



NODE-JS / EXPRESS / HANDLEBARS / MONGO-DB

par Stéphane MASCARON Architecte Logiciels Libres
Formation Greta des Landes DAX - Sud Aquitaine

Node, Express, MongoDB

• Installation d'express

- *Pour créer des applications **express** il existe plusieurs possibilités, la plus souple semble être l'installation du module express-generator*

```
$ npm install express-generator -g
```

- *Ensuite il peut être nécessaire de configurer un lien dans /usr/bin pour accéder à la commande "express" du module express-generator.*

```
$ sudo ln -s <chemin vers votre module>/express-generator/bin/express  
/usr/bin/express
```

NB : Cela peut arriver si vous utilisez l'outil "nvm" qui stocke les nodejs dans le dossier **'.nvm'** de votre **/home**.

Node, Express, MongoDB

• Installation d'express (suite)

- *Maintenant que vous avez un express générateur d'application vous pouvez l'utiliser pour créer votre première application Express :*

```
stephane@UX303UB:~/workspaceGreta$ express --view=hbs myExpressHbsApp
```

```
create : myExpressHbsApp
create : myExpressHbsApp/package.json
create : myExpressHbsApp/app.js
create : myExpressHbsApp/public
create : myExpressHbsApp/public/javascripts
create : myExpressHbsApp/routes
create : myExpressHbsApp/routes/index.js
create : myExpressHbsApp/routes/users.js
create : myExpressHbsApp/views
create : myExpressHbsApp/views/index.hbs
create : myExpressHbsApp/views/layout.hbs
create : myExpressHbsApp/views/error.hbs
create : myExpressHbsApp/bin
create : myExpressHbsApp/bin/www
create : myExpressHbsApp/public/images
create : myExpressHbsApp/public/stylesheets
create : myExpressHbsApp/public/stylesheets/style.css
install dependencies:
$ cd myExpressHbsApp && npm install
run the app:
$ DEBUG=myexpresshbsapp:* npm start
```

Node, Express, MongoDB

• Configuration première application

- *Nous allons configurer les dépendances de notre première application Express :*

```
stephane@UX303UB:~/workspaceGreta/myExpressHbsApp$ npm install
```

- *La configuration est terminée lorsque le prompt revient :*

```
(...)  
├── morgan@1.7.0  
│   ├── basic-auth@1.0.4  
│   ├── debug@2.2.0  
│   │   └── ms@0.7.1  
│   └── on-headers@1.0.1  
└── serve-favicon@2.3.2  
stephane@UX303UB:~/workspaceGreta/myExpressHbsApp$
```

Node, Express, MongoDB

- *Exécution de votre première application Express :*

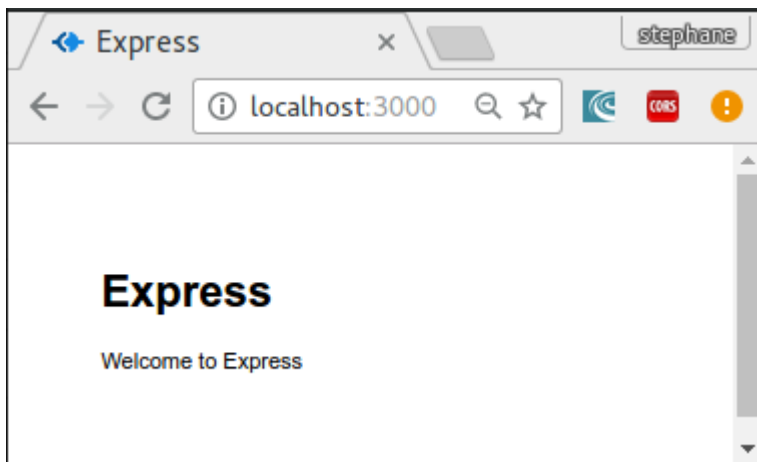
```
stephane@UX303UB:~/workspaceGreta/myExpressHbsApp$ npm start
```

```
> myexpresshbsapp@0.0.0 start /home/stephane/workspaceGreta/myExpressHbsApp  
> node ./bin/www
```

- *Lancez un navigateur sur l'url suivante :*

```
http://localhost:3000
```

- *Vous devriez voir ceci dans le navigateur et les logs dans la console:*



```
stephane@UX303UB:~/workspaceGreta/myExpressHbsApp$ npm start  
  
> myexpresshbsapp@0.0.0 start /home/stephane/workspaceGreta/myExpressHbsApp  
> node ./bin/www  
  
GET / 200 61.222 ms - 204  
GET /stylesheets/style.css 200 4.395 ms - 111
```

Node, Express, MongoDB

• Analyse du code généré

- *Nous allons regarder le code généré pour cette application : arborescence description :*

```
stephane@UX303UB:~/workspaceGreta/myExpressHbsApp$ tree
```

```
├── app.js ..... le code source de l'application
├── bin ..... le dossier du lanceur
│   └── www ..... le lanceur de l'application (main)
├── package.json ..... la config de l'app + les dépendances
├── public ..... le dossier public du serveur web node
│   ├── images ..... répertoire des images clientes
│   ├── javascripts ..... répertoire des fichier JS clients
│   ├── stylesheets ..... répertoire des feuilles de style css
│   │   └── style.css ..... une feuille css
├── routes ..... répertoire contenant les Controleurs
│   ├── index.js ..... controleur sur action '/'
│   └── users.js ..... controleur sur action '/user'
└── views ..... répertoire contenant les Vues
    ├── error.hbs ..... Vue 'error'
    ├── index.hbs ..... Vue 'accueil' pour action '/'
    └── layout.hbs ..... Découpage de l'interface (head, body, footer, nav)
```

Node, Express, MongoDB

- **Les vues utilisent handlebars**

- *Un premier fichier est créé pour définir le “layout”, c’est à dire le découpage de l’interface :*

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{title}}</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

- *Le code HTML des vues : “**index.hbs**” et “**error.hbs**” seront affichées à la place du tag {{{body}}}.*

Node, Express, MongoDB

- **Découpons plus en détail l'interface.**

- Il faut pour ajouter des “**partials**”, c’est à dire des blocs de HTML qui seront assemblés pour créer l’interface sur le navigateur. Dans `app.js` :

```
(...)  
var hbs = require('hbs');  
hbs.registerPartials(__dirname + '/views/partials', function() {  
  console.log('partials registered');  
});  
var app = express();  
(...)
```

- Ensuite vous allez dans le dossier ‘**views**’ créer un répertoire ‘**partials**’. Dans ce dossier créez un fichier nommé ‘**head.hbs**’ :

```
<head>  
  <title>{{title}}</title>  
  <link rel='stylesheet' href='/stylesheets/style.css' />  
</head>
```


Node, Express, MongoDB

- *Découpons plus en détail l'interface*
 - *Le code du **layout.hbs** va changer lui aussi :*

```
<!DOCTYPE html>
<html>
  {{> head}}
  <body>
    {{{body}}}
  </body>
</html>
```

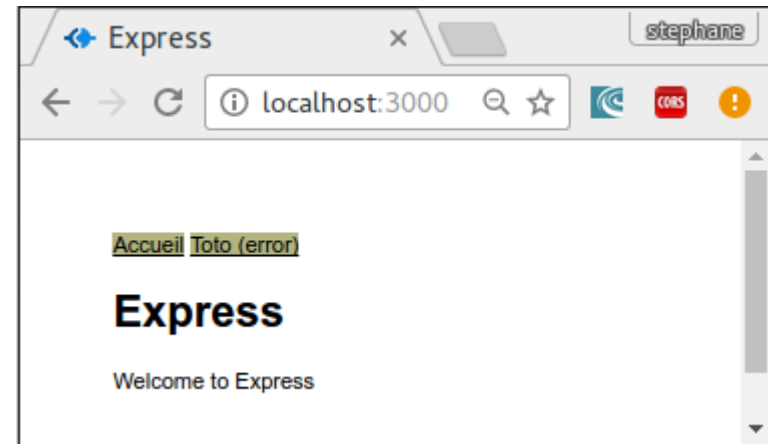
- *De la même façon nous allons ajouter une barre de navigation (lien href). On crée un fichier **nav.hbs** dans 'partials' :*

```
<nav class="barnav">
  <ul id="nav">
    <li><a href="/">Accueil</a></li>
    <li><a href="/toto">Toto (error)</a></li>
  </ul>
</nav>
```

Node, Express, MongoDB

- *Découpons plus en détail l'interface*
 - *Le fichier **layout.hbs** a changé lui aussi :*

```
<!DOCTYPE html>
<html>
  {{> head}}
  <body>
    {{> nav}}
    {{{body}}}}
  </body>
</html>
```



- *La feuille **style.css** est à modifier aussi :*

```
.barnav {
  width: 100%;
}

#nav {
  margin:0;
  padding:0;
}
(...) //→ colonne suivante
```

```
#nav li {
  display:inline;
  padding:0;
  margin:0;
}
#nav a:link, #nav a:visited {
  color:#000;
  background:#b2b580;
}
```

Node, Express, MongoDB

- **Exercice : ajouter un footer à notre application**

- En utilisant les exemples précédents, ajouter un **footer.hbs** donnant le nom de votre entreprise ;-) la date les logos twitters ce que vous voulez
- Correction : footer.hbs

```
<footer class="footer">  
  <h1> ceci est le footer : &copy; SMaLL 2017</h1>  
</footer>
```

- modifiez style.css

```
.footer {  
  font: 9px "Arial";  
  background:#b2b580;  
}
```

Node, Express, MongoDB

- **Accéder à une base de données**

- Pour accéder à une base de données nous allons installer mongodb sur notre poste :
- A adapter en fonction de votre système d'exploitation.
- Dans le répertoire de votre application vous devez ajouter mongodb aux dépendances et dans les **node_modules** :

```
$ sudo apt install mongodb
```

```
$ npm install mongodb --save
```

- Nous allons faire des tests dans un module pour valider la connexion avec la base de données 'testmongodb.js' :

Node, Express, MongoDB

• Accéder à une base de données :

```
var mongoClient = require('mongodb').MongoClient
    , assert = require('assert');
var url = 'mongodb://localhost:27017/myproject'; // Connection URL
// Use connect method to connect to the server
MongoClient.connect(url, function(err, db) {
    assert.equal(null, err);
    console.log("Connected successfully to server");
    db.close();
});
```

- Il faut dans un terminal que vous lancier la base de données MongoDB :

```
$ sudo mongod
(...) [initandlisten] waiting for connections on port 27017
```

- Il est temps de tester la connexion :

