'),
('ASM ', 'AS ', 'SAMOA AMERICAINES '),
('STP ', 'ST ', 'SAO TOME-ET-PRINCIPE '),
('SEN ', 'SN ', 'SENEGAL '),
('SRB ', 'RS ', 'SERBIE'),
('SYC ', 'SC ', 'SEYCHELLES'),
('SLE ', 'SL ', 'SIERRA LEONE'),
('SGP ', 'SG ', 'SINGAPOUR'),
('SVK ', 'SK ', 'SLOVAQUIE'),
('SVN ', 'SI ', 'SLOVENIE'),
('SOM ', 'SO ', 'SOMALIE'),
('SDN ', 'SD ', 'SOUDAN'),
('LKA ', 'LK ', 'SRI LANKA'),
('SWE ', 'SE ', 'SUEDE '),
('CHE ', 'CH ', 'SUISSE'),
('SUR ', 'SR ', 'SURINAME'),
('SJM ', 'SJ ', 'SVALBARD ET ILE JAN MAYEN '),
('SWZ ', 'SZ ', 'SWAZILAND'),
('SYR ', 'SY ', 'SYRIENNE, REPUBLIQUE ARABE '),
('TJK ', 'TJ ', 'TADJIKISTAN'),
('TWN ', 'TW ', 'TA�WAN, PROVINCE DE CHINE'),
```

