accesDA une preuve de concept!

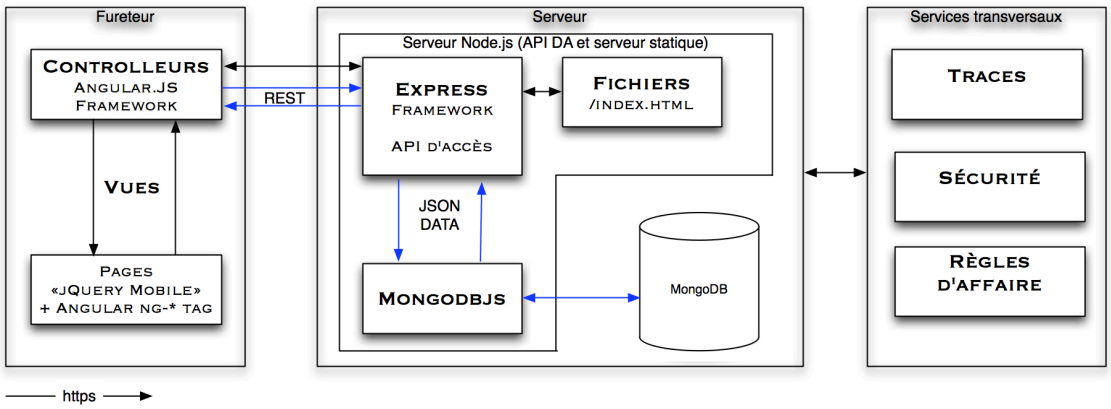
Présentation

Dans le cadre de la recherche d'une architecture cible pour le prochain dossier académique, nous avons réalisé une preuve de concept d'une application WEB monopage qui nous permettra d'accéder au dossier étudiant via une interface web conviviale et orientée mobile. L'expérience de l'utilisateur est au centre de ce type d'application qui réagit telle une application de bureau.

Nous appellerons cette preuve accesDA. Ce premier article dans une série de trois décrit la partie serveur de l'application. Un second article traitera de la partie présentation et un troisième traitera des services transversaux tels que l'authentification, le filtrage des données en fonction de la sécurité et le service d'audit.

De prime abord, deux constats s'imposent. Le développement du prochain dossier académique demandera quelques années, à terme, nous devons nous assurer que les technologies utilisées ne seront pas désuètes et ne nous obligerons pas à planifier de nouveau un prochain dossier académique. De plus, nous avons maintenant une obligation légale, la loi C133. Cette preuve de concept tente donc de tenir compte de ces deux éléments.

Le schéma suivant résume sommairement l'architecture d'accesDA, une architecture **simple et efficace**.



Cette preuve de concept existait déjà dans l'environnement technologique « *jQuery mobie, PHP, SQL* » mais ici on utilise un ensemble technologique plus moderne et mature que nous avons nommé « jMEAN » (*jQuery mobile, Mongodb, Express.js, Angular.js* et *Node.js*)*. Dans cet article, nous nous proposons de décrire la partie serveur d'accesDA, soit la partie utilisant dans *Node.js*. Pour le moment, nous prenons pour aquis que MongoDB est installé et fonctionnel.

Node.js est une plateforme de développement d'applications Web à haute scalabilité qui utilise JavaScript (l'engin V8 de Google ayant une licence libre de type MIT).

JavaScript est utilisé comme langage de programmation de bout en bout. Une architecture de type *REST* est utilisée pour accéder aux ressources du DA via le protocole HTTPS.

Ici, par souci de simplification dans la preuve de concept, le serveur *Node.js* sert à la fois les pages statiques et l'accès aux API, ce qui pourrait être effectué sur 2 serveurs différents. Évidemment, pour la même raison, le dépôt de données (MongoDB) est sur le même serveur alors qu'il devrait être sur un autre.

Pourquoi JavaScript?

Question dont la réponse déborderait de beaucoup cette simple présentation. Mais pour résumer l'importance de JavaScript, voici quelques faits :

- Le fureteur est l'application la plus utilisée, toute catégorie confondue;
- JavaScript (JS) est l'environnement d'exécution le plus utilisé;
- JS ne demande aucun effort de déploiement, pas d'application à installer pas de *Store* à accéder, simple et disponible;
- Les dernières de JS sont très rapides;
- Par nature cet environnement d'exécution est indépendant du système d'exploitation;
- JS est un excellent environnement pour développer des applications mobiles multiplateformes;
- Une multitupe d'outils vient enrichir l'environnement JS, *Node.js* Angular.js, Ember.js, Grunt, Yeoman, etc.;
- Depuis 2011, 1 milliard d'appareils mobiles (Android et iOS) avec l'environnement d'exécution JS se sont ajoutés.

Vous croyez toujours que JavaScript est un jouet ?

Préparation du projet.

Afin de nous aider à bâtir rapidement notre application du côté serveur, nous utiliserons le framework *Express.js*. Ce framework est une extension

d'un autre framework qui fait partie de *Node.js*, soit *Connect*. Ce dernier utilise des *intergiciels* pour créer la logique d'une application tout en facilitant la réutilisation de ces composantes.

Dans *Connect*, une composante signifie une fonction qui intercepte les objets réponse et demande du serveur HTTP, exécute sa logique et transmet l'exécution à la composante suivante ou la termine. *Connect* relie ces composantes à l'aide d'un répartiteur. Le framework *Express.js* va bien au-delà de cette présentation, nous vous invitons donc à jeter un coup d'oeil sur les sites d'*Express.js* et de *Node.js* pour plus d'information.