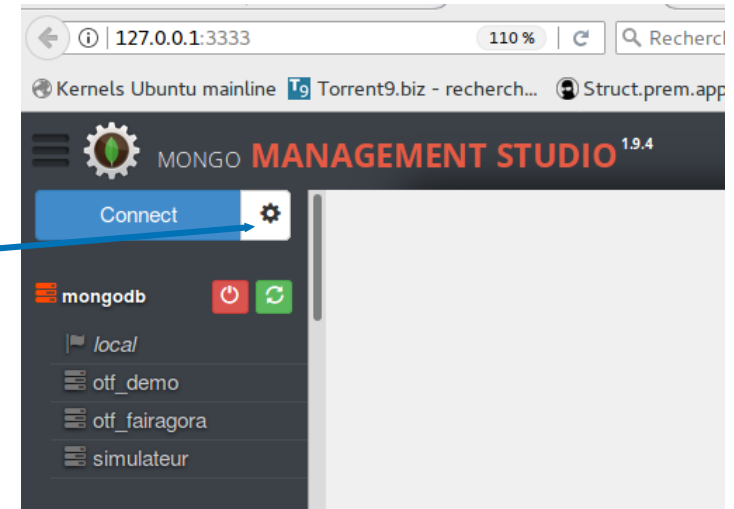
```
$ node ./testmongodb.js
Connected successfully to server
```

- Nous allons pouvoir utiliser cette connexion dans notre application

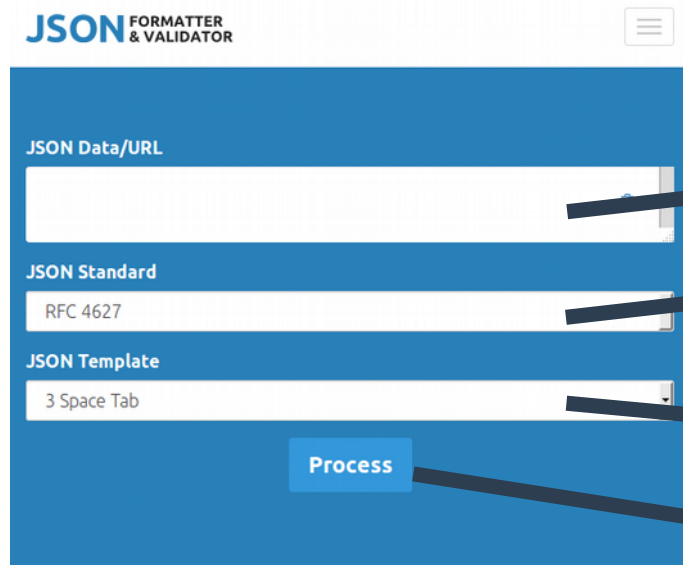
Node, Express, MongoDB

- Outils pour le développement avec mongoDB :

- Mongo Management Studio cliquez sur cet icône pour la configuration de MMS (!= types d'affichage)



- **JSON Formatter-Validator**



Coller ici le texte en JSON

Format RFC (laissez par défaut)

Indentation du code JSON

Cliquez sur "Process"

Node, Express, MongoDB

- Ajoutons une connexion base de données pour notre action **‘/exos’**
 - Créons la connexion dans **‘app.js’** :

```
(...)  
// Configuration de la connexion à la base de données :  
GLOBAL.db={};  
var mongoClient = require('mongodb').MongoClient;  
// Connection URL  
var url = 'mongodb://127.0.0.1:27017/gretajs';  
// Use connect method to connect to the server  
mongoClient.connect(url, function(err, db) {  
    GLOBAL.db = db; //astuce pour le moment on met en global la connexion  
    console.log("Connected successfully to server: GLOBAL.db initialized ");  
});  
module.exports = app;  
(...)
```

Node, Express, MongoDB

- Modification du code de **'exos.js'** :

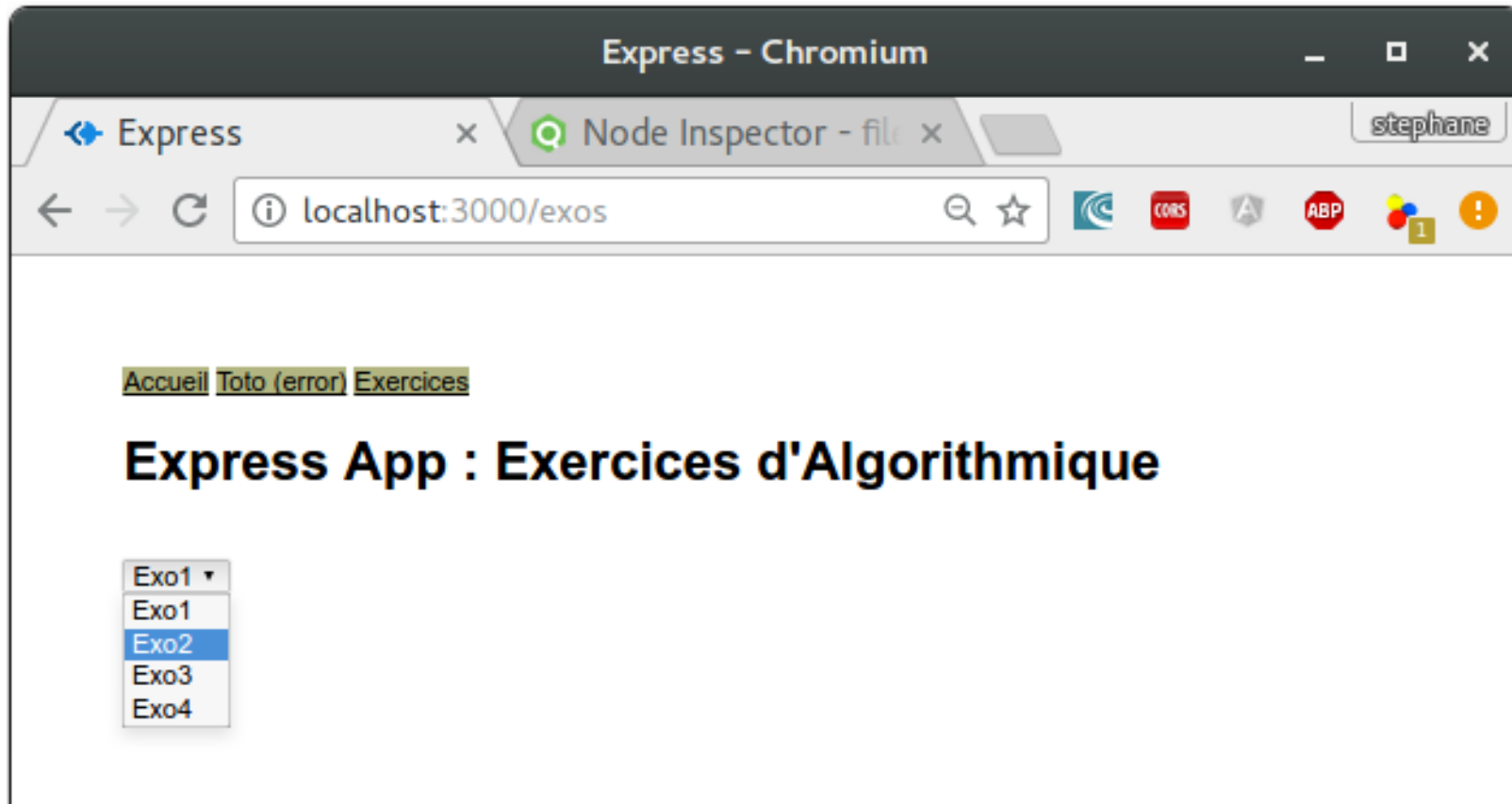
```
var express = require('express');
var router = express.Router();
/* GET home page. */
router.get('/', function(req, res, next) {
  GLOBAL.db.collection('exercices').find().toArray(function(err, result) {
    if (err) {
      throw err;
    }
    console.log(result);
    res.render('exos', {title: 'Express', exos: result[0]});
  });
});
module.exports = router;
```

- Voyons le code dans la vue **'exos.hbs'** pour afficher une liste déroulante

```
<h1> Express App : {{exos.titre}}</h1><br>
<select name="exos">
  {{#each exos.liste}}
    <option value="{{this}}">{{this}}</option>
  {{/each}}
</select>
```


Node, Express, MongoDB

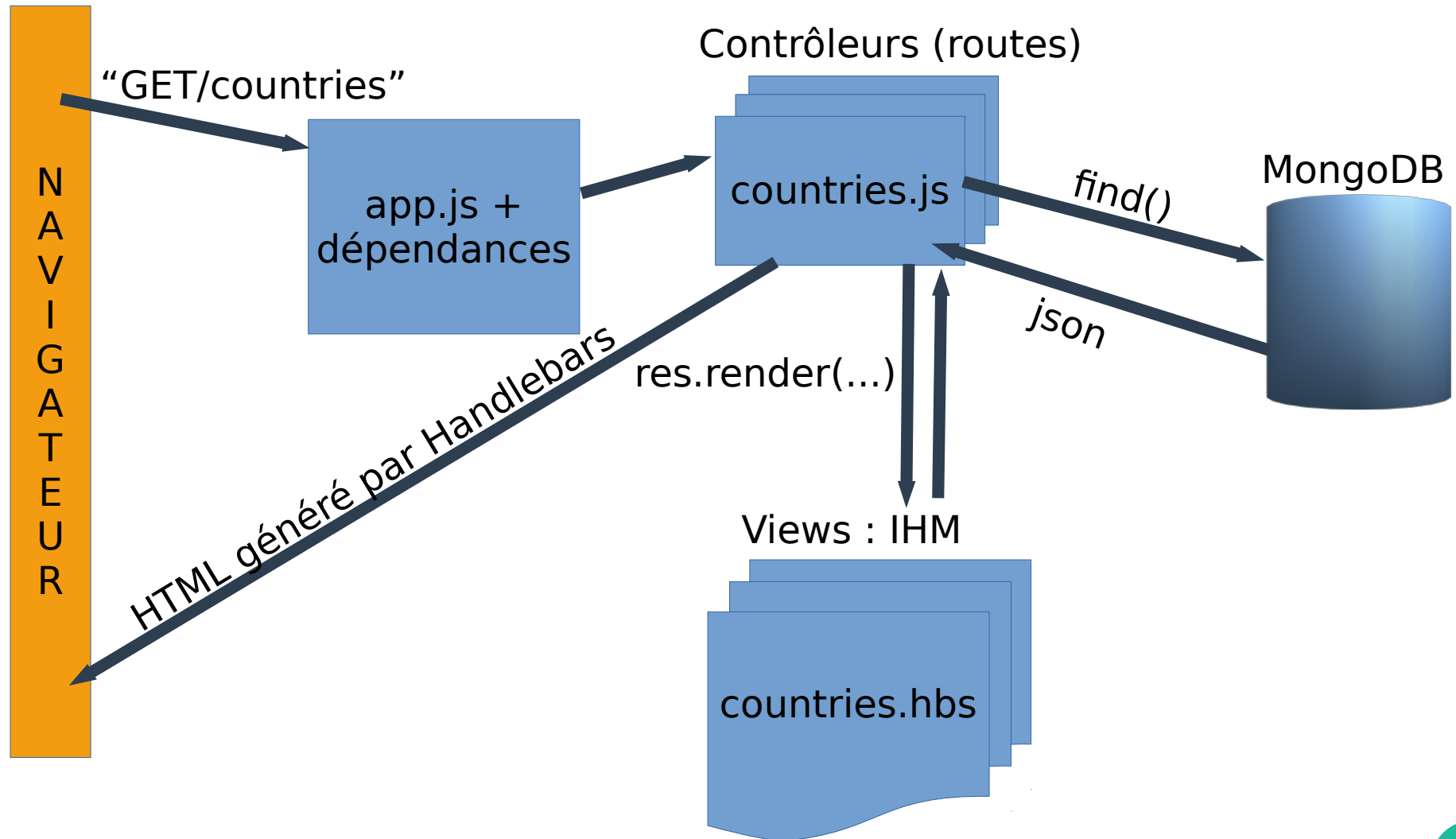
- **Résultat de l'appel de l'action 'exos' :**



NB : Attention en copiant depuis le PDF il se peut que des caractères invisibles déclenche des erreurs silencieuses

Node, Express, MongoDB

- Schéma architecture Application Express/Handlebars/MongoDB :



Node, Express, MongoDB

• Exercice :

- En utilisant le code vu précédemment, vous ajouterez une action nommée **'/countries'** un contrôleur **'countries.js'** et une vue **'countries.hbs'** qui affiche la liste des pays dans un select.
- Le fichier countries est dans vos mails. (ou [ici](#))

```
[  
  {  
    "_id": "572377f1e3d6f2d245000001",  
    "name": "Afghanistan",  
    "code": "AF",  
    "created_date": "2016-06-20T11:14:24.525Z"  
  },  
  (...)
```

- Vous pouvez l'importer via Mongo Management Studio dans la base **'gretajs'**.

