formation-zenika-nodejs

Travaux Pratiques

zenika
ARCHITECTURE INFORMATIQUE

Pré-requis

Node.js et NPM

Windows

Si une connexion internet est disponible, se rendre sur http://nodejs.org et télécharger la dernière version de Node.js. NPM est embarqué. Après l'installation, lancer une invite de commande et exécuter `node --version` pour vérifier que la commande `node` est disponible. Faire de même pour `npm`. En cas de problème, vérifier que le dossier d'installation de Node.js se trouve bien dans le `PATH`, et l'ajouter si nécessaire.

Sans connexion internet, utiliser la version fournie par le formateur dans le dossier `Node`. Décompresser l'archive `node-windows.zip` et ajouter le dossier résultant au `PATH`. Pour plus de faciliter, renommer `node-64.exe` (ou `node-32.exe` sur un système 32 bits) en `node.exe`.

Linux

Si une connexion internet est disponible, se rendre sur http://nodejs.org et télécharger la dernière version de Node.js. NPM est embarqué. Décompresser, l'archive et ajouter le dossier `bin` au `PATH`.

Sur un système Ubuntu ou Mint, il est possible d'utiliser APT, mais les versions sont un peu anciennes. Il faut passer par l'ajout d'un `ppa`. La procédure est documentée via le site en passant par `Download`, `Installing from package managers`, `Ubuntu Mint...`, ou suivre le lien suivant (https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager#ubuntu-mint-elementary-os).

Sans connexion internet, utiliser la version fourni par le formateur dans le dossier `Node`. Décompresser l'archive `node-v0.10.22-linux-x64.zip` et ajouter le dossier `bin` au `PATH`.

MongoDB

Un des TPs illustre la connexion à une base de données. La base de données en question est MongoDB. Installer une instance sur son poste est donc nécessaire.

Windows

Se rendre sur http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/ et suivre les instructions. En résumé, il s'agit de télécharger l'archive dans la bonne version, décompresser l'archive, préparer un répertoire pour les données et lancer Mongo.

Linux

Si vous utilisez une distribution compatible RedHat ou Ubuntu, des packages sont disponibles. Pour RedHat, CentOS, Fedora : http://docs.mongodb.org/manual/tutorial/install-mongodb-on-red-hat-centos-or-fedora-linux/ Pour Unbuntu, Mint : http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

Fil conducteur : Application de gestion de contact

Le sujet du TP, tout au long de la formation, sera de créer une application de gestion de contacts type carnet d'adresses.

Node.js étant une technologie côté serveur, développer une interface Web ne fait pas partie du périmètre de la formation. L'interface Web de l'application est par conséquent fournie dans les fichiers du TP.

Notre application va évoluer au fur et à mesure de la formation, du script sans interaction jusqu'au serveur REST et WebSocket appuyé sur une base de données, en passant par une interface en ligne de commande.

TP 1 : Bonnes pratiques JavaScript

JavaScript n'est pas seulement orienté objet. C'est un langage beaucoup plus flexible que peuvent l'être des langages comme Java. Aborder une application JavaScript en faisant une conception objet n'est pas forcément une bonne approche.

Néanmoins, reproduire les concepts de programmation objet est une très bonne mise en pratique des principes de prototypage ou de closure, ainsi cela sera tout de même le thème du premier TP.

Création des classes

Dans un nouveau fichier `.js`, créer les prototypes d'objet Contact pour contenir le nom et le prénom d'un contact et Contact**s** pour gérer la liste des contacts connue.

- Créer un fichier `contacts.js`
- Créer un constructeur pour l'objet `Contact` qui prend id, prénom (firstName), nom (lastName), adresse (address) et numéro de téléphone (phone) en arguments et qui les enregistre sous forme de champs de l'objet.
- Créer un constructeur pour l'objet `MemoryContacts` sans argument mais qui initialise une liste de contacts.

- Créer un constructeur pour l'objet `Contacts` qui ne fait rien pour le moment.
- Associer au prototype de l'objet `Contacts` une instance de `MemoryContacts`.
- Ajouter une méthode `toString` dans le `prototype` de la classe `Contact` retournant le nom en majuscule et le prénom du contact.