Installation d'*Express.js*

Lors de l'installation de *Node.js*, un outil d'installation de modules supplémentaires *npm* est également installé. Cet outil permet d'installer tous les modules *Node.js* qui sont nécessaires à la réalisation d'une application, nous utiliserons donc cet outil pour installer *Express.js*. Généralement, les modules de *Node.js* composant une application sont installés localement au projet (dans le répertoire du projet), ainsi on rend le projet indépendant du point de vue des modules tout en évitant de polluer l'installation globale de *Node.js*. Mais comme *Express.js* fournit un outil qui peut être utilisé par l'ensemble de nos projets, nous l'installerons d'une façon globale.

Installation d'Express.js

```
$ sudo npm -g install express
...
npm http 200 https://registry.npmjs.org/readable-stream/-/readable-stream-1.0.17.tgz
/usr/local/bin/express -> /usr/local/lib/node_modules/express/bin/express
express@3.4.0 /usr/local/lib/node_modules/express
 methods@0.0.1
 cookie-signature@1.0.1
 range-parser@0.0.4
 fresh@0.2.0
 buffer-crc32@0.2.1
 cookie@0.1.0
 debug@0.7.2
 mkdirp@0.3.5
 send@0.1.4 (mime@1.2.11)
 commander@1.2.0 (keypress@0.1.0)
 connect@2.9.0 (uid2@0.0.2, pause@0.0.1, qs@0.6.5, bytes@0.2.0, multiparty@2.1.8)
$
```

Une fois *Express.js* installé, nous sommes en mesure de créer notre application. *Express.js* est un framework qui permet de bâtir des applications Web (MVC), mais comme notre preuve de concept est une application de type monopage, il ne sera pas nécessaire d'utiliser l'ensemble de l'application d'*Express.js*. Nous allons configurer un répertoire pour livrer les pages statiques ainsi qu'une composante agissant comme une artère vers les API qui accéderont aux données DA.

Création de l'application

L'application et ses ressources résident dans un répertoire créé par l'utilitaire *express*. En mode terminal, déplacez vous à l'endroit où vous désirez créer votre projet, puis entrez la commande *express* suivi du nom du projet. Cette commande va créer le répertoire du projet ainsi qu'une structure conventionnelle pour une application *Express.js*.

Création du projet

```
Air:~ simoneau$ cd Sources/git-vrsi
Air:git-vrsi simoneau$ express accesDA
   create : accesDA
   create : accesDA/package.json
   create : accesDA/app.js
   create : accesDA/public
   create : accesDA/public/javascripts
   create : accesDA/public/stylesheets
   create : accesDA/public/stylesheets/style.css
   create : accesDA/public/images
   create : accesDA/routes
   create : accesDA/routes/index.js
   create : accesDA/routes/user.js
   create : accesDA/views
   create : accesDA/views/layout.jade
   create : accesDA/views/index.jade
   install dependencies:
...
```

En créant l'application, la commande *express* a également créé un fichier qui va nous aider à gérer les dépendances du projet, soit *package.json*. À tout moment vous pouvez changer ce fichier pour ajouter des dépendances. Une fois le fichier modifié, vous n'avez qu'à entrer la commande *npm install* pour mettre à jour les dépendances du projet. Comme nous aurons besoin d'accéder à *MongoDB, déplacez-vous dans le répertoire de votre application accesDA et* modifiez le fichier *package.json* afin d'y ajouter cette dépendance.

Ajouter Mongo aux dépendances

```
{
  "name" : "application-name",
  "version" : "0.0.1",
  "private" : true,
  "scripts" : {
    "start" : "node app.js"
  },
  "dependencies" : {
    "express" : "3.4.0",
    "jade" : "*",
    "mongodb" : "*"
  }
}
```

Une fois cette modification effectuée, lancez la commande *npm install*. Vous pouvez voir que 3 dépendances ont été installées dans le nouveau répertoire node_modules (créé par la même occasion) soit *express, jade* et *mongoDB* puisqu'ils sont tous dans le fichier de configuration *packag e.json*.

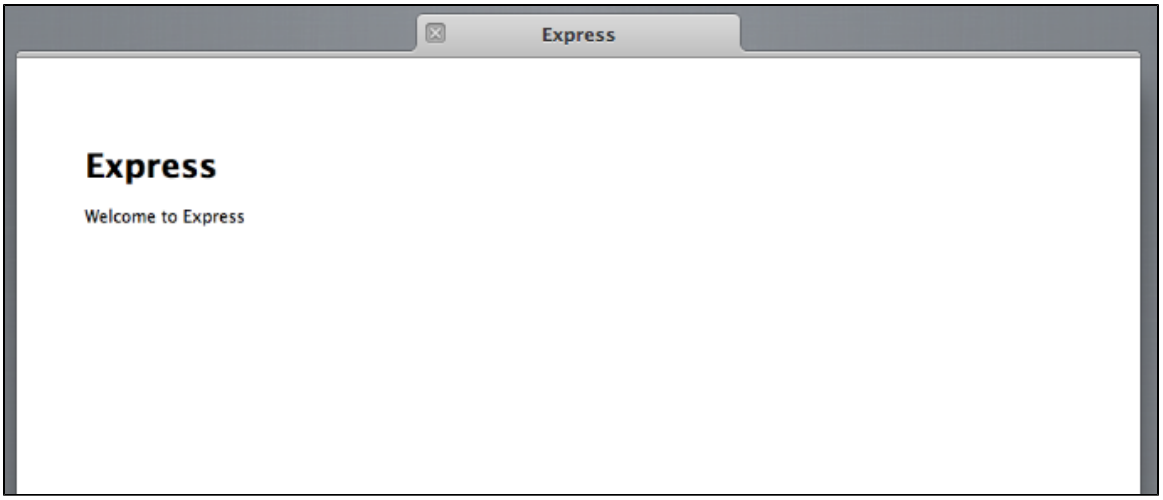
Mise à jour

```
$ cd accesDA/;npm install
...
npm http GET https://registry.npmjs.org/mongodb
npm http GET https://registry.npmjs.org/jade
npm http GET https://registry.npmjs.org/express/3.4.0
...
 methods@0.0.1
 range-parser@0.0.4
 cookie-signature@1.0.1
 fresh@0.2.0
 buffer-crc32@0.2.1
 cookie@0.1.0
 debug@0.7.2
 mkdirp@0.3.5
 commander@1.2.0 (keypress@0.1.0)
 send@0.1.4 (mime@1.2.11)
 connect@2.9.0 (uid2@0.0.2, pause@0.0.1, qs@0.6.5, bytes@0.2.0, multiparty@2.1.8)
jade@0.35.0 node_modules/jade
 character-parser@1.2.0
 commander@2.0.0
 mkdirp@0.3.5
 monocle@1.1.50 (readdirp@0.2.5)
 with@1.1.1 (uglify-js@2.4.0)
 constantinople@1.0.2 (uglify-js@2.4.0)
 transformers@2.1.0 (promise@2.0.0, css@1.0.8, uglify-js@2.2.5)
mongodb@1.3.19 node_modules/mongodb
 bson@0.2.2
 kerberos@0.0.3
$
```