Node, Express, MongoDB

- **Helpers ! Pour plus de souplesse dans les templates :**
 - Un Helper dans Handlebars c'est une méthode qui va réaliser un traitement pendant la fusion entre les données JSON et la vue HTML.
 - On va utiliser la méthode suivante pour enregistrer un helper :
- ```
hbs.registerHelper('helper_name', function(...) { ... });
```
- On peut créer des fonctions qui vont permettre de faire des comparaisons, des conversions, l'intégration de composants.

# Node, Express, MongoDB

- **Intégration dans notre 'app.js'**

- Ajoutons un simple Helper qui va comparer 2 chaînes de caractères :

```
hbs.registerHelper('compare', function (lvalue, rvalue, options) {
 console.log("##### COMPARE lvalue :",lvalue," et rvalue: ",rvalue);
 if (arguments.length < 3)
 throw new Error("Handlerbars Helper 'compare' needs 2 parameters");
 var operator = options.hash.operator || "==";
 var operators = {
 '==': function (l, r) {
 return l == r;
 }
 }
 if (!operators[operator])
 throw new Error("'compare' doesn't know the operator " + operator);
 var result = operators[operator](lvalue, rvalue);
 if (result) {
 return options.fn(this);
 } else {
 return options.inverse(this);
 }
});
```

# Node, Express, MongoDB

- **Utilisation du Helper :**

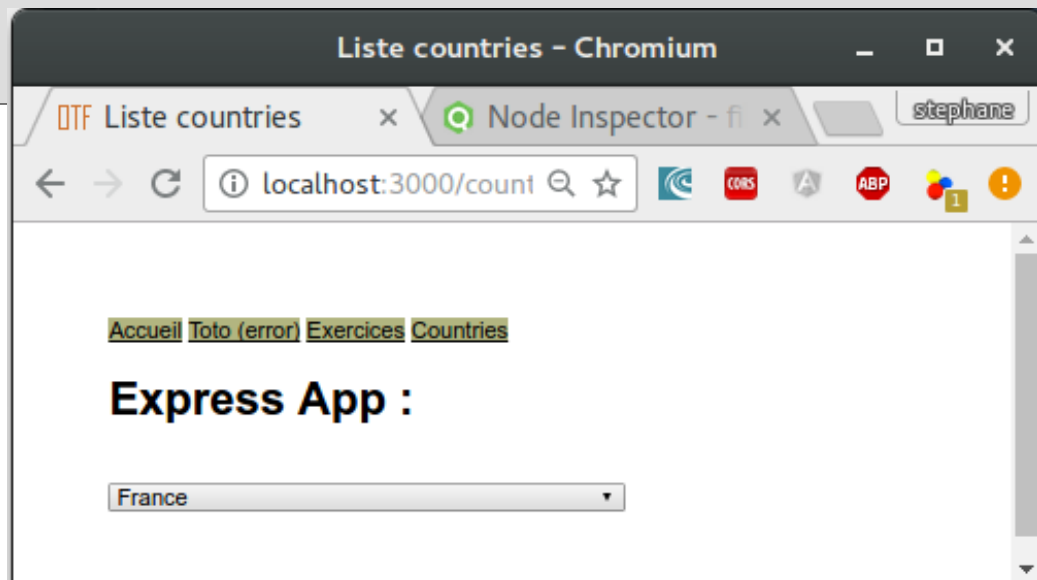
- Dans le fichier countries.hbs nous allons ajouter un test dans la syntaxe Handlebars :

```
<h1> Express App : {{country.title}}</h1>

<select name="country">
 {{#each country}}
 {{#compare this.code 'FR' operator==='}}
 <option value="{{this.code}}" selected>{{this.name}}</option>
 {{else}}
 <option value="{{this.code}}">{{this.name}}</option>
 {{/compare}}
 {{/each}}
</select>
```

- **Résultat :**

nb : pensez à mettre à jour le menu '**nav.hbs**'



# Node, Express, MongoDB

- **Il existe des modules qui mettent à disposition des Helpers déjà écrit.**
  - Il est toujours intéressant de savoir comment on écrit ce genre de code, ça peut vous sauver un contrat pendant un dev ou une charrette de dév chez le client :-)
    - `hbs-helpers`
    - `handlebars-helpers`
  - Vous pourrez également regarder les codes source d'**OTF<sup>2</sup>** qui contient plusieurs helpers développés au fur et à mesure des besoins.

# Node, Express, MongoDB

- **Exercice :**

- Ajouter un nouvel Helper qui permet de savoir si un tableau est vide ou non ?

On pourra utiliser le nom suivant pour le Helper : **'isEmptyTab'**

- On pourra utiliser cet Helper pour modifier l'interface de notre liste countries si la base de données nous retourne un tableau vide.



# Node, Express, MongoDB

- **Exercice correction (sans ajouter un helper) mais un opérateur :**

- On crée la fonction **'isEmptyTab'** dans app.js :

```
var operators = {
 '==': function (l, r) {
 if (l == r) console.log(l + ' - ' + r + ' : ' + (l == r));
 return l == r;
 },
 'isEmptyTab': function (obj) {
 if (!obj || obj.length == 0)
 return true;
 return false;
 }
}
```

- Code permettant d'appeler l'opérateur **'isEmptyTab'** :

```
{{#compare country null operator="isEmptyTab"}}
 <option value="rien">Pas de données dans la base</option>
 {{else}}
 {{#each country}}
 (...) // le code existant ici
 {{/each}}
 {{/compare}}
```

# Node, Express, MongoDB

- **Nous allons ajouter la mise à jour dans notre Express App.**
  - Pour ce faire nous allons créer une nouvelle collection dans **MongoDB**, via **Mongo Mangement Studio** pour stocker des utilisateurs : “**users**”. On créé 1 user :

```
{
 "name": "MASCARON",
 "firstName": "Stéphane",
 "login": "steph",
 "mdp": "azerty",
 "function": "Free Software Architecte",
 "office": "SMaLL Stéphane MASCARON ARchitecte Logiciels Libres",
 "date_naiss": "Wed Apr 22 1970 09:30:22 GMT+0100 (CET)",
 "adresses1": "260 avenue des arènes",
 "adresse2": "",
 "cp": "40090",
 "city": "SAINT-PERDON",
 "mobile_phone": "0609090909",
 "home_phone": "0558080808",
}
```

# Node, Express, MongoDB

- Nous devons créer **un contrôleur**, **une vue** et **une action** qui affiche la liste des users : **“/users”** dont le contrôleur sera **“users.js”** :

```
var express = require('express');
var router = express.Router();

/* GET users listing. */
router.get('/', function(req, res, next) {
 GLOBAL.db.collection('users').find().toArray(function(err, result) {
 if (err) {
 throw err;
 }
 console.log('users: ', result);
 res.render('users', {title: 'List of users', users: result});
 });
});

module.exports = router;
```

**NB:** Dans ce contrôleur nous réalisons une requête sur la collection “users” pour laquelle on récupère l’ensemble des utilisateurs (db.collection(‘users’).find() ...)

# Node, Express, MongoDB

- Il faut créer la vue pour cette liste des utilisateurs, on va utiliser un select pour afficher les utilisateurs : **"users.hbs"**

```
<h1> Express App : {{title}}</h1>

<select name="users" id="user">
 {{#compare users null operator="tabempty"}}
 <option value="rien">Il n'y a pas de données dans la base users</option>
 {{else}}
 <option value="0" >Select a user to modify his datas</option>
 {{#each users}}
 <option value="{{this._id}}" >{{this.name}} {{this.firstName}}-
 {{this.function}}</option>
 {{/each}}
 {{/compare}}
</select>
<h2>or</h2>

<button id="newUser">Create a new user</button>
<script>
 var usr = document.getElementById('user');
 usr.addEventListener('change', function(evt) {
 window.location = "/formUser/"+usr.value;
 });
 var btn_new = document.getElementById('newUser');
 btn_new.addEventListener('click', function(evt) {
 window.location = "/newUser";
 });
</script>
```

# Node, Express, MongoDB

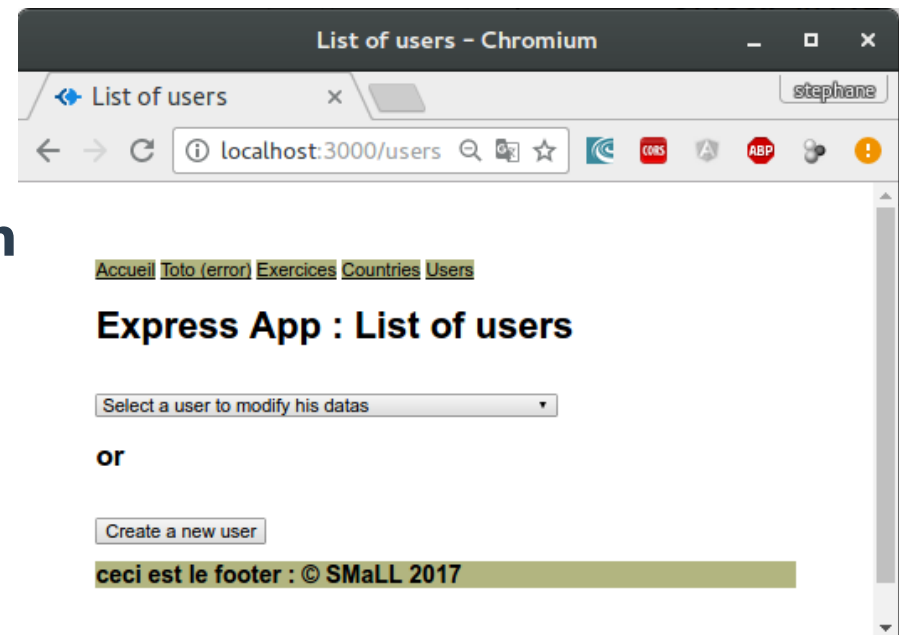
- Nous allons ajouter dans notre partials “**nav.hbs**” le lien permettant d’accéder à cette liste des users :

```
<nav class="barnav">
 <ul id="nav">
 Accueil
 Toto (error)
 Exercices
 Countries
 Users

</nav>
```

- Voici le résultat :

Si vous sélectionnez un user dans la liste le script déclenchera automatiquement un appel à l’action /formUser qui affiche un formulaire avec les données prêtes à être modifiées. Le bouton “Create ...” affiche la vue formUser.hbs vide de données prête pour une saisie d’un nouvel utilisateur.