Lecture des contacts

Sans entrer dans le détail du fonctionnement de la fonction `require` fournie par `Node.js`, nous allons nous en servir pour charger la liste des contacts présente dans le fichier fourni `contacts.json`.

- En haut du fichier, ajouter la commande `let contacts = require('./contacts.json');`
- Toujours en haut de ce même fichier, charger `Lodash` avec la commande `let _ = require('./lib/lodash');`
- Dans le constructeur de l'objet `MemoryContacts`, itérer sur les `contacts` et instancier un objet `Contact` pour chaque élément. S'appuyer sur `Lodash` et principalement des fonctions `_.each` et `_.bind`.
- Ajouter une méthode `get` dans l'objet `MemoryContacts`. Cette méthode doit prendre un callback en paramètre et lancer ce callback avec les contacts en mémoire.
- Ajouter une méthode `print` à la classe `Contacts` qui affiche tous les contacts enregistrés.
- Instancier un objet `Contacts` et utiliser sa méthode `print`.

Aller plus loin

Remplacer l'utilisation d'`Lodash` par des fonctionnalités ECMAScript 5.

- Dans le constructeur de l'objet `MemoryContacts`, utiliser `Array.prototype.forEach` plutôt que `_.each`.
- Toujours dans cette même fonction, utiliser `Function.prototype.bind` à la place de `_.bind`.

TP 2 : Premiers contacts avec Node.js

Le deuxième TP a pour but de commencer à manipuler des librairies Node.js. Pour cela, nous allons enrichir notre code avec l'ajout de logs en couleurs. Ensuite, nous utiliserons la librairie commander.js (https://github.com/visionmedia/commander.js/) qui permet de

rapidement transformer le script en une commande "outillée" (options, documentations...).

Transformer le projet en module Node.js

Cette première étape a pour but de créer un fichier `package.json` pour décrire le projet et ses dépendances. Ensuite, il sera possible d'installer des dépendances avec NPM pour utiliser des modules Node.js.

- Utiliser la commande `npm init` pour initialiser automatiquement le fichier `package.json`. Répondre aux questions naturellement.
- Nous allons changer l'utilisation de lodash. Il est actuellement chargé directement depuis un fichier, nous allons maintenant l'installer sous forme de dépendance Node.js. Pour cela, utiliser la commande `npm install --save lodash`
- Supprimer l'ancienne inclusion de lodash au profit de `var _ = require('lodash');`
- On pourra constater que le fichier `package.json` a été renseigné en conséquence.

Ajouter de la couleur dans les logs

Maintenant que l'on dispose de la gestion des dépendances avec NPM, utilisons une des dépendances les plus courantes pour faire une évolution très simple : mettre un peu de couleur dans les logs.

- Installer le module `colors`, toujours avec l'option `--save`
- Charger cette dépendance avec `require('colors');`. Cas particulier, cette librairie étend le prototype de l'objet `String`. Il n'est pas utile d'affecter la valeur de retour du `require` à une variable.
- À partir de la documentation de colors (https://github.com/Marak/colors.js), afficher les noms de famille en bleu et les prénoms en rouge.

Utilisation de commander.js

Après l'utilisation d'un module très simple, essayons un module plus complet. commander.js, à l'instar de commander pour Ruby ou encore JCommander pour Java, permet de manipuler facilement le script par des paramètres passés à la ligne de commande.

- Installer le module `commander` et le charger.
- À partir de la documentation de commander.js ([commander.js] (https://github.com/visionmedia/commander.js/)), utiliser commander pour parser

les paramètres en ligne de commande.

- Dès ce moment, `node <votre fichier .js> --help` devrait fonctionner.
- Ajouter une *command* qui propose de lister les contacts disponibles. Brancher l'affichage des contacts lors de l'appel à cette command.
- Ajouter une *option* colors qui active ou non l'utilisation des couleurs. (Le plus simple sera d'utiliser une closure pour accéder à la valeur du paramètre, directement au niveau du toString de Contact)

TP3 : Architecture de Node.js

Afin de mettre en œuvre la programmation asynchrone et de manipuler l'API de Node.js, nous allons manipuler le module File System (http://nodejs.org/api/fs.html). Nous allons remplacer `MemoryContacts` par un `FileContacts` qui manipulera le fichier `contacts.json`.