Vous êtes maintenant prêt à démarrer l'application en entrant la commande *node app.js*. Utilisez un fureteur et accédez votre application à l'URL « http://localhost:3000/ ». Vous remarquerez que cette requête sera imprimée sur le terminal. Voilà la première application terminée.

Lancer l'application

```
Air:accesDA simoneau$ node app.js
Express server listening on port 3000
GET / 200 445ms - 170b
GET /stylesheets/style.css 200 10ms - 110b
```



Ajout de l'API

Modification du fichier de configuration des dépendances

Nous allons maintenant modifier cette application pour mettre en place nos API d'accès au DA. D'abord regardons la structure de répertoire générée par la commande *express* et sa signification:

```
app.js       : contient le code l'application.
package.json : contient le fichier de configuration de l'application.
public       : contient les éléments statiques (HTML, CSS, JS) de l'application.
routes       : contient le routage en fonction des URLs.
views        : dans une application MVC, contient les vues (gabarits).
```

Maintenant que nous savons que l'application réside dans le fichier « app.js » ouvrons le et examinons sommairement le code.

app.js

```
/**
 * Module dependencies.
 */

var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));


// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Express.js n'est pas le but de cette présentation mais voici quelques explications afin d'éclaircir l'architecture de l'application. Cette dernière est gérée à l'aide d'une instance de la classe « *express.HTTPServer »* retournée par l'appel de la méthode *express() à la ligne 11.* On associe la route '/' et la méthode GET à une fonction de rappel(*callback)* index (localisée dans fichier « ./roure/index.js ») à la ligne 30 qui va effectuer le rendu du gabarit « index.jade » (localisé dans « ./views/index.jade ») en lui passant un objet contenant le titre *Express*.

Retrait du code généré non nécessaire

Comme nous n'utiliserons pas une application conventionnelle de type MVC du côté serveur, nous pouvons retirer la partie des vues du code source. De plus, comme notre application ne validera pas l'utilisateur localement mais plutôt d'une façon transversale, nous pouvons également retirer le code relatif aux utilisateurs.

JSLint une bonne pratique

Puisque nous sommes à modifier le code source, profitons de l'occasion pour glisser un mot au sujet de *JSLint*. Cet outil d'analyse statique du code source JavaScript permet de renforcer votre code source en validant ce dernier auprès de règles de codage. Comme JavaScript est par nature permissif, il est fortement recommandé de suivre les suggestions de *JSLint* lorsque votre projet dépasse quelques lignes.

Voici la nouvelle version épurée du code et passée dans JSLint

app.js

```
var express = require('express');
var http    = require('http');
var routes  = require('./routes');
var path    = require('path');


var app = express();

// Dans tout les environnements.
app.set('port', process.env.PORT || 3000);
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express['static'](path.join(__dirname, 'public'))); // JSLint. Syntaxe
différence parce que static est un mot réservé.
app.use(app.router);


// L'environnement de développement.
if ('development' === app.get('env')) {
   app.use(express.errorHandler({dumpExceptions : true, showStack : true}));
}


// L'environnement de production.
if ('production' === app.get('env')) {
   app.use(express.errorHandler());
}


http.createServer(app).listen(app.get('port'), function () {
   console.log('Express est démarré sur le port ' + app.get('port'));
});
```

Si vous ré-exécutez cette application, et accédez à « http://localhost:3000/ » vous aurez le message suivant:

«

```
Cannot GET /
```

»

C'est normal puisque nous n'avons pas de remplacement pour la vue. Nous allons ajouter un fichier HTML qui contiendra éventuellement l'application monopage, mais pour le moment on peut simplement ajouter un fichier HTML incluant *jQuery* qui nous sera nécessaire lorsque nous ferons la partie présentation de l'application.

index.html

```
<!doctype html>
<html lang="fr">
<head>

  <title>html5</title>

  <meta charset="utf-8">
  <script src="http://code.jquery.com/jquery-1.8.0.min.js"></script>

  <script>
    /*jslint         browser : true, continue : true,
      devel  : true, indent  : 3,    maxerr   : 50,
      newcap : true, nomen   : true, plusplus : true,
      regexp : true, sloppy  : true, vars     : true,
      white  : true
    */

    /*global jQuery */
    jQuery(document).ready(function() {
      console.log('Bonjour Express!');
    });
  </script>

  <style></style>

</head>

<body>
  <p>Bonjour Express!</p>
</body>

</html>
```

Si vous accédez à nouveau, vous devriez voir « Bonjour Express ! ». Maintenant voici une explication sommaire d'une partie du code simplifié de l'application.

app.js