# Node, Express, MongoDB

- Mettons en oeuvre cette mise à jour dans notre Express App.
  - Il faut initialiser les routes pour chaque action (pathname) dans **app.js** à deux endroits :

```
(...)
var formUser = require('./routes/formUser');
var modifyUser = require('./routes/modifyUser');
(...)
```

- Puis nous allons gérer les routes, qui seront plus complexes que pour une sélection globale :

```
(...)
app.use('/formUser', formUser); // affichera le formulaire
app.use('/modifyUser', modifyUser); // valide les données dans la base
(...)
```

# Node, Express, MongoDB

- Vous allez devoir créer 2 vues et 2 contrôleurs :

## 1- formUser.hbs :

```
<h1> Express App : {{title}}</h1>
<form method="POST" action="{{form_action}}" id="formAddUser">
 <input type="hidden" name="id" value="{{user._id}}" />
 name : <input type="text" name="name" value="{{user.name}}" />

 firstName : <input type="text" name="firstName" value="{{user.firstName}}" />

 login : <input type="text" name="login" value="{{user.login}}" />

 mdp : <input type="password" name="mdp" value="{{user.mdp}}" />

 function : <input type="text" name="function" value="{{user.function}}" />

 office : <input type="text" name="office" value="{{user.office}}" />

 date_naiss : <input type="text" name="date_naiss" value="{{user.date_naiss}}" />

 adressel : <input type="text" name="adressel" value="{{user.adressel}}" />

 adresse2 : <input type="text" name="adresse2" value="{{user.adresse2}}" />

 cp : <input type="text" name="cp" value="{{user.cp}}" />

 city : <input type="text" name="city" value="{{user.city}}" />

 mobile_phone : <input type="text" name="mobile_phone" value="{{user.mobile_phone}}" />

 home_phone : <input type="text" name="home_phone" value="{{user.home_phone}}" />

 <input type="submit" value="Confirmer la {{libelle}}" />
</form>

<!-- Script de validation du formulaire de modification -->
<script>
 var form = document.getElementById('formAddUser');
 form.addEventListener('submit', function(evt) {
 form.action += "/{{user._id}}";
 form.submit();
 });
</script>
```

# Node, Express, MongoDB

## 2- modifyUser.hbs :

```
<h1> Express App : {{title}}</h1>

 name :..... {{user.name}}

 firstName :..... {{user.firstName}}

 login :..... {{user.login}}

 mdp :..... {{user.mdp}}

 function :..... {{user.function}}

 office :..... {{user.office}}

 date_naiss :..... {{user.date_naiss}}

 adresses1 :..... {{user.adresses1}}

 adresse2 :..... {{user.adresse2}}

 cp :..... {{user.cp}}

 mobile_phone :..... {{user.mobile_phone}}

 home_phone :..... {{user.home_phone}}


```

- La première vue affiche un formulaire, qui sera utilisé à la fois pour les modifications d'un utilisateur existant ou pour la création. La seconde vue permet d'afficher les données qui viennent d'être modifiées dans la base de données MongoDB.



# Node, Express, MongoDB

## 3- formUser.js :

```
var express = require('express');
var router = express.Router();
var mongoose = require('mongoose');
var ObjectId = require('mongodb').ObjectId;

/* GET users fromm _id. */
router.route('/:_id').get(function(req, res) {
 console.log('req.originalUrl : ' , req.originalUrl);
 GLOBAL.db.collection('users')
 .find({_id: new ObjectId(req.params._id)})
 .toArray(function(err, result) {
 if (err) { throw err; }
 console.log('formUser: ', result);
 res.render('formUser', {
 title: "Form user\'s datas",
 libelle: "modification",
 form_action: "/modifyUser",
 user: result[0] // il n'y a qu'une réponse possible puisque requête via _id user
 });
 });
});

module.exports = router;
```

**NB :** On remarque dans le paramètre de la fonction route('/:\_id'), il permet de filtrer sur une valeur dans l'URL c'est le principe du protocole REST très utilisé pour développer des APIs online.

# Node, Express, MongoDB

## 4- modifyUser.js :

```
var express = require('express');
var router = express.Router();
var mongoose = require('mongoose');
var ObjectID = require('mongodb').ObjectID;
/* SET user from _id with new data for an update into mongoDB . */
router.route('/:_id').get(function (req, res) {
 console.log('req.originalUrl : ', req.originalUrl);
 GLOBAL.db.collection('users').update(
 {_id: new ObjectID(req.params._id)},
 {$set: req.query},
 function (err, result) {
 if (err) { throw err; }
 console.log('modifyUser: ', result);
 GLOBAL.db.collection('users').find({_id: new new ObjectID(req.params._id)
 }).toArray(function (err, result) {
 if (err) { throw err; }
 console.log('users: ', result);
 res.render('modifyUser', {
 title: 'User modified without error',
 user: result[0]
 });
 }); // fin du find() après update
 } // fin callback de l'update
); // fin de l'update()
}); // fin de la gestion de la route
module.exports = router;
```

**NB :** On remarque le même filtre par l'\_id dans l'url, mais on enchaîne ici un select après l'update sur l'\_id pour réafficher les données modifiées dans la base.

# Node, Express, MongoDB

- Il ne nous reste plus qu'à définir 2 actions pour la création d'un utilisateur :
  - Une pour afficher le formulaire de saisie : **"/newUser"** et l'autre pour l'insertion effective dans la base de données, suite à la validation du formulaire : **"/createUser"** dans **app.js** :

```
(...)
var newUser = require('./routes/newUser');
var createUser = require('./routes/createUser');
(...)
```

- Cela doit devenir pour vous un réflex, il faut donc deux contrôleurs et une seule vue, car on va réutiliser la vue **formUser.hbs** pour la création d'un utilisateur.

# Node, Express, MongoDB

- **newUser.js** : le contrôleur qui appelle le formulaire de saisie **formUser.hbs** :

```
var express = require('express');
var router = express.Router();

/* GET formUser page to insert a new user */
router.get('/', function(req, res, next) {
 res.render('formUser',
 {title: 'Create a new user',
 libelle: "creation",
 form_action: "/createUser"
 });
});
module.exports = router;
```

NB : Vous pouvez remarquer que la fonction **“render”** qui appelle **“formUser”** reçoit également **en paramètre** un objet littéral qui contient **le titre de la page**, le **libellé du bouton** de validation du formulaire et **l’action du formulaire**, on pourrait paramétrer également la méthode (POST/GET) etc ...

Cela nous permet de réutiliser la vue modification pour la saisie.

# Node, Express, MongoDB

- **createUser.js** : est le contrôleur qui va exécuter la fonction d'insertion dans la base de données :

```
var express = require('express');
var router = express.Router();
var mongoose = require('mongoose');
/* Insert one new user into database. */
router.route('/').get(function (req, res) {
 console.log('req.originalUrl : ', req.originalUrl);
 GLOBAL.db.collection('users').insert([req.query],
 function (err, result) {
 if (err) {
 throw err;
 }
 console.log('createUser: ', result);
 res.render('modifyUser', {
 title: 'Creating User without error with datas below :',
 user: result.ops[0]
 });
 } // fin callback de l'insert
); // fin de l'insert()
}); // fin de la gestion de la route
module.exports = router;
```

NB : On remarque la syntaxe de la fonction “**insert**” qui prend comme paramètre un tableau d’objet. Nous lui passons req.query qui contient les données du formulaire au format JSON. Regardez en debug (via node-inspector) la variable **result** ...

# Node, Express, MongoDB

- **Conclusion :**

- Vous avez vu comment à partir **d'Express, MongoDB** et **Handlebars** construire une application Web simple mais qui repose sur une **architecture MVC** qui a fait ses preuves.
- On remarquera que pour d'importantes applications Web, le fait de devoir **ajouter dans app.js** l'ensemble **des routes et des actions** peut vite devenir **difficilement maintenable**.
- Il serait intéressant de **réfléchir** à une **organisation de la gestion des routes** ( ou actions ex. : **"/users"**) plus **générique**.

# Node, Express, MongoDB

## • Exercice :

- Ecrire un chargeur de controleurs génériques permettant de définir les routes (actions) et les contrôleurs dans une description JSON :

```
{
 "GET/" : {"controler": "index"},
 "GET/users": {"controler": "users"},
 "GET/exos": {"controler": "exos"},
 "GET/countries": {"controler": "countries"},
 "GET/formUser": {"controler": "formUser"},
 "GET/modifyUser": {"controler": "modifyUser"},
 "GET/newUser": {"controler": "newUser"},
 "GET/createUser": {"controler": "createUser"}
}
```

- Créez ce fichier “**config\_actions.json**” dans le dossier “**/routes**” de notre application.

# Node, Express, MongoDB

## • Exercice (suite) :

- Afin de garder notre précédente application, nous allons copier “**app.js**” dans “**appdyn.js**”.
- Voici les lignes de code à supprimer dans le nouveau fichier appdyn.js :

```
// Lignes 9 à 20
var index = require('./routes/index');
var users = require('./routes/users');
var exos = require('./routes/exos');
var countries = require('./routes/countries');

var users = require('./routes/users');

var formUser = require('./routes/formUser');
var modifyUser = require('./routes/modifyUser');

var newUser = require('./routes/newUser');
var createUser = require('./routes/createUser');
```

```
// ligne 72 à 83
app.use('/', index);
app.use('/users', users);
app.use('/exos', exos);
app.use('/countries', countries);
app.use('/users', users);
app.use('/formUser', formUser);
app.use('/formUser/:_id', formUser);
app.use('/modifyUser', modifyUser);
app.use('/modifyUser/:_id', modifyUser);
app.use('/newUser', newUser);
app.use('/createUser', createUser);
```

- Nous allons ajouter la lecture de notre configuration JSON des routes :



# Node, Express, MongoDB

## • Exercice correction :

- Nous allons ajouter la lecture de notre configuration JSON des routes à la place de la liste des requires des contrôleurs :

```
/* chargement configuration JSON des actions --> controleurs */
GLOBAL.actions_json = JSON.parse(fs.readFileSync("./routes/config_actions.json", 'utf8'));
```

- “**actions\_json**” est un tableau associatif des actions comme clé et des contrôleurs comme valeurs.
- Nous allons devoir créer un **routeur Dynamique** qui en fonction de la configuration dans le fichier JSON associe le contrôleur à l'action (le pathname). Appelons le “**dynamicRouter.js**” et placez le dans le même dossier que **app.js** et **appdyn.js**

```
// Gestion des routes dynamiques via configuration json
require('./dynamicRouter')(app);
```