Chargement du fichier de contacts

Nous allons donc revenir sur la lecture du fichier contacts.json pour le faire manuellement avec Node. Puis nous pourrons ajouter l'édition en réécrivant le fichier à chaque modification.

Attention, le contenu du fichier ne doit pas être conservé en mémoire.

- Supprimer la ligne `var contacts = require('./contacts.json');`
- Charger le module `fs`. Il est dans le cœur de Node, pas besoin de l'installer avec NPM.
- Créer un constructeur pour un objet `FileContacts`. A l'intérieur, définir la variable `this.path = './contacts.json'`.
- Toujours dans le constructeur, définir un méthode `this.read(callback)` qui utilise `fs.readFile(<path>, <callback>)` pour lire le fichier `contacts.json`.
- Dans le callback de `readFile`, utiliser `JSON.parse` pour convertir le texte en objet JS.
- Créer une liste de `Contact` et lancer le `callback` avec cette liste en argument.
- Implémenter la méthode `get` pour répondre à la même API que `MemoryContacts` en s'appuyant sur la méthode `read`.

Ajout d'opération de modifications

Maintenant que nous avons la main sur le fichier, il est possible d'ajouter des opérations

de modification dans la commande que nous développons. Nous allons ajouter les opérations d'ajout et de suppression de contact.

Attention, faites une sauvegarde du fichier contacts.json sans quoi, un mauvais code pourrait vous en faire perdre le contenu.

- Ajouter une méthode `this.write(contacts, callback)` qui utilise `fs.writeFile(<path>, <content>, <callback>)`. Pour passer des objets à une string JSON, utiliser `JSON.stringify`.
- Réaliser une méthode `add` qui prend en arguments un nom, un prénom et un callback. La méthode doit lire le fichier, ajouter un nouveau `Contact` puis réécrit le fichier. Il faudra définir automatiquement un nouvel id à l'objet.
- Réaliser une méthode `delete` qui prend un id et un callback en argument. Comme pour l'ajout, la suppression doit lire, modifier et écrire à la suite.
- Ajouter dans commander : les commandes `add` et `delete` et les options `--id, --lastName, --firstName`.
- En fonction de la commande demandée et parsée avec commander, lancer les méthodes de `Contacts`

Bonus : `fs.watch`

En bonus à ce TP, nous allons rajouter une commande qui permet de se mettre en écoute des changements sur le fichiers `contacts.json`. En effet, l'API de Node permet de faire cela très simplement. Il sera alors possible, avec deux terminaux, d'utiliser une fois le script pour surveiller le fichier et une seconde fois pour réaliser des modifications.

- Ajouter une méthode `watch` qui utilise `fs.watch(path, callback)` sur le fichier des contacts, le callback sera lancé à chaque modification sur le fichier.
- Lors d'une modification, nous n'avons pas d'information sur ce qui a été modifié. Pour répondre à cette problématique, relancer la lecture du fichier.
- Attention, il est possible que `watch` déclenche le callback dans des situations ou le fichier JSON n'est pas valide, gérer ce cas d'erreur.
- Une fois en possession des deux versions du fichier, un petit code est fourni pour permettre d'en connaitre les différences : `lib/diff.js`. Il s'utilise de cette façon :

```
var diff = require('./lib/diff.js');
diff(array1, array2); --> {additions: [...], deletions: [...]}
```

- Afficher ces données dans la console.
- Ajouter une commande `watch` dans commander qui pilote l'utilisation de notre

nouvelle méthode.

TP4 : Modularisation

A cette étape, on s'aperçoit bien que continuer à tout mettre dans un seul fichier trouve ses limites. L'objectif de ce TP est de compléter le fichier `package.json` afin de définir proprement notre projet. Ensuite, nous diviserons le code de notre application en plusieurs fichiers.

Renseigner le `package.json`

Si vous devez un jour publier le code de votre projet, on s'attendra à trouver un fichier `package.json` contenant de nombreuses informations. On souhaitera également ne pas avoir de warning à l'utilisation de la commande `npm`.

- Compléter les éléments de description du projet : `name`, `version`, `description`, `keywords`. Attention, ce dernier est un tableau.
- Ajouter un fichier `README.md` au projet.
- Ajouter les URLs `homepage` et `repository`.
- Ajouter un `author` et des `contributors`.
- Ajouter les paramètres plus techniques : `main`, `scripts` (on définira le script test qui lance mocha qu'on étudiera plus tard), `engines`.