```
app.set('port', process.env.PORT || 3000);                 // 1
// Section middleware                                      // 2
app.use(express.favicon());                                // 3
app.use(express.logger('dev'));                            // 4
app.use(express.bodyParser());                             // 5
app.use(express.methodOverride());                         // 6
app.use(app.router);                                       // 7
app.use(express.static(path.join(__dirname, 'public')));   // 8

if ('development' == app.get('env')) {                     // 9
  app.use(express.errorHandler({dumpExceptions : true,     // 10
                                showStack : true
                               }));
}

http.createServer(app).listen(app.get('port'), function()  // 11
```

1. On définit le port sur lequel répondra le serveur *HTTP*.
2. À partir d'ici, chacun des appels à *app.use* représente un *intergiciel*. Voir la discussion précédente à ce sujet.
3. Cette composante s'occupe d'envoyer le fichier « /favicon.ico » pour la requête *HTTP*.
4. Toutes les requêtes seront imprimées sur le moniteur.
5. S'il s'agit d'un POST/PUT, les éléments sont automatiquement extraits du formulaire.
6. Une façon de contourner votre fureteur, si ce dernier ne peut envoyer un *PUT* ou un *DELETE*.
7. On utilise l'URL afin d'associer le service ou la fonction métier à la requête. Si pas placé avant static toutes les requêtea iront sur le disque.
8. On définit l'endroit où l'on place les fichiers statiques de l'application.
9. Dépendant de l'environnement on n'imprime pas de les mêmes informations.
10. On passe des options à la fonction des erreurs.
11. Démarrage du serveur http.

*Representational state transfer (**REST**)*

Deux mots au sujet de *REST*. Cette architecture s'appuie sur le protocole *HTTP*. La simplicité est au coeur de cette architecture, on accède aux ressources (noms, objets métier) avec un nombre très limité de verbes qui vous permettre d'effectuer les opérations « standards » de création, lecture, mise à jour et d'effacement (*Create, Read, Update, Delete* de là l'acronyme *CRUD*). On associe donc les méthodes d'accès du protocole *HTTP* au type d'opérations (*CRUD*) que l'on désire effectuer. Pour plus d'information vous pouvez consulter le site *RESTful Resources Required Reading*. Évidemment ici la création, mise à jour et la destruction d'un dossier étudiant ne sont que pour illustrer l'utilisation des méthodes *POST, PUT* et *DELETE*.

Méthode d'accès **HTTP / HTTPS**	**Opération**	**URL pour accéder aux ressources**	**Action**
POST	Création	POST /libreda.uqam.ca/da/etudiants	Création d'un étudiant
GET	Lecture	GET /libreda.uqam.ca/da/etudiants/ GET /libreda.uqam.ca/da/etudiants/:id GET /libreda.uqam.ca/da/etudiants?nom=	Liste des étudiants Un étudiant précis Rechercher par nom
PUT	Mise à jour	PUT /libreda.uqam.ca/da/etudiants	Une ou plusieurs mises à jour selon les éléments soumis

DELETE	Effacement	DELETE /libreda.uqam.ca/da/etudiants/:id	Effacement d'un étudiants précis

Ajout du code des API

Comme l'accès aux API est différent de l'accès Web en général, et qu'il requiert des méthodes spécifiques (GET, POST, PUT, DELETE), nous utiliserons un autre type de fonction d'Express.

« app.[méthode HTTP] » (plutôt que *app.use*) nous permet de **limiter** à une méthode spécifique l'association d'une route. Ainsi « app.get ('/libreda.uqam.ca/da/etudiants/', fonction) » nous permettra d'obtenir la liste des étudiants en répondant uniquement à la méthode *GET*. C'est comme lors l'utilisation de « app.use » à l'exception qu'on l'utilise une méthode spécifique. De cette façon, nous n'avons pas à analyser nous même le type méthode d'accès pour chaque requête.

Évidemment ici nous utilisons l'API *REST* pour ajouter, pour mettre à jour ou pour effacer à des fins de démonstration. Dans la preuve de concept complète, nous utiliserons des services transversaux afin d'effectuer ce type d'opérations complexes qui font appel à des services transversaux, par exemple le service de règles d'affaires qui valide l'ajout d'un nouvel étudiant.

Nous allons maintenant créer un module pour les API d'accès aux étudiants. Cet API ne contenant que les fonctions CRUD, il est facile à remplacer si en cours de route nous décidons d'utiliser une autre base de données.

Pour créer le module, ajoutez un nouveau fichier « etudiant.js » que vous devez placer sous le répertoire « routes ». Dans ce fichier, ajoutez le code source suivant.

API

```
var ObjectId = require('mongodb').ObjectID;

/* APIs */
exports.chercherParId = function (db) {
   return function (req, res) {
      console.log(req.params);
      var id = req.params.id.toString();
      console.log('chercherParId: ' + id);
      db.collection('dossiers', function (err, collection) {
         collection.findOne({ '_id': new ObjectId(id) }, function (err, item) {
            console.log(item);
            res.json(item);
         });
      });
   };
};

exports.chercher = function (db, limit) {
   return function (req, res) {
      var limitDoc = limit || 10,
         name = req.query.nom;
      db.collection('dossiers', function (err, collection) {
         if (name) {
            collection.find({"nomComplet": new RegExp(name, "i")}, {"nomComplet": 1,
"codePerm": 1}).limit(limitDoc).toArray(function (err, items) {
               res.json(items);
            });
         } else {
            collection.find().limit(limitDoc).toArray(function (err, items) {
               res.json(items);
            });
         }
      });
   };
};