# Node, Express, MongoDB

## • Exercice correction (suite) :

### – Comment écrire ce “dynamicRouter.js” ?

```
var express = require("express");
var router = express.Router();
var appContext;
var url = require("url");

function dynamicRouter(app) {
 //-- Context applicatif
 appContext = app;
 // -- Perform Automate action
 router.use(manageAction);
 // -- routeur global
 appContext.use(router);
}
(...) // → suite colonne de droite
```

NB : Pensez à changer dans le fichier /bin/www le nom du fichier app en appdyn

```
var app = require('..../appdyn');
```

```
/* Fonction qui permet d'agguiller les requêtes HTTP
vers le bon contrôleur en fonction de l'action du pathname */
function manageAction(req, res, next) {
 var path; // Le pathname après le port 3000 dans l'URL.
 var type; //(GET ou POST, ... http méthode)
 var controler; // nom du contrôleur à charger
 path = url.parse(req.url).pathname;
 // Il faut supprimer pour le routage le param après l'action
 if (path.split('/').length > 0) path = '/' + path.split('/')[1]
 type = req.method;
 if (typeof GLOBAL.actions_json[type + path] == 'undefined') {
 console.log("Erreur pas d'action : " + path);
 next();
 }
 else {
 instanceModule = require('./routes/'
 + GLOBAL.actions_json[type + path].controler);
 router.use(path, instanceModule);
 next();
 }
}

module.exports = dynamicRouter;
```

## • Conclusion Routeur Dynamique

- Nous n'avons volontairement géré que le nom du contrôleur dans notre fichier de configuration JSON. Pour simplifier l'exemple.
- Mais nous pourrions ajouter des informations, comme la vue, le type de retour (html, json), des paramètres supplémentaires, ...
- Maintenant dans notre application Express, MongoDB, Handlebars il n'est plus nécessaire d'ajouter du code dans appdyn.js pour ajouter une action.
- Il suffit de compléter et d'ajouter dans le fichier config\_actions.json une action et un contrôleur.

# Node, Express, MongoDB

- **Mongoose : Un outil pour “schématiser” l'accès à la base de données**
    - **Mongoose** est une librairie qui permet de créer des abstractions objets sur les bases de données MongoDB. (hibernate en Java)
    - Pour l'installer, il faut ajouter le module mongoose :
- ```
$ npm install mongoose --save
```
- Dans notre application nous pouvons instanciés **“Mongoose”**.
 - C'est lui qui va gérer la connexion à la base MongoDB.

Node, Express, MongoDB

- Voyons un extrait de code test initialisant une connexion à MongoDB via Mongoose :

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://127.0.0.1:27017/gretajs', function (err) {
  if (err) {
    throw err;
  } else console.log('Connected');
});
// Schéma définissant une collection
var countriesSchema = new mongoose.Schema({
  _id: {type : mongoose.Schema.ObjectId},
  code: {type: String},
  name: {type: String}
});
// Association entre le schéma et la collection retourne un Model Mongoose
var collection = mongoose.model('Countries', countriesSchema, 'countries');
collection.find(function (err, comms) {
  if (err) {
    throw err;
  }
  console.log(comms);
  mongoose.connection.close();
});
```

Node, Express, MongoDB

- **Au vu du code de connexion avec Mongoose, il semble possible de créer des schémas au format Json :**
 - En effet il est possible de décrire un schéma de mongoose au format JSON dans un fichier texte (.json) plutôt que dans du code JavaScript.
 - Reste à concevoir un module générique qui va initialiser les Schémas au démarrage de l'app en lisant le fichier database_schema.json.

NB : Ce module existe dans OTF² :

```
/otf_core/lib/otf_schema_loader.js
```

Node, Express, MongoDB

- **Implémentation d'un module de chargement de schéma Mongoose.**
 - Comme nous avons chargé les actions → contrôleurs pour le **dynamicRouter**, nous allons charger le schéma mongoose des collections.
 - Créez un fichier “**database_schema.json**” à la racine du projet sur ce modèle :

```
{  
  "Countries" : {  
    Collection: "countries",  
    Schema: {  
      _id: "ObjectId",  
      code: "String",  
      name: "String"  
    }  
  }, (...)  
}
```

Node, Express, MongoDB

- Exemple de fichier “database_schemas.json” :

```
{
  "Exercices" : {
    "collection": "exercices",
    "schema": {
      "_id": "ObjectId",
      "titre": "String",
      "liste": []
    }
  },
  "Countries" : {
    "collection": "countries",
    "schema": {
      "_id": "ObjectId",
      "code": "String",
      "name": "String"
    }
  },
}
```

(...) // → suite colonne de droite

```
  "Users": {
    "collection": "users",
    "schema": {
      "_id": "ObjectId",
      "name": "String",
      "firstName": "String",
      "login": "String",
      "mdp": "String",
      "function": "String",
      "office": "String",
      "date_naiss": "String",
      "adresse1": "String",
      "adresse2": "String",
      "cp": "String",
      "city": "String",
      "country": {
        "type": "ObjectId",
        "ref": "Countries"
      },
      "mobile_phone": "String",
      "home_phone": "String"
    }
  }
}
```


Node, Express, MongoDB

- Nous allons tester le chargement et l'utilisation de nos schémas dans le fichier "testmongoose2.js" :

```
var fs = require('fs');
GLOBAL.schemas = {};
var mongoose = require('mongoose');
mongoose.connect('mongodb://127.0.0.1:27017/gretajs', function (err) {
  if (err) {
    throw err;
  } else console.log('Connected');
});
// chargement des schémas depuis le fichier de configuration JSON dans une variable
var database_schemas = JSON.parse(fs.readFileSync("database_schemas.json", 'utf8'));
// Initialisation de chaque schéma par association entre le schéma et la collection
for (modelName in database_schemas) {
  GLOBAL.schemas[modelName] = mongoose.model(modelName, database_schemas[modelName].schema,
    database_schemas[modelName].collection);
}

/* On obtient un tableau de Models à partir des schémas accessible via
 * la variable GLOBAL.schemas qui permettent d'exécuter des requêtes.*/
GLOBAL.schemas["Countries"].find({code : "FR"}, function (err, comms) {
  if (err) { throw err; }
  // comms est un tableau de hash
  console.log(comms);
  mongoose.connection.close();
});
```

Node, Express, MongoDB

- **Exercice d'intégration de Mongoose :**
 - Modifiez le code de connexion à la base de données dans votre application Express App et utilisez Mongoose
 - Puis intégrez le chargeur de schéma que nous avons testé dans “**testmongoose2.js**” dans l'application, dans “**appdyn.js**”
 - Enfin modifiez les contrôleurs pour qu'ils réalisent les requêtes via les Models Mongoose `find()`, `create()` et `update()` et `remove()`.

Node, Express, MongoDB

• Exercice correction intégration de Mongoose :

- Dans “**appdyn.js**” nous allons modifier la connexion à la base de données :

```
GLOBAL.schemas = {};  
  
// Configuration de la connexion à la base de données via Mongoose :  
var mongoose = require('mongoose');  
mongoose.connect('mongodb://127.0.0.1:27017/gretajs', function (err) {  
    if (err) {throw err;} else console.log('Connected');  
});  
// chargement des schémas depuis le fichier de configuration database_schemas.json  
var database_schemas = JSON.parse(fs.readFileSync("database_schemas.json", 'utf8'));  
  
for (modelName in database_schemas) {  
    GLOBAL.schemas[modelName] = mongoose.model(modelName,  
        database_schemas[modelName].schema,  
        database_schemas[modelName].collection);  
}  
(...)
```

NB : pensez à supprimer en bas du fichier “appdyn.js” la connexion via le driver natif mondodb.

Node, Express, MongoDB

• Exercice correction intégration de Mongoose :

- Voyons la modification des contrôleurs pour utiliser **GLOBAL.schemas[“<NOM_SCHEMA>”]** exemple avec “**countries.js**” :

```
var express = require('express');
var router = express.Router();

/* GET list of countries */
router.get('/', function(req, res, next) {
  GLOBAL.schemas["Countries"].find({}, function(err, result) {
    if (err) {
      throw err;
    }
    //console.log(result);
    res.render('countries', {title: 'Liste countries', country: result});
  });
});
module.exports = router;
```

NB : on utilise la méthode `find` avec un objet vide pour récupérer l'ensemble des données de la collection `countries`.

Node, Express, MongoDB

• Exercice correction intégration de Mongoose : “createUser.js” :

```
var express = require('express');
var router = express.Router();
var mongoose = require('mongoose');
var ObjectId = mongoose.Types.ObjectId;
/* Insert one new user into database. */
router.route('/').get(function (req, res) {
  console.log('req.originalUrl : ', req.originalUrl);
  if (!req.query.hasOwnProperty("_id")) req.query._id = new ObjectId();
  GLOBAL.schemas["Users"].create([req.query], function (err, result) {
    if (err) { throw err; }
    console.log('createUser: ', result);
    res.render('modifyUser', {
      title: 'Creating User without error with datas below :',
      user: result[0]._doc
    });
  } // fin callback de l'insert
); // fin de l'insert()
}); // fin de la gestion de la route

module.exports = router;
```

NB : On remarque l'utilisation de `mongoose.Type.ObjectId` pour créer un id pour l'enregistrement à insérer. On l'ajoute à `req.query` que l'on passe en paramètre de la méthode “`create([req.query],...`”

Node, Express, MongoDB

• Exercice correction intégration de Mongoose : “formUser.js” :

```
var express = require('express');
var router = express.Router();
var mongoose = require('mongoose');
var ObjectId = mongoose.Types.ObjectId;

/* GET user from _id into url */
router.route('/:_id').get(function (req, res) {
  GLOBAL.schemas["Users"].find({_id: new ObjectId(req.params._id)}, function (err, result) {
    if (err) { throw err; }
    console.log('formUser: ', result);
    res.render('formUser', {
      title: "Form user\'s datas",
      libelle: "modification",
      form_action: "/modifyUser",
      user: result[0]
    });
  });
});
module.exports = router;
```

NB : On remarquera la création d'un ObjectId à partir de la chaîne de caractère `_id` récupéré dans la liste déroulante des utilisateurs.

Node, Express, MongoDB

• Conclusion intégration Mongoose :

- Intégration de Mongoose afin de rendre plus souple et plus dynamique l'intégration de collections dans notre application.
- L'ajout des configurations JSON décrivant les schémas qui permettront de construire la base de données et les actions permettent plus de souplesse dans la maintenance et les développements futurs.
- Vous pourriez ajouter des paramètres dans le fichier "**config_actions.json**" pour rendre générique les accès à la base de données.

Node, Express, MongoDB

- **Exercice : Reflexion d'architecture logicielle**

- Avec l'intégration de Mongoose et des schémas décrit dans un fichier de configuration JSON on peut modifier le code et on obtient pour un find() un code ressemblant a ceci, illustration avec le contrôleur exos.js :

```
var express = require('express');
var router = express.Router();