```
('TZA ', 'TZ ', 'TANZANIE, REPUBLIQUE UNIE DE'),
('TCD ', 'TD ', 'TCHAD'),
('CZE ', 'CZ ', 'TCHEQUE, REPUBLIQUE '),
('ATF ', 'TF ', 'TERRES AUSTRALES FRANCAISES'),
('THA ', 'TH ', 'THAILANDE'),
('TLS ', 'TL ', 'TIMOR-LESTE'),
('TGO ', 'TG ', 'TOGO'),
('TKL ', 'TK ', 'TOKELAU'),
('TON ', 'TO ', 'TONGA'),
('TTO ', 'TT ', 'TRINITE-ET-TOBAGO '),
('TUN ', 'TN ', 'TUNISIE'),
('TKM ', 'TM ', 'TURKMENISTAN '),
('TCA ', 'TC ', 'TURKS ET CAIQUES, ILES'),
('TUR ', 'TR ', 'TURQUIE'),
('TUV ', 'TV ', 'TUVALU'),
('UKR ', 'UA ', 'UKRAINE'),
('URY ', 'UY ', 'URUGUAY'),
('VUT ', 'VU ', 'VANUATU'),
('VEN ', 'VE ', 'VENEZUELA'),
('VNM ', 'VN ', 'VIET NAM'),
('WLF ', 'WF ', 'WALLIS-ET-FUTUNA'),
('YEM ', 'YE ', 'YEMEN '),
('ZMB ', 'ZM ', 'ZAMBIE'),
('ZWE ', 'ZW ', 'ZIMBABWE');
```

5.7 Liste avec recherche et pagination

Source complet du script développé au chapitre 3.6.7 :

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

// préparation des redirections pour les routes statiques
app.use('/', EXPRESS.static(__dirname + '/www'));
app.use('/js', EXPRESS.static(__dirname + '/node_modules/bootstrap/dist/js'));
app.use('/js', EXPRESS.static(__dirname + '/node_modules/jquery/dist'));
app.use('/css', EXPRESS.static(__dirname + '/node_modules/bootstrap/dist/css'));

// chargement du package personnalisé pour la connexion à MariaDB
const DBMODULE = require('./dbconnex');
// initialisation du connecteur à MariaDB
const conn = DBMODULE.init();

var pagination_tpl = require('./library/pagination_bootstrap.js');
var template_page = require('./library/template_html.js') ;
var template_table = require('./library/template_table_head.js') ;
var template_search_form = require('./library/search_form_countries.js') ;

app.get('/', function (req, res, next) {
    console.log('arret / route 1');
    res.end(template_page('page hello world', 'hello world'));
});

var countriesList = function (req, res, next) {

    // Variable qui servira de container pour les critères de
    //  sélection du formulaire, à intégrer dans la route pour assurer
    //  un bon fonctionnement du composant de pagination
    var current_prelink = '';

    // Détermination du numéro de page courant
    //  (indispensable pour le composant de pagination et pour déterminer
    //  les valeurs de la clause LIMIT dans la requête SQL stockée dans "sql2".
    var current_page = 1;
    if (req.params.page) {
        current_page = parseInt(req.params.page);
    }

    // On fixe le nombre de lignes maxi qu'une page peut afficher
    var max_rows_per_page = 10;

    // Préparation des requêtes de comptage et d'affichage
    var sql1 = "SELECT count(*) as comptage FROM countries";
    var sql2 = "SELECT codinter, codfra, countryname FROM countries";

    // Préparation du tableau qui contiendra d'éventuels critères de recherche
    //  à transmettre à SQL
    var sql_params = [];

    // On teste si des critères de recherche ont été saisis dans le formulaire
    //  si oui, on les trouve soit dans "req.query" (lors de la validation du
    //  formulaire, ou dans req.params, pour les paginations
    if ('col_rech' in req.query || req.params.col_rech) {
        var col_rech = '';
```

```
var typ_rech = '';
var lib_rech = '';

// Le paramètre "col_rech" est présent dans req.query lors de la
//  validation du formulaire, ensuite sa valeur est intégrée dans la
//  route et est récupérable via l'objet "req.params". Et c'est
//  pareil pour les paramètres "typ_rech" et "lib_rech".
// Quand l'internaute demande une nouvelle sélection, on force le
//  repositionnement sur la page 1 (sinon la pagination peut être faussée).
if ('col_rech' in req.query) {
    // 1er affichage => repositionnement obligatoire sur page 1
    current_page = 1;
    col_rech = req.query['col_rech'];
    typ_rech = req.query['typ_rech'];
    lib_rech = req.query['lib_rech'];
} else {
    col_rech = req.params.col_rech;
    typ_rech = req.params.typ_rech;
    lib_rech = req.params.lib_rech;
}

// Intégration des paramètres du formulaire dans la route standard
current_prelink += '/col_rech/' + col_rech;
current_prelink += '/typ_rech/' + typ_rech;
current_prelink += '/lib_rech/' + lib_rech;

// Préparation de la clause WHERE qui sera ajoutée aux requêtes SQL
var where = ' WHERE ';

// clause WHERE modulée selon critère indiqué dans le formulaire
if (col_rech === '1') {
    // recherche sur code France
    where += 'codfra';
} else {
    if (col_rech === '2') {
        // recherche sur code international (ISO)
        where += 'codinter';
    } else {
        // recherche sur libellé
        where += 'countryname';
    }
}

// recherche SQL de type "égal", "contient" ou "commence par" ?
if (typ_rech === '1') {
    // recherche de type "égal"
    where += ' = ?';
    sql_params.push(lib_rech);
} else {
    if (typ_rech === '2') {
        // recherche de type "contient"
        where += ' LIKE ?';
        sql_params.push('%'+lib_rech+'%');
    } else {
        // recherche de type "commence par"
        where += ' LIKE ?';
        sql_params.push(lib_rech+'%');
    }
}