exports.ajouterEtudiant = function (db) {
```

```
   return function (req, res) {
      var etudiant = req.body;
      // Todo vérifier si _id et créer un ObjectId pour l'ajout.
      // Servira principalement pour les tests.
      console.log('Ajouter etudiant: ' + JSON.stringify(etudiant));
      db.collection('test', function (err, collection) {
         collection.insert(etudiant, {safe: true}, function (err, result) {
            if (err) {
               res.send('Erreur:' + JSON.stringify(err));
            } else {
               console.log('Succès:' + JSON.stringify(result[0]));
               res.send(result[0]);
            }
         });
      });
   };
};

exports.majEtudiant = function (db) {
   return function (req, res) {
      var id = req.params.id.toString(),
         etudiant = req.body;
      console.log('mise à jour(id): ' + id);
      console.log(JSON.stringify(etudiant));
      db.collection('test', function (err, collection) {
         collection.update({'_id': new ObjectId(id)}, etudiant, {safe: true}, function
(err, result) {
            if (err) {
               console.log("Erreur lors de la mise à jour de l'étudiant: " + err);
               res.send('Erreur:' + JSON.stringify(err));
            } else {
               console.log(String() + result + ' document(s) mise à jour(s).');
               res.send(result[0]);
            }
         });
      });
   };
};

exports.effacerEtudiant = function (db) {
   return function (req, res) {
      var id = req.params.id.toString();
      console.log('Effacer(id): ' + id);
      res.send(id);
      db.collection('test', function (err, collection) {
         collection.remove({'_id': new ObjectId(id)}, {safe: true}, function (err,
result) {
            if (err) {
               console.log("Erreur lors de l'effacement de l'étudiant: " + err);
               res.send('Erreur:' + JSON.stringify(err));
            } else {
               console.log(String() + result + ' document(s) retirés(s).');
               res.send(result[0]);
            }
         });
```

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

À première vue, la syntaxe du code est un peu complexe, mais tout ce qu'il fait est d'exporter les fonctions correspondantes aux opérations CRUD. Ce qu'il faut retenir ici, c'est que l'objet contenant la requête (*req*), nous permet d'extraire les informations pour effectuer l'opération à la base de données et que l'objet réponse(*res*), est enrichi par le retour de la base de données. La fonction « db.collection.[find | insert | update | remove] » est tout ce qui est nécessaire pour effectuer les opérations *CRUD* avec *MongoDB*.

Il est à remarquer que nous n'avons pas eu besoin d'un schéma ou d'une liste précise d'éléments d'information pour effectuer les opérations précédentes. *MongoDB* est une base de données « sans schéma » ce qui veut dire que même si nous ajoutons de nouveaux éléments dans une collection(équivalent d'une table) de la base de données, nous n'avons aucune autre application à modifier ou même à recompiler. Cette souplesse du schéma dynamique permet de commencer à travailler tout en modifiant ce dernier en cours de route lorsqu'il y a des changements de spécifications, ce qui est tout à fait normal. Dans cette preuve de concept, il y a plus de 300 éléments dans la structure hiérarchique « dossier étudiant » et nous n'avons rien eu à spécifier pour ajouter les services *REST*. De plus, ce simple service de mise à jour nous permet de modifier n'importe quelle partie de cette structure « dossier étudiant ».

Première utilisation de l'API avec *curl*

Afin d'essayer notre nouvel API, nous utiliserons la commande *curl*. Voici une liste des commandes pour appeler chacune des fonctions de l'API.

1. Obtenir la liste des étudiants.

```
curl -i H "Accept: application/json" -X GET http://localhost:3000/etudiants
```

2. Rechercher un étudiant par son nom.

```
curl -i H "Accept: application/json" -X GET http://localhost:3000/etudiants?nom=Martin
```

3. Ajouter un nouvel étudiant. Ici le « Content-Type » nous permet de passer le contenu de la création en JSON.

```
curl -i -H 'Content-Type: application/json' -H "Accept: application/json" -X POST -d '{"nomComplet":"Joe",
"adresses":[{"rue":"Berri"}]}'
```

4. Modifier l'adresse du nouvel étudiant. On utilise le « _id » généré par *MongoDB* lors de la requête précédente.

```
curl -i -H "Accept: application/json" -X DELETE http://localhost:3000/etudiants/52404e26bc6be46003000002
```

5. Effacer le document avec le « _id ».

```
curl -i -H "Accept: application/json" -X DELETE http://localhost:3000/etudiants/52404e26bc6be46003000002
```

À la suite des opérations 4 et 5, on utilise un *GET* avec le « _id » afin de valider l'opération précédente. Cette validation pourrait être déplacée dans le service lui-même.

Utilisation de l'API

```
$ curl -i H "Accept: application/json" -X GET  http://localhost:3000/etudiants
    HTTP/1.1 200 OK
    X-Powered-By: Express
    Content-Type: application/json; charset=utf-8
    Content-Length: 87692
    ETag: "441357573"
    Date: Mon, 23 Sep 2013 14:07:41 GMT
```

```
    Connection: keep-alive
    [
      {
        "_id": "5225e511ef863a1dea8d5f28",
        "codePerm": "XXXX00000000",
        "nomComplet": "Paul Illinois",
        "nom": "Illinois",
    ...


$ curl -i H "Accept: application/json" -X GET
http://localhost:3000/etudiants?nom=Martin
    HTTP/1.1 200 OK
    X-Powered-By: Express
    Content-Type: application/json; charset=utf-8
    Content-Length: 230
    Date: Mon, 23 Sep 2013 14:12:56 GMT
    Connection: keep-alive
    [
     {
      "_id": "5225e52bef863a1dea8d6f0b",
      "codePerm": "XXXX00000000",
      "nomComplet": "MARTIN PAILLER"
     },
     {
      "_id": "5225e52bef863a1dea8d6f12",
      "codePerm": "XXXX00000000",
      "nomComplet": "Martin Ciboulette"
     }
     ...