/* GET Exercices list. */
router.get('/', function(req, res, next) {
  var type = req.method;
  var path = req.originalUrl;

  GLOBAL.schemas['Exercices'].find({}, function(err, result) {
    if (err) { throw err; }
    console.log(result);
    res.render('exos', {title: 'Express', exos: result[0]});
  });
});
module.exports = router;
```

NB : Trouver une structure JSON qui permet de paramétrer le nom du “Model” (au sens Mongoose) pour l’action considérée

Node, Express, MongoDB

- **Correction : Reflexion d'architecture logicielle**
Exemple structure JSON :

```
(...) },  
  "GET/exos": {  
    "controler": "exos",  
    "modelName": "Exercices",  
    "view": "exos"  
  },  
(...)
```

- **Ci-dessous le code modifié du contrôleur exos.js**

```
var express = require('express');  
var router = express.Router();  
  
/* GET Exercices list. */  
router.get('/', function(req, res, next) {  
  var type = req.method;  
  var path = req.originalUrl;  
  //if (path.split('/').length > 0) path = '/' + path.split('/')[1]  
  GLOBAL.schemas[GLOBAL.actions_json[type + path].modelName].find({}, function(err, result) {  
    if (err) {  
      throw err;  
    }  
    console.log(result);  
    res.render(GLOBAL.actions_json[type + path].view, {title: 'Express', exos: result[0]});  
  });  
});  
module.exports = router;
```

Node, Express, MongoDB

• Exercice Correction :

- Nous avons pris le parti de modifier la ve **“users.hbs”** et d'ajouter une dèxième liste déroulante puisque on a déjà les données.
- Les modification seront faite sur le projet :

MyExpressHbsAppMongoose

- Nous devons ajouter une action dans notre fichier config_actions.json :

```
(...) },  
  "GET/deleteUser": {  
    "controler": "deleteUser"  
  }  
}
```

Node, Express, MongoDB

• Exercice Correction (suite) :

- Le code source du contrôleur “**deleteUser.js**” :

```
var express = require('express');
var router = express.Router();
var mongoose = require('mongoose');
var ObjectId = mongoose.Types.ObjectId;

/* DELETE user from _id into url */
router.route('/:_id').get(function (req, res) {
  console.log('req.originalUrl : ', req.originalUrl);
  GLOBAL.schemas["Users"].remove({
    _id: new ObjectId(req.params._id)
  }, function (err, result) {
    if (err) { throw err; }
    GLOBAL.schemas["Users"].find({}, function (err, result) {
      res.render('users', {
        title: "List of users",
        users: result
      });
    });
  });
});

module.exports = router;
```

Node, Express, MongoDB

• Exercice Correction (suite) :

- Le code source à ajouter à la vue “**users.js**” pour pouvoir déclencher l’action “**/deleteUser**”

```
(...)  
<h2>or</h2><br/>  
<select name="user_id" id="listusers">  
  {{#compare users null operator="tabempty"}}  
    <option value="rien">Il n'y a pas de données dans la base users</option>  
  {{else}}  
    <option value="0" >Select a user to delete its record</option>  
    {{#each users}}  
      <option value="{{this._id}}" >{{this.name}} {{this.firstName}} -  
                                                {{this.function}}</option>  
    {{/each}}  
  {{/compare}}  
</select>  
<script>  
(...)  
var usrDel = document.getElementById('listusers');  
usrDel.addEventListener('change', function(evt) {  
  window.location = "/deleteUser/"+usrDel.value;  
});  
</script>
```

Node, Express, MongoDB

- **Le module Passport pour l'authentification**
 - Nous avons presque fini notre template d'application utilisant :
 - Node.js avec comme modules principaux :
 - Express
 - Handlebars
 - MongoDB
 - Mongoose
 - Il nous reste à gérer **l'authentification** d'un utilisateur, pour permettre l'accès à un espace privé sur notre application, avec PassportJS.
 - Pour installer PassportJS et les dépendances :

```
$ npm install passport passport-local express-session --save
```

Node, Express, MongoDB

- **Le module Passport pour l'authentification**

- Résultat dans le terminal :

```
stephane@UX303UB:~/workspaceGreta/gretaexos2$ npm install passport passport-local express-session --save
myexpresshbsapp@0.0.0 /home/stephane/workspaceGreta/gretaexos2
├─ express-session@1.15.1
├─ passport@0.3.2
└─ passport-local@1.0.0

stephane@UX303UB:~/workspaceGreta/gretaexos2$
```

- De la documentation sur passport :
 - <http://passportjs.org/>
 - Article sur SupInfo : Authentification avec passport
 - Article sur JWTToken et la sécurisation d'API
 - Article SupInfo sur : JWTToken & passportJS
- **Exercice** : Essayer de mettre en oeuvre passport dans votre application, de façon simple gérez la route **app.post('/authenticated', ...)** dans **appyn.js**. (50mn)

Node, Express, MongoDB

- **Le module Passport pour l'authentification**
 - Nous allons maintenant intégrer les modules à notre App Express, dans **appdyn.js** :

```
(...)  
var fs = require('fs');  
var session = require('express-session');  
var passport = require("passport");  
var LocalStrategy = require('passport-local').Strategy;  
(...)
```

- Dans la liste des middlewares, il faut ajouter en suivant ceux concernant passport et la session :

```
(...)  
app.use(express.bodyParser());  
app.use(session({ cookieName: 'sessiongreta', secret: 'AsipfGjdp*%dsDKNFNFKqoeID1345' }));  
app.use(passport.initialize());  
app.use(passport.session());  
(...)
```

Node, Express, MongoDB

• Le module Passport pour l'authentification

- Nous avons déjà une collection qui gère le login et le mot de passe (**users**) et son schéma correspondant "**Users**" dans le fichier **database_schemas.json**.
- Nous devons donc maintenant définir notre **stratégie d'authentification**, nous allons faire une authentification locale c'est à dire dans notre base de données MongoDB.
- Avant tout, toujours dans **appdyn.js** nous allons définir 2 méthodes callback pour: **serializeUser** et **deserializeUser** permettant de stocker en session un user via son user.id

```
(...)  
app.use(passport.session());  
passport.serializeUser(function(user, done) {  
  done(null, user.id);  
});  
  
passport.deserializeUser(function(id, done) {  
  GLOBAL.schemas["Users"].findById(id, function(err, user) {  
    done(err, user);  
  });  
});  
(...)
```


Node, Express, MongoDB

• Le module Passport pour l'authentification

- Nous devons créer la **stratégie** utilisée par **passport** pour l'authentification dans notre exemple un **username** et un **password** (login et mdp) :

```
passport.deserializeUser(function(id, done) {
  GLOBAL.schemas["Users"].findById(id, function(err, user) {
    done(err, user);
  });
});

passport.use(new LocalStrategy(
  function (username, password, done) {
    GLOBAL.schemas["Users"].findOne({ login: username }, function (err, user) {
      if (err) { return done(err); }
      if (!user) {
        console.log("pas d'utilisateur trouvé");
        return done(null, false, { message: 'Incorrect username.' });
      }
      if (user.mdp !== password) {
        console.log("password erroné");
        return done(null, false, { message: 'Incorrect password.' });
      }
      console.log("utilisateur : ", user);
      return done(null, user);
    });
  })
);
(...)
```

Node, Express, MongoDB

• Le module Passport pour l'authentification

- Il faut maintenant gérer la route qui va demander l'authentification, prenons par exemple le pathname :

`/authenticated`

- Lorsque un client demandera cette route via un POST http, nous devons appeler la méthode :

`passport.authenticate('local')`

- Dans la fonction de callback de de la gestion de cette route nous devons vérifier si le user est ajouté dans la session par passport, toujours dans **appdyn.js** :

```
app.post('/authenticated', passport.authenticate('local'), function (req, res) {  
  if (req.session.passport.user != null) {  
    res.redirect('/index'); //le user est authentifié on affiche l'index il est en session  
  } else {  
    res.redirect('/'); // il n'est pas présent on renvoie à la boîte de login  
  }  
});  
// Routes Managed dynamically  
require('./dynamicRouter')(app);
```

Node, Express, MongoDB

• Le module Passport pour l'authentification

- Nous avons quelques modification du fichier de configuration des actions. Nous devons ajouter une action et modifier la première pour afficher une boîte de login sur '/' :

```
(...)  
  "GET/" : {  
    "controller": "login"  
  },  
  "GET/index" : {  
    "controller": "index"  
  },  
(...)
```

- Voici le code du controleur “**login**” qui ne fait qu’afficher la page de login via login.hbs :

```
var express = require('express');  
var router = express.Router();  
  
/* GET home page. */  
router.get('/', function(req, res, next) {  
  res.render('login', { title: 'Express authentication' });  
});  
module.exports = router;
```

Node, Express, MongoDB

• Le module Passport pour l'authentification

- Il nous reste la vue à configurer pour pouvoir tester l'authentification, voici login.hbs à écrire :

```
<h1>{{title}}</h1>
<form action="/authenticated" method="POST">
  <div>
    <label>Username:</label>
    <input type="text" name="username"/>
  </div>
  <br/>
  <div>
    <label>Password:</label>
    <input type="password" name="password"/>
  </div>
  <div>
    <input type="submit" value="Log In"/>
  </div>
</form>
```

- Mais il nous manque quelque chose d'important, il faut que chaque contrôleur qui nécessite une authentification vérifie avant de réaliser le traitement que la requête est authentifiée.

Node, Express, MongoDB

- **Le module Passport pour l'authentification**
 - Vérifier la présence dans la session d'un user, donc qui a passé l'étape d'authentification. Prenons le module **routes/countries.js** :

```
var express = require('express');
var router = express.Router();

/* GET countries page. */
router.get('/', function (req, res, next) {
  if ((req.session.passport) && (req.session.passport.user != null)) {
    GLOBAL.schemas["Countries"].find({}, function (err, result) {
      if (err) { throw err; }
      res.render('countries', {
        title: 'Liste countries',
        country: result
      });
    });
  } else res.redirect('/'); // affichage boîte de login si pas authentifié
});
module.exports = router;
```

NB : Il vous reste à modifier l'ensemble des controleurs qui nécessitent une authentification: exos, users, createUser, modifyUser, formUser, newUser. Ne pas modifier index et login !

Node, Express, MongoDB

- **Le module Passport pour l'authentification**
 - Exécuté et debugguez votre code.
 - **Exercice** : Vous ajouterez la possibilité de se déconnecter via une action **“/logout”** qui doit annuler l'accès aux fonctionnalités précédemment définies.
 - Pour ce faire vous ajouterez un lien dans :
`views/partials/nav.hbs`
 - Vous ajouterez une entrée dans le fichier :
`config_actions.json`