Diviser les fichiers

Si pour du JavaScript côté navigateur ou dans certains autres langages, diviser le code dans plusieurs fichiers n'a pas d'impact fonctionnel (sous réserve d'être lu dans le bon ordre), ce n'est pas le cas avec Node. Ainsi, réaliser la séparation en fichiers demande de penser la modularisation de l'application.

On propose comme organisation pour le TP de conserver uniquement le parsing des arguments dans le fichier d'entrée de notre application. Le cœur de l'application qui se trouve maintenant dans la "classe" `Contacts` aura son propre fichier `Contacts.js`. Enfin, le code qui prend en compte les paramètres passés et lance les méthodes correspondantes sera localisé dans un fichier `router.js` (Router au sens routage des paramètres de ligne de commande).

- Créer un fichier `index.js` qui sera le point d'entrée de l'application.
- Diviser les objets dans un fichier chacun : `contact.js`, `contacts.js`, `contact-file.js`, `contact-memory.js`.
- Chacun de ces fichier exportera son constructeur et dépendra des autres au

besoin.

- Créer un fichier `router.js`. Il exporte une fonction qui prend en paramètre une instance de `Contacts`.
- Y mettre le code de paramétrage de commander.
- Dans le fichier `index.js`, effectuer un require sur `./contacts` et `./router`
- Instancier un objet `Contacts` et lancer le router

TP 5 : Création d'un serveur Web

Dans ce nouveau TP, nous allons toujours modifier notre commande pour qu'avec une simple nouvelle commande, cela déclenche un serveur HTTP permettant de manipuler les contacts en REST.

De plus une application Web réalisée avec Angular.js est fournie pour rendre l'utilisation de cette API REST visuelle.

Lancer un serveur Express

Cette fois-ci, comme nous avons déjà modularisé notre application, nous allons créer tout de suite un nouveau module pour la gestion du serveur Web.

- Ajouter une command `serve` dans commander.
- Créer un nouveau module `web.js`. Il publie une fonction qui permet de l'initialiser en passant une instance de `Contacts` en argument.
- Télécharger et ajouter (`--save`) les dépendances : `express`, `body-parser` et `serve-static`. Les charger dans `web.js`.
- Dans la fonction d'initialisation du serveur web, démarrer une instance d'express, y ajouter les middleware body parser et serve static. Pointer les fichiers static de l'application web qui a été fournie.
- Démarrer le serveur sur un port de votre choix

A cette étape, il devrait être possible de voir le site fonctionner. Cependant, comme l'API REST n'est pas encore présente, le site ne présente pour le moment aucun contact.

Ajouter les routes REST

- Créer une fonction `router` qui prend en paramètre l'application express ainsi que l'instance de Contacts.
- Dans cette fonction, déclarer la route `/rest/contacts` sur la méthode `GET` qui

renvoie la liste de tous les contacts.

- Déclarer une route `/rest/contacts/:id` sur la méthode `GET` qui renvoie le contact avec l'identifiant demandé.
- Déclarer une route `/rest/contacts` sur la méthode `POST` qui crée un nouveau contact.
- Déclarer une route `/rest/contacts/:id` sur la méthode `PUT` qui modifie un contact. Passer par une suppression suivie d'une création.

Tester la navigation de l'application web.

Bonus : Implémentation générique du getById

La route get sur un seul contact nécessite de tout charger puis de sélectionner le bon élément dans la liste. Il serait plus "propre" de réaliser cette opération dans `contacts.js` mais cela demande de s'appuyer sur la souplesse du JavaScript.

L'objectif est de définir la méthode get qui si un `id` est passé en premier argument, retournera seulement le contact correspondant, sinon, il faut faire appel à la fonction get du prototype.

- Dans `contacts.js`, ajouter la méthode `get` dans l'objet `Contacts`. Ne pas définir d'arguments en entrée, il faudra les lire dynamiquement.
- Réaliser un aiguillage en fonction du type du premier argument (`arguments[0]`). S'il s'agit d'une fonction, il s'agit d'un get all sinon un get by id.
- Dans le cas du get all, appeler la fonction `get` du prototype en passant par `Contacts.prototype.get`. Pour transmettre les mêmes paramètres, on peut utiliser `Function.prototype.apply(context, arguments)`.
- Dans le cas du get by id, faire un get all, sélectionner le bon contact et appeler le callback avec ce contact.