$ curl -i -H 'Content-Type: application/json' -H "Accept: application/json" -X POST -d
'{"nomComplet":"Joe", "adresses":[{"rue":"Berri"}]}' http://localhost:3000/etudiants
    HTTP/1.1 200 OK
    X-Powered-By: Express
    Content-Type: application/json; charset=utf-8
    Content-Length: 116
    Date: Mon, 23 Sep 2013 14:20:22 GMT
    Connection: keep-alive
    {
     "nomComplet": "Joe",
     "adresses": [
       {
         "rue": "Berri"
       }
     ],
     "_id": "52404e26bc6be46003000002"
     ...


$ curl -i -H 'Content-Type: application/json' -H "Accept: application/json" -X PUT -d
'{"nomComplet":"Joe Itel", "adresses":[{"rue":"Ste-Catherine"}]}'
http://localhost:3000/etudiants/52404e26bc6be46003000002
    HTTP/1.1 200 OK
    X-Powered-By: Express
    Date: Mon, 23 Sep 2013 14:24:14 GMT
    Connection: keep-alive
    Transfer-Encoding: chunked
```

```
$ curl -i H "Accept: application/json" -X GET
http://localhost:3000/etudiants/52404e26bc6be46003000002
    HTTP/1.1 200 OK
    X-Powered-By: Express
    Content-Type: application/json; charset=utf-8
    Content-Length: 129
    Date: Mon, 23 Sep 2013 14:25:50 GMT
    Connection: keep-alive
    {
     "_id": "52404e26bc6be46003000002",
     "nomComplet": "Joe Itel",
     "adresses": [
       {
         "rue": "Ste-Catherine"
       }
     ]
    }


$ curl -i -H "Accept: application/json" -X DELETE
http://localhost:3000/etudiants/52404e26bc6be46003000002
    HTTP/1.1 200 OK
    X-Powered-By: Express
    Content-Type: text/html; charset=utf-8
    Content-Length: 24
    Date: Mon, 23 Sep 2013 14:28:08 GMT
    Connection: keep-alive


$ curl -i H "Accept: application/json" -X GET
http://localhost:3000/etudiants/52404e26bc6be46003000002
    HTTP/1.1 200 OK
    X-Powered-By: Express
    Content-Type: application/json; charset=utf-8
```

```
    Content-Length: 4
    Date: Mon, 23 Sep 2013 14:29:38 GMT
    Connection: keep-alive
```

Tests unitaires

Une application n'est pas terminée si elle n'est pas testée! Afin de tester l'ensemble des fonctionnalités de notre application, il faut diviser les tests en 2 sections. Une première qui va tester l'accès Web et l'autre qui testera les API. Évidemment, pourquoi refaire manuellement les tests quand il est possible d'automatiser les tests unitaires. Pour se faire, nous utiliserons ici *mocha* qui est un excellent outil de tests (facilitant les tests asynchrones) avec *Express.js*. Les tests unitaires sont un sujet en soi dépassant de beaucoup cette présentation. Nous n'expliquerons pas en détail, mais survolerons plutôt la création de ces tests.

Installation

Pour la même raison qu'*Express.js*, nous installons *mocha* d'une façon globale afin de rendre accessible la commande *mocha* à tous nos projets. Nous devons également modifier notre fichier de configurations *packages.json* pour ajouter *mocha* et *SuperTest* que nous utiliserons également dans nos tests.

package.json

```
{
  "name" : "accesDA",
  "version" : "0.0.1",
  "private" : true,
  "scripts" : {
    "start" : "node app.js"
  },
  "dependencies" : {
    "express" : "3.4.0",
       "jade" : "*",
    "mongodb" : "*",
      "mocha" : "*",
   "supertest": "*",
  }
}
```

Installation de moche

```
$ sudo npm -g install mocha
npm http GET https://registry.npmjs.org/mocha
npm http 200 https://registry.npmjs.org/mocha
npm http GET https://registry.npmjs.org/mocha/-/mocha-1.13.0.tgz
...
/usr/local/bin/_mocha -> /usr/local/lib/node_modules/mocha/bin/_mocha
mocha@1.13.0 /usr/local/lib/node_modules/mocha
 growl@1.7.0
 diff@1.0.7
 debug@0.7.2
 commander@0.6.1
 mkdirp@0.3.5
 jade@0.26.3 (mkdirp@0.3.0)
 glob@3.2.3 (graceful-fs@2.0.1, inherits@2.0.1, minimatch@0.2.12)

$ npm install
npm http GET https://registry.npmjs.org/supertest
...
 methods@0.0.1
 superagent@0.15.1 (emitter-component@1.0.0, cookiejar@1.3.0, debug@0.7.2, qs@0.6.5,
mime@1.2.5, formidable@1.0.9)

$
```

La commande *mocha* vérifie s'il existe un répertoire nommé « test » dans le projet courant et exécute les fichiers JavaScript s'y trouvant. Nous allons donc créer un premier test afin de tester l'accès Web. D'abord, veuillez créer le répertoire « test » et y ajoutez un premier fichier de tests « accesDA.js ».

Tests accesDA.js

```
/*jslint */
/*global describe, it */


var app    = require('../app'),
   assert  = require('assert'),
   request = require('supertest');


describe('Test app', function () {
   'use strict';
   var url = 'http://localhost:3000';
   var cId;