```
    },  
    "GET/logout": {  
      "controller": "logout"  
    }  
  }  
}
```

Node, Express, MongoDB

- **Le module Passport pour l'authentification**
 - **Correction Exercice** : Voyon le code de notre contrôleur de déconnexion :

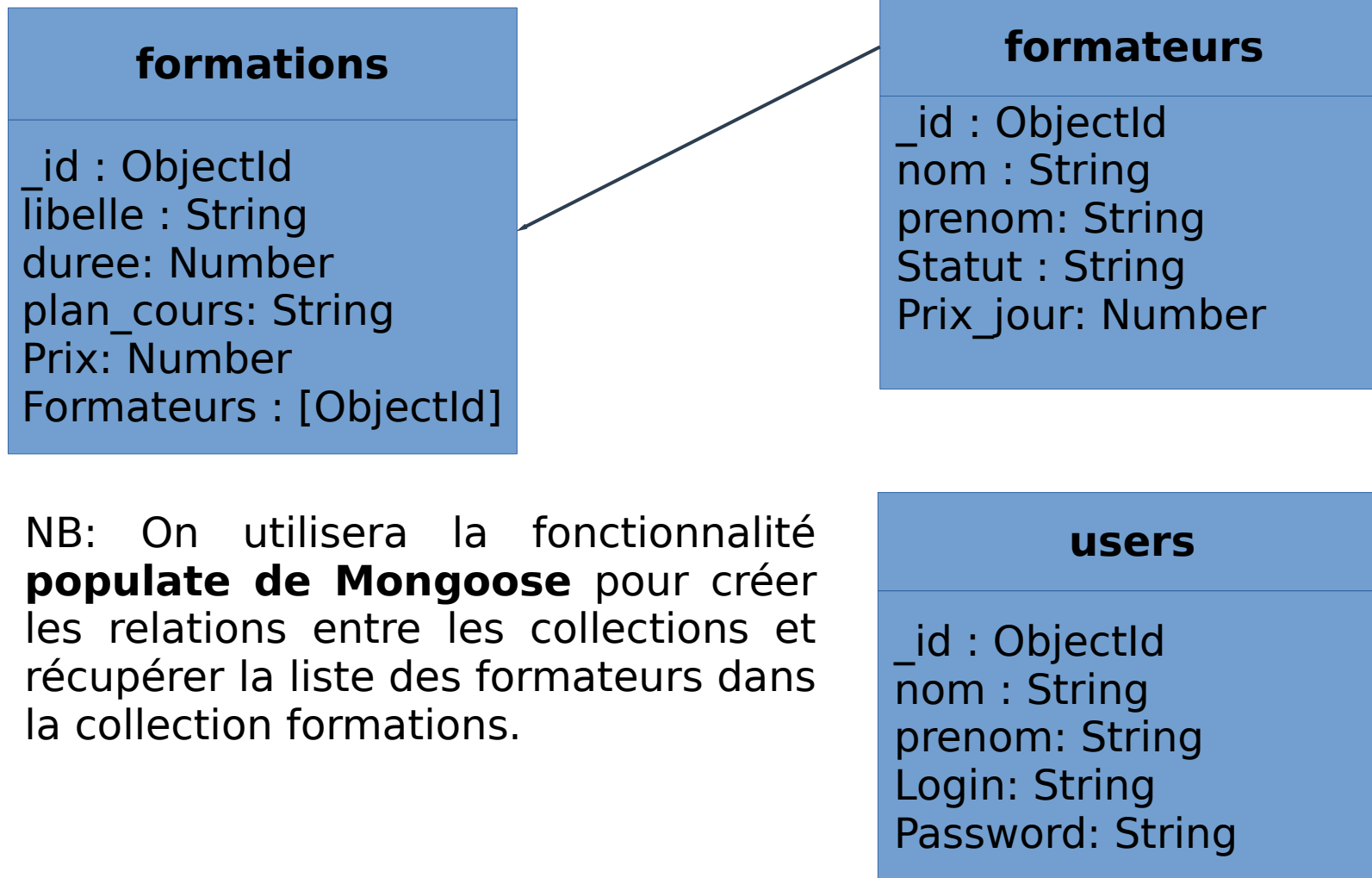
```
var express = require('express');
var router = express.Router();
/* set logout action and redirect to login page. */
router.get('/', function(req, res, next) {
    if ((req.session.passport) && (req.session.passport.user != null)) {
        req.logout(); // efface de la session.passport la propriété user
        res.redirect('/');
    } else res.redirect('/');
});
module.exports = router;
```

Node, Express, MongoDB

- **TD : Réaliser une application Web Node.js**
 - **Vous allez devoir écrire une application web qui permet à un utilisateur authentifié de saisir des formations.**
 - **Une formation se caractérise par :**
 - Un nom : String,
 - Une durée en jours : Number,
 - Un plan de cours résumé : String
 - Un prix en € : Number
 - Des formateurs (on utilisera une collection différente et on réalisera un lien entre les 2 collections “formations” et “formateurs” donc un tableau d’ObjectId dans formations.
 - On pourra, sans être authentifié, visualiser la liste des formations et cliquer sur l’une d’elle pour voir le plan de cours, le prix et les formateurs qui peuvent la dispenser.

Node, Express, MongoDB

- TD : Réaliser une application Web Node.js
 - Schémas de la base de données :



Node, Express, MongoDB

- **TD : Réaliser une application Web Node.js :**
 - En utilisant les cours et l'application Express App que nous avons créée durant le cours, vous réaliserez une application web modulaire, c'est à dire que les traitements sont réalisés par des modules "contrôleurs" et non dans app.js.
 - Vous intégrerez les notions vues dans les cours Système et Web Design en intégrant :
 - Bootstrap : pour le style de l'interface Web
 - Git : en créant un dépôt sur github.com pour votre projet.
 - Bon développement !


Node, Express, MongoDB et Bases SQL

• Connectivité SQL : Introduction.

- Il est possible de se connecter à des bases de données SQL en utilisant les pilotes natifs ou JavaScript comme pour **postgreSQL** ou **MySQL**, **MariaDB** **sqlite** :
 - **node-postgres**
 - **pg**
 - **pg-native**
 - **node-mysql**
 - **node-sqlite (sqlite)**
 - ...
- Nous utiliserons ensuite un ORM qui permet de se connecter à l'ensemble de ces bases de données citées plus haut.

Node, Express, MongoDB et Bases SQL

- **Ajouter une table dans notre base de données : la table “countries”**
 - dont les données en csv sont récupérable ici :
 - <http://blog.plsoucy.com/wp-content/uploads/2012/04/countries-20140629.csv>
 - On crée les champs “code”, “libelle_us” et “libelle_fr” dans la table countries via PgAdminIII :



The screenshot displays the PgAdminIII interface. On the left, the 'Tables (1)' folder is expanded, showing the 'countries' table. Under 'Colonnes (3)', the columns 'code', 'libelle_us', and 'libelle_fr' are listed. Below this, 'Contraintes (0)' and 'Index (0)' are shown. On the right, the data of the 'countries' table is displayed in a grid. The table has 100 rows, with the first 8 rows visible. Each row contains an index number, a two-letter country code, the country name in English, and the country name in French.

17	BA	Bosnia and Herzegovina	Bosnie-Herzégovine
18	BB	Barbados	Barbade
19	BD	Bangladesh	Bangladesh
20	BE	Belgium	Belgique
21	BF	Burkina Faso	Burkina Faso
22	BG	Bulgaria	Bulgarie
23	BH	Bahrain	Bahreïn
24	RT	Rurundi	Rurundi

100 lignes.

Node, Express, MongoDB, et Baseq SQL

- **Pilote PostgreSQL : le module “pg”**

- Installez le module via :

```
$ npm install --save pg
```

- Nous allons créer un fichier nommé “**testPostgreSQL.js**” à la racine de notre projet puis l’exécuter dans un terminal :
node ./testPostgreSQL.js

```
var postgres = require('pg');
var config = { user: 'steph', database: 'gretajs', password: 'azerty', host: 'localhost',
               port: 5432, max: 10, idleTimeoutMillis: 30000
};
var client = new postgres.Client(config);
// connexion à la base de données
client.connect(function(err) {
  if (err) throw err;
  // execute a query on our database
  client.query('SELECT * from countries', [], function(err, result) {
    if (err) throw err;
    // afficher le résultat dans la console
    console.log(result); // outputs: { name: 'brianc' }
    // déconnexion du client
    client.end(function(err) { if (err) throw err; });
  });
});
```

Node, Express, MongoDB, et Baseq SQL

- Créer un pool de connexion : créer un autre fichier “testpostgreSQLPool.js” :

```
var postgres = require('pg');
var config = { user: 'steph', database: 'gretajs', password: 'azerty', host: 'localhost',
               Port: 5432, max: 10, idleTimeoutMillis: 30000
};
var pool = new postgres.Pool(config);
// connect to our database
pool.connect(function(err, client, done) {
  if (err) {
    return console.error('error pour récupérer une cnx cliente dans le pool', err);
  }
  // execute a query on our database
  client.query('SELECT * from countries', [], function(err, result) {
    done(err);
    if (err) {
      return console.error('error d\'exécution de la requête postgreSQL query', err);
    }
    // just print the result to the console
    console.log(result); // outputs: { name: 'brianc' }
    // disconnect the client
    client.end(function(err) { if (err) throw err; });
  });
});
```

NB : on peut utiliser des variables d'environnement pour la connexion à la base cf. [Github du module node-postgres](#).

Node, Express, MongoDB et Bases SQL

• Connectivité SQL via ODM : Sequelize

- Vous trouverez de la documentation sur le site officiel : <http://sequelize.readthedocs.io/en/latest/>
- L'installation se fait via npm :

```
$ npm install --save sequelize
// Puis au choix en fonction de la base sur laquelle vous devez vous connecter
$ npm install --save pg pg-hstore
$ npm install --save mysql2
$ npm install --save sqlite3
$ npm install --save tedious // MSSQL
```

- Configurer une connexion :

```
var sequelize = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: 'mysql' | 'sqlite' | 'postgres' | 'mssql',

  pool: {
    max: 5,
    min: 0,
    idle: 10000
  });
```

Node, Express, MongoDB et Bases SQL

• Connectivité SQL : Sequelize

- Nous allons créer un fichier **testSequelize.js** à la racine de notre projet :

```
var Sequelize = require ("sequelize");

var sequelize = new Sequelize('gretajs', 'steph', 'azerty', {
  host: 'localhost',
  dialect: 'postgres',

  pool: {
    max: 5,
    min: 0,
    idle: 10000
  }
});

sequelize // la syntaxe est celle d'une Promise
  .authenticate()
  .then(function(err) {
    console.log('Connection has been established successfully.');
```

```
  })
  .catch(function (err) {
    console.log('Unable to connect to the database:', err);
  });
```

NB : exécutez ce fichier dans un terminal :

\$ node ./testSequelize.js *// vous devez voir le log ...**successfully***

Node, Express, MongoDB et Bases SQL

- Nous allons réaliser une requête SQL simple sur notre table countries :
 - Ajoutez ce code pour exécuter une requête SQL :