// ajout de la clause WHERE à la fin des 2 requêtes SQL
```

```
        sql1 += where ;
        sql2 += where ;

        // quelques mouchards pour contrôle, à supprimer ultérieurement (TODO)
        console.log(sql1);
        console.log(sql2);
        console.log(sql_params);
    }

    // Exécution de la requête de comptage
    conn.execute(sql1, sql_params, function (err, rows, result) {
        //if (err) throw err;
        if (err) {
            console.log("ERREUR SUR REQUETE : " + err.code +
                    " (" + err.message + ")");
            return;
        }

        // récupération du nombre de lignes total
        var nb_lig_total = 0;
        if (rows[0]) {
            nb_lig_total = rows[0].comptage ;
        }

        // tableau contenant l'intitulé des colonnes du tableau HTML
        var liste_cols = ['CODE ISO', 'CODE FRANCE', 'DESCRIPTION'];

        // génération entête de page (incluant formulaire de recherche
        //   et entête de tableau HTML
        var tmp_liste = template_search_form(col_rech, typ_rech, lib_rech) +  +
                template_table('Liste des pays', liste_cols);

        // mouchard à supprimer ultérieurement (TODO)
        console.log('nombre de lignes total renvoyé par SQL1 : '+nb_lig_total);

        if (nb_lig_total === 0) {
            // Affichage du tableau HTML avec une ligne vide
            tmp_liste += '<tr><td colspan="3">Pas de données</td></tr>\n';
            tmp_liste += '</tbody>\n</table>\n';

            res.end(template_page(
                'Formulaire',
                tmp_liste
            ));
        } else {
            // calcul du numéro de ligne SQL correspondant au début
            //   de la page courante (pour intégration dans la clause LIMIT)
            var num_ligne = (current_page - 1) *  max_rows_per_page ;

            // ajout de la clause LIMIT à la requête SQL d'affichage
            sql2 += ` limit ${num_ligne}, ${max_rows_per_page}`;

            // mouchard à supprimer ultérieurement (TODO)
            console.log('requête SQL2 avec clause LIMIT : ' + sql2);

            // Execution de la requête SQL numéro 2 pour l'affichage des pays
            conn.execute(sql2, sql_params, function (err, rows, result) {
                if (err) {
                    console.log("ERREUR SUR REQUETE : " + err.code +
                            " (" + err.message + ")");
                    return;
```

```
                }

                // construction des lignes détail du tableau HTML
                for (var i in rows) {
                    tmp_liste += `<tr>
    <td width="15%">${rows[i].codinter}</td>
    <td width="15%">${rows[i].codfra}</td>
    <td width="70%">${rows[i].countryname}</td>
  </tr>\n`;
                }

                // ajout du pied de tableau HTML
                tmp_liste += '</tbody>\n</table>\n';

                // Objet définiisant les parammètres de pagination
                var datapage = {
                    prelink: current_prelink + '/liste', current: current_page,
                    rowsPerPage: max_rows_per_page,
                    totalResult: nb_lig_total, slashSeparator: true
                };

                // ajout de la barre de pagination
                tmp_liste += pagination_tpl(datapage);

                // rendu final du tableau HTML
                res.end(template_page(
                    'Formulaire',
                    tmp_liste
                ));

                // conn.end();  // TODO : à revoir ultérieurement (pas urgent)

            });
        }
    });
};

// Préparation des routes, avec et sans critères de sélection, avec et
//  sans pagination, soit 4 possibilités
app.get("/liste", countriesList);

app.get("/liste/page/:page", countriesList);

app.get("/col_rech/:col_rech/typ_rech/:typ_rech/lib_rech/:lib_rech/liste",
countriesList);

app.get("/col_rech/:col_rech/typ_rech/:typ_rech/lib_rech/:lib_rech/liste/page/:page",
countriesList);

// Route par défaut pour erreur 404
app.use(function(req, res, next){
    res.status(404);
    res.end(template_page(
            'Erreur 404',
            'Désolé, page non trouvée'
    ));
});
app.listen(8080);
```

6 Changelog

Version 1.0 publiée le 12/01/2018 :
- première version forcément très incomplète :(

Version 1.1 publiée le 19/01/2018 :
- ajout de quelques précisions dans le chapitre de présentation de NPM (chapitre 2.1)
- chapitre 2.2 : titre « ligne de commande » renommé en « REPL » + ajout de précisions sur les raccourcis claviers et commandes usuelles en fin de chapitre
- ajout de nombreux compléments dans le chapitre sur Express (chapitre 3.5)

Version 1.2 publiée le 21/01/2018 :
- compléments sur les littéraux de modèle ajoutés à la fin du chapitre 2.6
- Rédaction du chapitre sur Express (chapitre 3.5)
- Rédaction du chapitre « mini-projet » (chapitre 3.6)
- Rédaction du chapitre sur la Dataviz avec D3.js et P5.js (chapitre 3.7)
- Rédaction d'un chapitre dédié au Creative Coding (chapitre 3.8)

Travail en cours et reste à faire :
- WebSocket, Mongo, PostgreSQL, Yeoman, Bower, Gulp, Grunt, Mocha, HBS, etc.