   /////////// ACCES WEB
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////
///////
   describe('Accès Web', function () {
      it('GET /index.html devrait retourner le statut 200, le type de contenu html et
«Bonjour Express!».', function (done) {
         request(url).get('/index.html')
            .expect('Content-Type', /html/)
            .expect(200)
            .expect(/Bonjour Express!/, done);
      });
   });
});
```

Ce premier test va demander le fichier « /index.html » et valider que le contenu est de type HTML et que la requête retourne le code *HTTP* 200 qui équivaut à succès. De plus, on vérifie que la page contient « Bonjour Express! ».

Il faut maintenant ajouter les tests pour chacune des fonctions de l'API. Pour ce faire nous effectuons les opérations suivantes dans l'ordre :

- Demander une liste d'étudiants. **GET /etudiants/** ;
    - Vérifier que le code *HTTP* a la valeur 200, que le type est *JSON*, et que la taille du tableau de retour est plus grand que 1.
- Ajouter un nouvel étudiant. **POST** /etudiants ;
    - {nomComplet : 'Joe', adresses : [{"rue" : "Berri"}]} ;
    - Vérifier que le code *HTTP* a la valeur 200, que le type est *JSON*, et que le « _id » de *MongoDB* a une longueur de 24 caractères.
- Tenter d'ajouter un doublon. **POST** /etudiants ;
    - {nomComplet : 'Joe', adresses : [{"rue" : "Berri"}]} ;
    - Vérifier que rien n'est retourné, puisqu'il s'agit d'un doublon.
- Modifier le nouvel étudiant. **PUT** /etudiants ;
    - {nomComplet : 'Joe Itel', adresses : [{"rue" : "Ste-Catherine"}]} ;
- Vérifier la modification. **GET** /etudiants/:id ;
- Effacer le nouvel étudiant. **DELETE** /etudiants/:id ;
- Vérifier l'effacement. **GET** /etudiants/:id .

Voici le code nécessaire pour la vérification de l'API.

Tests unitaires de l'API

```
/////////// API
////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
///////


   describe('Accès API', function () {

      it('GET /etudiants. Devrait retourner le statut 200 et un array de type json >
1.', function (done) {
         request(url).get('/etudiants/')
            .expect('Content-Type', /json/)
            .expect(200)
            .end(function (err, res) {
               if (err) {
                  throw err;
               }
               assert(JSON.parse(res.text).length > 1);
               done();
            });
      });

      it('POST /etudiants. Devrait ajouter un nouvel étudiant.', function (done) {
         var body = {
            codePerm : 'ZZZZ10000000',
            nomComplet : 'Joe',
            adresses : [{"rue" : "Berri"}]
         };
         request(url).post('/etudiants').send(body)
            .expect('Content-Type', /json/)
            .expect(200)
            .end(function (err, res) {
               if (err) {
                  throw err;
               }
               assert(res.body._id.length === 24);
               cId = res.body._id;
               done();
            });
      });

      it('POST /etudiants. Tenter d\'ajouter un nouvel étudiant en double.', function
(done) {
         var body = {
            codePerm : 'ZZZZ10000000',
            nomComplet : 'Joe',
            adresses : [{"rue" : "Berri"}]
         };
         request(url).post('/etudiants').send(body)
            .expect('Content-Type', /html/)
            .expect(200)
            .end(function (err, res) {
               if (err) {
                  throw err;
               }
               assert(JSON.stringify(res.body) === '{}');
               done();
            });
      });

      it('PUT /etudiants. Devrait modifier l\'étudiant nouvelement ajouté.', function
```

```
(done) {
         var body = {
            nomComplet : 'Joe Itel',
            adresses : [{"rue" : "Ste-Catherine"}]
         };
         request(url)
            .put('/etudiants/' + cId)
            .send(body)
            .expect(200)
            .end(function (err, res) {
               if (err) {
                  throw err;
               }
               done();
            });
      });

      it('GET /etudiants devrait récupérer l\'étudiant.', function (done) {
         request(url)
            .get('/etudiants/' + cId)
            .expect('Content-Type', /json/)
            .expect(200)
            .end(function (err, res) {
               if (err) {
                  throw err;
               }
               assert(res.body.nomComplet === 'Joe Itel');
               done();
            });
      });

      it('DELETE /etudiants. Devrait affacer l\'étudiant.', function (done) {
         request(url)
            .del('/etudiants/' + cId)
            .expect(200)
            .end(function (err, res) {
               if (err) {
                  throw err;
               }
               done();
            });
      });

      it('GET /etudiants. Ne devrait pas récupérer l\'étudiant.', function (done) {
         request(url)
            .get('/etudiants/' + cId)
            .expect('Content-Type', /json/)
            .expect(200)
            .end(function (err, res) {
               if (err) {
                  throw err;
               }
               assert(JSON.stringify(res.body) === '{}');
               done();
```

```
            });
        });
    }); // Accès API
```

Il ne reste qu'à exécuter les tests unitaires en entrant la commande *mocha*.

Exécution des tests unitaires

```
$ mocha --reporter spec

Express server listening on port 3000
  Test app
    Accès Web
       GET /index.html devrait retourner le statut 200, le type de contenu html et
«Bonjour Express!».: GET /index.html 200 6ms - 649b
       GET /index.html devrait retourner le statut 200, le type de contenu html et
«Bonjour Express!».
    Accès API
       GET /etudiants. Devrait retourner le statut 200 et un array de type json > 1.:
GET /etudiants/ 200 17ms - 85.64kb
       GET /etudiants. Devrait retourner le statut 200 et un array de type json > 1.
       POST /etudiants. Devrait ajouter un nouvel étudiant.: POST /etudiants 200 5ms -
146b
       POST /etudiants. Devrait ajouter un nouvel étudiant.
       POST /etudiants. Tenter d'ajouter un nouvel étudiant en double.: POST
/etudiants 200 1ms - 171b
       POST /etudiants. Tenter d'ajouter un nouvel étudiant en double.
       PUT /etudiants. Devrait modifier l'étudiant nouvelement ajouté.: PUT
/etudiants/524094876d3ff0be0e000001 200 2ms
       PUT /etudiants. Devrait modifier l'étudiant nouvelement ajouté.
       GET /etudiants devrait récupérer l'étudiant.: GET
/etudiants/524094876d3ff0be0e000001 200 1ms - 159b
       GET /etudiants devrait récupérer l'étudiant.
       DELETE /etudiants. Devrait affacer l'étudiant.: DELETE
/etudiants/524094876d3ff0be0e000001 200 0ms - 24b
       DELETE /etudiants. Devrait affacer l'étudiant.
       GET /etudiants. Ne devrait pas récupérer l'étudiant.: GET
/etudiants/524094876d3ff0be0e000001 200 1ms - 4b
       GET /etudiants. Ne devrait pas récupérer l'étudiant.