TP 6 : Gestion de l'asynchronisme

Dans ce TP, nous allons reprendre la méthode `write` de l'objet `FileContacts` et lui ajouter de la complexité qui mettra en avant les questions d'asynchronisme.

Ensuite, il s'agira de réaliser l'implémentation de cette méthode avec le module `async` et enfin avec les promesses.

`write` avec les callbacks

- Créer un fichier `write-implems.js` dans lequel nous mettrons les différentes

implémentation de la fonction `write`.

- Dans `contacts-files.js`, supprimer l'implémentation de la méthode `write` et brancher une des implémentations à venir dans `write-implems.js`.
- De retour dans `write-implems.js`, exporter une première implémentation `exports.callbacks = function(contacts, callback) {...}`
- Cette nouvelle implémentation :
  - Lit le fichier `contacts.json`
  - Réalise un backup dans `contacts.json.back`
    - Réécrit le fichier `contacts.json` avec les contacts en paramètre
  - En cas d'erreur à cette étape, renomme le fichier `contacts.json.back` en `contacts.json`
- Cette méthode ne doit utiliser que des callbacks.

On constatera l'empilement de callbacks et l'indentation qui s'additionne. C'est ce qu'on appelle le callback hell ou pyramid of doom.

`write` avec les promesses

- Installer et charger le module `denodeify`.
- Exporter une troisième fonction `exports.promise = ...`.
- Utiliser les promesses et l'appel aux fonctions Node.js via `denodeify`.
- Changer le pointeur de la fonction `write` dans `Contacts`.

L'enchaînement des traitements asynchrones devrait maintenant être plus lisible.

`write` avec les générateurs

- Installer et charger le module `co`.
- Exporter une quatrième fonction `exports.generators = ...`.
- Utiliser un générateur, `co` et l'appel aux fonctions Node.js via `denodeify`.
- L'appel à `co` retourne une promesse. Utiliser `then` et `catch` pour appeler le callback passé en paramètre de `write`.
- Changer le pointeur de la fonction `write` dans `Contacts`.

TP7 : Mise à jour temps réel des contacts

Nous allons mettre en place Socket.io sur le serveur afin de pouvoir prévenir le client (le navigateur du client) que la liste des contacts a changé.

Nous pourrons nous appuyer sur la fonction `watch` déjà prévu dans FileContacts.

Ajouter Socket.io au serveur Express

Nous allons simplement commencer par brancher Socket.io au serveur.

- Installer le module `socket.io` avec `npm`.
- Faire le `require` puis créer une instance de socket.io qui écoute sur le serveur Express (`var io = socketio.listen(server);`)
- Passer l'instance de Socket.io à la fonction `router`.
- Dans le routeur, écouter l'évènement de connexion d'une socket et logger quelque chose.

L'application frontend a déjà Socket.io de configuré. La connexion devrait s'établir automatiquement. Comme Socket.io est configuré avec des logs assez détaillés par défaut, il sera possible de constater la connexion dans les logs.

`watch`

La fonction `watch` prévue en bonus du TP4 sera maintenant nécessaire. Si vous n'aviez pas eu le temps de la faire à ce moment là, il faut reprendre cet exercice.

Utiliser la surveillance sur un fichier est un bon moyen de créer des évènements côté serveur qu'il est intéressant de pousser côté client.

Surveiller les modifications

L'application frontend est programmée pour pouvoir mettre à jour la liste de contacts si elle reçoit un message Socket.io nommé `contacts` contenant une nouvelle liste de contacts.

- Modifier la fonction watch pour prendre un callback en paramètre. Elle doit lancer ce callback lorsqu'une modification a lieu.
- Lancer le `watch` dans le router, lorsqu'une modification a lieu, transmettre la nouvelle liste de contacts par un message Socket.io
- Le serveur ne doit utiliser le `watch` qu'une seule fois et prévenir tous les clients lors d'une modification.

Pour tester, démarrer le serveur web avec la commande `web`, puis, dans une autre console, utiliser la commande `add` et vérifier que le nouveau contact apparait dans la

page web, sans rafraichissement.