```
(...)  
// Connexion effective à la base de données via la méthode authenticate()  
// qui retourne une Promise (.then, .catch)  
sequelize  
  .authenticate()  
  .then(function(err) {  
    console.log('Connection has been established successfully.');    // Requête SQL via l'instance sequelize  
    sequelize.query("SELECT * FROM countries", {  
      type: sequelize.QueryTypes.SELECT  
    })  
    .then(function(countries) {  
      console.log('listes des pays : ', countries);  
    })  
    .catch(function(err) {  
      console.log('error select', err);  
    });  
  })  
  .catch(function(err) {  
    console.log('Unable to connect to the database:', err);  
  });
```

Node, Express, MongoDB et Bases SQL

- Nous allons ajouter la capacité de connexion à la base PostgreSQL pour notre Express App:

- Pour ce faire nous avons déjà intégré les modules via :

npm install --save sequelize pg pg-hstore.

- Il nous faut donc juste créer la configuration Sequelize dans **appdyn.js** (avant cnx mongoose) et définir la config sequelize en GLOBAL pour pouvoir y accéder dans nos modules:

```
var Sequelize = require("sequelize");  
// configuration des paramètres de la connexion  
GLOBAL.sequelize = new Sequelize('gretajs', 'steph', 'azerty', {  
  host: 'localhost',  
  dialect: 'postgres',  
  Pool: { max: 5, min: 0, idle: 10000 }  
});
```

Node, Express, MongoDB et Bases SQL

• Le code du contrôleur : countriesSQL.js

```
var express = require('express');
var router = express.Router();

/* GET countriesSQL page. */
router.get('/', function(req, res, next) {
  if ((req.session.passport) && (req.session.passport.user != null)) {
    // Requête SQL via l'instance sequelize
    GLOBAL.sequelize.query("SELECT * FROM countries", {
      type: sequelize.QueryTypes.SELECT
    }).then(function(countries) { // sql query success
      console.log('listes des pays : ', countries);
      res.render('countriesSQL', {
        title: 'List countries from SQL postgreSQL',
        country: countries
      });
    }).catch(function(err) { // sql query error
      console.log('error select', err);
    });
  } else res.redirect('/');
});
module.exports = router;
```

- A partir d'une structure classique de contrôleur pour notre application ajoutez le code en bleu.

Node, Express, MongoDB et Bases SQL

- Avant de pouvoir tester il faut ajouter la route dans notre config_actions.json :

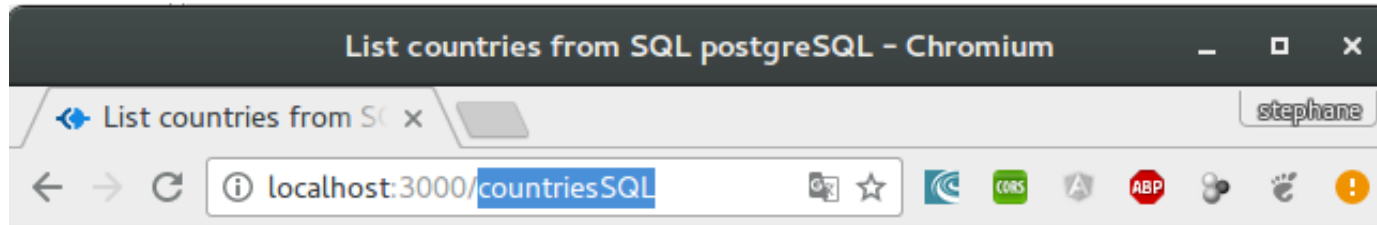
```
(...)  
"GET/logout": {  
  "controler": "logout"  
},  
"GET/countriesSQL": {  
  "controler": "countriesSQL"  
}  
}
```

- Puis dans notre barre de navigation des menus “nav.hbs” nous devons ajouter le lien pour exécuter l’action “/countriesSQL” :

```
<nav class="barnav">  
  <ul id="nav">  
    <li><a href="/index">Accueil</a></li>  
    <li><a href="/toto">Toto (error)</a></li>  
    <li><a href="/exos">Exercices</a></li>  
    <li><a href="/countries">Countries</a></li>  
    <li><a href="/countriesSQL">CountryeSQL</li>  
    <li><a href="/users">Users</a></li>  
    <li><a href="/logout">Déconnexion</a></li>  
  </ul>  
</nav>
```

Node, Express, MongoDB et Bases SQL

- Voyons le résultat à l'exécution de notre App Express :



[Accueil](#) [Toto \(error\)](#) [Exercices](#) [Countries](#) [CountrySQL Users](#) [Déconnexion](#)

Express App : List countries from SQL postgreSQL

France ▼

ceci est le footer : © SMaLL 2017

Node, Express, MongoDB et Bases SQL

- **Architecture logicielle :**

- Nous allons rendre générique le contrôleur qui permet de réaliser les requêtes SQL SELECT :
- Pour ce faire nous devons modifier le code précédent, et passer en paramètres les chaînes de caractères nécessaires à l'exécution d'une requête :
 - **Le nom du contrôleur** est déjà paramétré via le fichier config_actions.json
 - **La requête SQL** elle aussi est à passer en paramètre
 - **Le nom de la vue** (view handlebars) aussi doit être paramétrée.
- La question suivante c'est où que nous devons définir ces 2 paramètres supplémentaires, la réponse est assez évidente dans le fichier : **config_actions.json**
- Nous appellerons "**selectAllSQL.js**" notre contrôleur.

Node, Express, MongoDB et Bases SQL

- **Architecture logicielle (suite) :**

- Voyons comment modifier notre fichier d'actions :

```
(...)  
"GET/logout": {  
  "controler": "logout"  
},  
"GET/countriesSQL": {  
  "controler": "selectAllSQL",  
  "view": "countriesSQL",  
  "sql_query": "SELECT * FROM countries"  
},  
}
```

- Nous allons devoir récupérer ces variables “**view**” et “**sql_query**”, comme on les charge déjà pour récupérer le controler, nous pouvons les ajouter à l'objet requête “**req**”, nous allons le faire dans le **dynamicRouter.js** :
 - En ajoutant une propriété “**message**” à “**req**”
 - En ajoutant l'**action** au message (= **type+path**) :
 - En ajoutant “**view**” et “**sql_query**” dans le message.

Node, Express, MongoDB et Bases SQL

• Voici le code modifié du dynamicRouteur :

```
(...)  
path = url.parse(req.url).pathname;  
type = req.method;  
// Il faut supprimer pour le routage le param après l'action  
if (path.split('/').length > 0) path = '/' + path.split('/')[1]  
// configuration du message pour les contrôleurs génériques  
req.message = {};  
req.message.action = type + path;  
if (GLOBAL.actions_json[type + path].view)  
    req.message.view = GLOBAL.actions_json[type + path].view;  
else  
    req.message.view = null;  
if (GLOBAL.actions_json[type + path].sql_query)  
    req.message.sql_query = GLOBAL.actions_json[type + path].sql_query;  
else  
    req.message.sql_query = null;  
if (typeof GLOBAL.actions_json[type + path] == 'undefined') {  
    console.log("Erreur pas d'action " + path + " dans l'annuaire");  
    next();  
} else {  
    instanceModule = require('./routes/' + GLOBAL.actions_json[type + path].controler);  
    router.use(path, instanceModule);  
    next();  
}  
}  
module.exports = dynamicRouter;
```

NB : Mettez en pratique cette factorisation de paramètres.

Node, Express, MongoDB et Bases SQL

- **Exercice : dupliquez le contrôleur `countriesSQL.js` en le nommant `selectAllSQL.js`**
 - Vous devez remplacer les chaînes de caractères de la requête SQL et du nom de la vue par leur équivalent dans **`req.message`** ajouté par le **`dynamicRouteur`**.
 - Lancer le serveur et exécutez le code.
 - Pour des raisons de compréhension fonctionnelle de votre code vous devrez changer le noms de certaines variables en les rendant plus génériques.

Node, Express, MongoDB et Bases SQL

- **Correction de l'exercice :**
 - Le contrôleur “**selectAllSQL.js**” :

```
var express = require('express');
var router = express.Router();

/* GET database datas from SQL query. */
router.get('/', function(req, res, next) {
  if ((req.session.passport) && (req.session.passport.user != null)) {
    GLOBAL.sequelize.query(req.message.sql_query, {
      type: sequelize.QueryTypes.SELECT
    }) // SQL query success return datas into callback
    .then(function(datas) {
      console.log('listes des datas : ', datas);
      res.render(req.message.view, {
        title: 'List from SQL postgreSQL',
        result: datas
      });
    }) // SQL query error return error into callback
    .catch(function(err) {
      console.log('error select', err);
    });
  } else res.redirect('/');
});
module.exports = router;
```

NB : En bleu les modifications, on place les variables à la place des chaînes de caractères et on renomme les variables spécifiques. Pensez à modifier le template handlebars et utiliser result comme données dans le {{#each result}}...{{/each}}.

Node, Express, MongoDB et Bases SQL

- Vous allez coder un contrôleur nommé **“selectByIdSQL.js”** :

- Il permettra d'ajouter un paramètre dans une clause Where d'une requête. Le paramètre est passé dans l'URL.
- Nous allons utiliser le champ code de la table countries comme `_id`. Notre requête SQL sera :

```
SELECT * FROM countries where code = :_id
```

- Nous pouvons nous inspirer du contrôleur **“modifyUsers.js”** pour le type de route :

```
router.route('/:_id').get(function(req, res) {
```

- Une requête paramétrée Sequelize nécessite un objet **“options”** avec l'attribut **“replacements”**.

Node, Express, MongoDB et Bases SQL

- Requête paramétrée :

- Voici comment initialiser notre objet replacements dans le module contrôleur “**selectByIdSQL.js**” :

```
(...)  
if ((req.session.passport) && (req.session.passport.user != null)) {  
  var options = {};  
  options.replacements = req.params;  
  options.type = sequelize.QueryTypes.SELECT;  
(...)
```

- Nous pouvons maintenant exécuter la requête sur la base de données via la variable globale “**sequelize**” :

```
(...)  
GLOBAL.sequelize.query(req.message.sql_query, options)  
  // SQL query success  
  .then(function(datas) {  
    console.log('listes des datas : ', datas);  
    res.render(req.message.view, {  
      title: 'List from SQL postgreSQL',  
      result: datas  
    });  
  })  
  // SQL query error  
  .catch(function(err) { console.log('error select', err);});  
(...)
```

Node, Express, MongoDB et Bases SQL

- Contrôleur selectByIdSQL.js :

```
var express = require('express');
var router = express.Router();

/* GET one record from SQL query by one parameter which it
can be _id or other like field code into countries table. */
router.route('/:id').get(function(req, res, next) {
  if ((req.session.passport) && (req.session.passport.user != null)) {
    var options = {};
    options.replacements = req.params;
    options.type = sequelize.QueryTypes.SELECT;
    GLOBAL.sequelize.query(req.message.sql_query, options)
      // SQL query success
      .then(function(datas) {
        console.log('listes des datas : ', datas);
        res.render(req.message.view, {
          title: 'List from SQL postgresSQL',
          result: datas
        });
      }) // SQL query error
      .catch(function(err) {
        console.log('error select', err);
      });
    // Not authenticated redirect login
  } else res.redirect('/');
});
module.exports = router;
```

NB : On remarquera l'enchaînement des fonctions : .then et .catch cela car la query Sequelize est une **Promise.**