  8 passing (76ms)
```

Conclusion

Un certain nombre d'avantages se profilent suite à l'utilisation de cette architecture « jMEAN ». Nous allons passer rapidement en revue certains de ces avantages.

JavaScript de bout en bout

Un premier avantage est l'utilisation de JavaScript de bout en bout dans ce « stack » technologique. Même si nous n'avons pas encore vu la partie cliente de l'application, tant cette dernière que les parties métier (*Node.js*) et persistance (MongoDB) utilisent *JavaScript* comme langage de programmation. Le fait d'utiliser qu'un seul langage permet aux développeurs de garder l'accent sur un même langage et architecture pour ainsi perdre moins de temps lorsqu'ils doivent travailler sur une partie différente de l'application.

Le code source commun peut servir autant la partie client que la partie serveur de l'application, ce qui évite d'écrire les mêmes fonctionnalités

dans différents langages. Combiné au fait que JavaScript est un langage dynamique qui requiert généralement moins de code que d'autres langages, les applications résultantes sont donc moins volumineuses. Moins de code dans un seul langage se traduit généralement par moins de bogues, moins de temps de développement et de maintenance ainsi qu'une plus grande facilité lors de la création des tests et de l'utilisation des outils pour l'automatisation de l'assemblage et du déploiement.

Un deuxième avantage découlant de JavaScript est l'utilisation d'objets de bout en bout. Les objets de la base de données et du client JavaScript sont donc identiques à une exception près. Ce qui permet d'éviter les nombreuses conversions requises par l'utilisation d'un modèle de données relationnel. Lorsqu'on utilise une conversion, nous avons généralement une couche qui effectue l'association entre un objet et sa représentation relationnelle l'«Object Relational Mapping»(*ORM*) qui demande généralement beaucoup de ressources. La représentation des objets JavaScript *JSON* permet une circulation bilatérale des objets métiers sans se soucier de leur format. Comme ces objets sont naturels dans JavaScript, autant le fureteur que la base de données peuvent les manipuler. Pour le programmeur, le fait d'utiliser un seul type d'objet est un gain au niveau cognitif.

JavaScript n'a pas que des avantages, pour faire des applications bien structurées, ce langage requiert une grande discipline. De plus, il demande une bonne connaissance de certaines particularités de JavaScript par exemple la portée des variables, la notion d'objet basée sur prototypes plutôt que des classes, les fonctions anonymes, la fonction comme objet à part entière, etc.

Performance et scalabilité

Un des secrets de la grande scalabilité de *Node.js* est le traitement des requêtes *HTTP* par une boucle événementielle (*event-loop*). Cette architecture fondamentale à *Node.js*, fournit un modèle fort différent pour la manipulation des connexions clientes par rapport au serveur web traditionnel qui demande l'utilisation de « threads » ou des processus pour chacune des connexions clientes.

La boucle événementielle sur laquelle sont bâties les applications *Node.js* traite de manière transparente les connexions de vos clients en consommant très peu de ressource. L'utilisation de la mémoire par chaque client est négligeable, ainsi *Node.js* peut gérer des centaines de fois plus de clients qu'un serveur traditionnel.

Le modèle d'entrée/sortie asynchrone permet à *Node.js* de traiter plusieurs requêtes parallèlement sans attendre de réponse sérielle pour ses entrées/sorties. Ce qui a pour effet de permettre un type d'application ayant beaucoup de requêtes simultanées. Avons-nous besoin de cette performance et scalabilté pour accéder à un dossier académique? Nous n'avons pas encore vu la partie client, mais imaginez une recherche dynamique par nom où chaque fois que l'utilisateur entre un caractère, une requête (ajax) demande la liste des étudiants correspondant à cette recherche partielle, à chaque fois qu'un caractère est entré une requête se rend à la base de données. Imaginons maintenant quelque centaines d'utilisateurs qui effectuent une recherche simultanément, vous pouvez imaginer le nombre de requêtes si chacun d'eux entre quelques caractères à la seconde. Ici non seulement le serveur *Node.js* sera sollicité, mais également la base de données, le choix d'une base de données noSQL peut ici faire une différence.

Simplicité architecturale

Découlant des avantages précédents, à plus haut niveau, se dégage une simplicité architecturale (illustrée au début de l'article) par l'utilisation de *REST* et de JSON qui permet le développement d'une application hautement réactive, type d'application qui répond aux besoins d'une expérience efficace et agréable pour l'utilisateur. Nous discuterons plus en détail d'architecture dans l'article sur les services transversaux.

Le prochain article portera sur la partie « client » de cette preuve de concept. Le code source de la partie couverte par cet article sera accessible d'ici quelques jours sur Github.

Bibliographie

Michael Mikowski et Josh Powell, *Single Page Web Applications,* Manning Publications, 2013, ISBN 978-1617290756.

Mike Cantelon, TJ Holowaychunk, Nathan Rajlich. *Node.js in action,* Manning Publications, 2013, ISBN 978-1617290572.

Kristina Chodorow, *MongoDB: The Definitive Guide,* O'Reilly Media, 2e Edition, 2013, ISBN 978-1449344689.

Brad Green, Shyam Seshadri, *AngularJS,* O'Reilly Media, 2013, ISBN 978-1449344856.

Firtman, Maximiliano , *Up and Running with jQuery Mobile*, O'Reilly Media, 2010. ISBN 9781935182801