TP8 : Lecture des contacts par stream

Dans `contacts-file.js`, la lecture du fichier `contacts.json` est faite en une fois puis parsé en JSON.

Sur le modèle du TP7 sur l'asynchronisme, nous allons créer une nouvelle implémentation de la lecture en utilisant un stream.

Il existe une librairie capable de parser du JSON progressivement sur un stream. Nous allons nous en servir pour traiter les contacts un par un.

- Créer un fichier `read-implems.js` et y déplacer l'implémentation actuelle de la fonction `read` exporté dans la propriété `exports.original`.
- Créer une nouvelle implémentation dans `exports.stream`. Cette nouvelle implémentation se base sur le point d'entrée `fs.createReadStream(path)`.
- Installer les dépendances `through2` et `JSONStream`.
- Pour sélectionner les contacts dans le stream avec `JSONStream`, il faut utiliser le pipe suivant : `.pipe(JSONStream.parse('*'))`
- Attention, cette transformation propose en sortie un stream au format objet.
- Utiliser la librairie `through2` pour insérer un stream qui permettra de lire chaque contact.
- Pour chaque contact, créer un objet Contact et l'ajouter à un tableau.
- Écouter l'évènement `finish` du stream pour lancer le callback.
- Remplacer dans `contacts-file.js` l'implémentation originale par celle utilisant les streams, le fonctionnel devrait être le même.

TP9 : Persistance avec MongoDB et Mongoose

Avant d'aller plus loin, il est nécessaire pour ce TP d'installer un serveur MongoDB. La procédure d'installation est décrite dans les prérequis au début de ce document.

Une fois MongoDB installé, il est possible de charger la même base de contacts que celle existant dans le fichier en utilisant cette commande.

```
mongoimport --jsonArray -d test -c contacts contacts.json
```

Modélisation d'un Contact

Pour des raisons d'enchaînement des TPs, nous allons continuer d'utiliser les ids numériques existant actuellement. Cependant, pour un projet pensé avec MongoDB, le plus logique serait de passer sur l'identifiant géré automatiquement par Mongo en interne.

- Créer un fichier `contacts-mongo.js` qui aura pour but de se substituer au `contacts-file.js` utilisé actuellement.
- Charger le module `mongoose`
- Définir le schéma de Contact avec trois propriétés : id numérique et lastName et firstName des strings.
- Créer le modèle Mongoose à partir du schéma.

Création de MongoContacts

- Dans le constructeur de l'objet `MongoContacts` se connecter au serveur Mongo avec la ligne `mongoose.connect('mongodb://localhost/test');`.
- Implémenter la fonction `get` à partir de `ContactModel.find`
- Implémenter la fonction `add`. Il vous sera nécessaire de trouver le plus grand id en base, vous pourrez le faire de cette façon :

```
ContactModel.findOne().sort('-id').exec(function(err, data){

});
```

Utiliser ensuite `new ContactModel` puis `contact.save`.

- Implémenter la fonction `delete` à partir de `ContactModel.remove`.
- Le serveur web s'attend à trouver une méthode `watch`. Nous n'allons pas la réaliser, réaliser un bouchon permettant simplement au serveur de fonctionner.

Fin du processus

On constatera que la ligne de commande ne termine plus. En effet, la connexion à MongoDB reste ouverte. Pour que la commande termine correctement, il faudrait clôturer la connexion à Mongo lorsqu'elle n'est plus utile.

La nature asynchrone de Node.js fait qu'il ne suffit pas d'appeler la clôture de la connexion juste après l'appel à la fonctionnalité souhaité. La connexion serait coupé avant d'avoir eu le temps de réaliser l'opération.

Il faut donc que chaque opération propose un callback indiquant quand elle a terminée. C'est déjà le cas pour `add` et `delete`, mais pas encore pour `print`.

- Ajouter un callback dans la fonction `print` pour qu'il soit lancé quand l'opération est terminée.
- Dans `contacts-mongo.js` ajouter une méthode `close` qui lance la fonction `mongoose.disconnect()`.
- Dans `router.js`, créer une fonction qui, si l'instance de `Contacts` possède une fonction `close` la lance et ne fait rien sinon.
- Mettre cette fonction `close` en callback des fonctions `print`, `add`, `delete`.

Bonus : getById

Nous avons implémenté dans les TPs précédent, une fonction `get` générique qui travail sur l'ensemble des données. L'idée ici serait d'ajouter un cas particulier pour mongo afin que la sélection soit faite dans le moteur plutôt que dans Node.js.

- Ajouter une fonction `getById` dans MongoContacts. Elle prend un id en argument et rend le `Contact` correspondant.
- Modifier dans `contacts.js` la fonction `get`.
- Tester l'existence d'une fonction `getById` dans le prototype de `Contacts`.
- Si la fonction existe l'utiliser, sinon conserver le mécanisme actuel.

TP10 : Debug et Test

Débogage

Selon que vous soyez plus **ligne de commande** ou **interface graphique**.

Ligne de commande

- Ajouter un ou plusieurs breakpoints dans votre code avec la commande `debugger;`
- Lancer le script avec la commande `node debug`.
- Utiliser les contrôles pour gérer l'exécution du programme et d'étudier la valeur des variables.

Node Inspector

- Installer Node Inspector avec `npm install -g node-inspector`
- Lancer le script avec la commande `node-debug`.
- Une fenêtre de navigateur va s'ouvrir. Le programme sera automatiquement bloqué sur sa première ligne afin de vous permettre de positionner des breakpoints.
- Utiliser l'outil pour gérer l'exécution du programme et d'étudier la valeur des variables.

Test

Nous allons maintenant rajouter un test à notre application. Tout d'abord, il faut installer les librairies pour les tests.

```
npm install -g mocha
npm install --save-dev mocha chai sinon
```

Le test va consister à charger `Contacts` avec la première implémentation (`MemoryContacts`). On vérifiera que le `toString` d'un contact retourne bien le `lastName` en majuscule.

- Créer un répertoire `test` et un fichier `test-contacts.js`.
- Charger les modules `assert`, `contacts-memory`, `contacts` et `contact`.
- Créer la structure principal du test avec `describe` et `it`.
- Dans un `beforeEach`, surcharger le prototype de `Contacts` avec `MemoryContacts` et instancier un objet contacts.
- Dans le `it`, récupérer le premier contact, utiliser sa fonction `toString`.
- Tester la validité du résultat. Par exemple avec la RegExp `/[A-Z]* [A-Za-z]*/.test(string);`.

Test avec Chai et Sinon

Nous allons rajouter les fonctionnalités des librairies Chai et Sinon à notre test.

Chai va simplement nous permettre de faire l'assertion de façon plus élégante.

Sinon va nous permettre de mettre un espion sur une fonction et de vérifier qu'elle a été appelée.

- Charger les modules `chai` et `sinon`.
- Lancer la fonction `chai.should()` pour activer la possibilité d'utiliser l'API should de

chai.

- Remplacer l'assertion avec assert par un should `xxx.should.be.xxx`.
- Créer un `spy` sinon sur la fonction `toString` de l'objet `Contact`.
- A la fin du test, vérifier que la fonction a été appelée une seule fois.

TP11 : Cluster

L'objectif de ce dernier TP est de lancer notre serveur Web en mode cluster. Nous allons ajouter une option à la ligne de commande permettant de spécifier le nombre de workers.

- Dans `router.js`, ajouter une option `--cluster`, attention, `-c` est déjà utilisé pour colors.
- Passer la valeur au lancement du serveur web.
- Dans `web.js`, déplacer tout le code de la fonction exportée dans une nouvelle fonction privée. Il faudra lancer ce code pour les workers mais pas pour le master.
- Avant tout, nous allons maintenir le fonctionnement actuel en testant l'existence du second paramètre. Tester si une variable est **parsable en nombre** n'est pas forcément évident. Une solution est `isNaN(parseInt(value))`.
- Si le deuxième paramètre n'est pas un nombre, lancer simplement la fonction précédemment créée.
- Sinon nous allons démarrer le cluster.
- Charger le module noyau `cluster`.
- Dans le master (`cluster.isMaster`) lancer autant de `fork` que demandé en paramètre. logger également le passage dans le master.
- Dans le worker, lancer la fonction qui démarre le serveur. Logger le passage dans le worker avec son id (`worker.id`).

Lancer le serveur. Les logs indiqueront le nombre de workers et le démarrage des workers. On peut également constater au niveau système qu'il y a des processus supplémentaires qui ont été lancés.