NODE-JS / EXPRESS / MONGO-DB

par

Stéphane MASCARON
Architecte Logiciels Libres

Introduction

Définition : *« Node.js, en abrégé Node est une plate-forme logicielle libre et évènementielle en* ***JavaScript*** *orienté vers les applications réseau qui doivent pouvoir monter en charge ».*

Node n'est donc pas un Framework ! Comme on peut le lire parfois.

On pourrait plutôt comparer Node à la JVM pour Java, c'est un environnement d'exécution de scripts JavaScript (ES5, ES6).

C'est donc un interpréteur JavaScript basé sur le moteur JS de Google : V8.

Pourquoi Node.js

- **Sa simplicité** : de part l'utilisation du langage JavaScript : pas de multi-thread, mono-thread d'exécution, langage de script dynamique orienté Objets peu typé, langage évènementiel.
- **Ses performances** : Node.js utilise V8 le moteur JavaScript de Google, reconnu pour ses performances et surtout sa capacité à compiler le JavaScript en langage machine (x86, ARM ou MIPS)
- **Unification du langage** : Utiliser qu'un seul langage de développement côté client et serveur dans la mesure ou vous n’utilisez pas de framework à transpilation (TypeScript-Angular >2).

- **Installation d'express**
  - *Pour créer des applications **express** il existe plusieurs possibilité, 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. (Linux)*

```
$ 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 : **--view=hbs** précise l'utilisation d'Handlebars comme moteur de template)

```
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
```

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

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

  - *La configuration est terminé lorsque le preompt 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

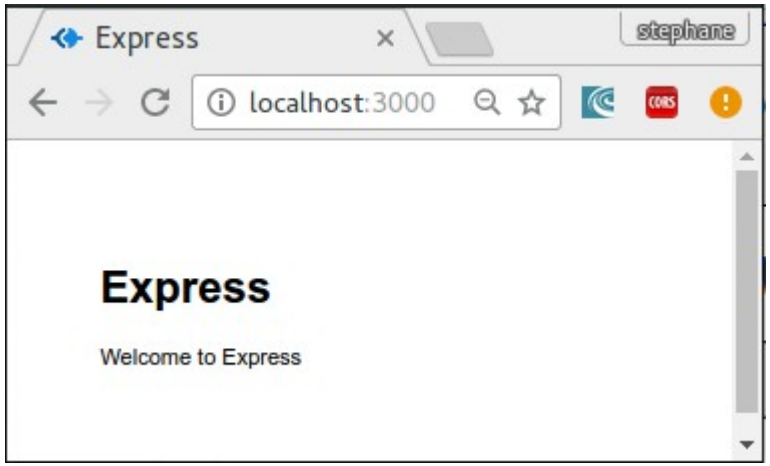
- *Exécution de votre première application Express :*

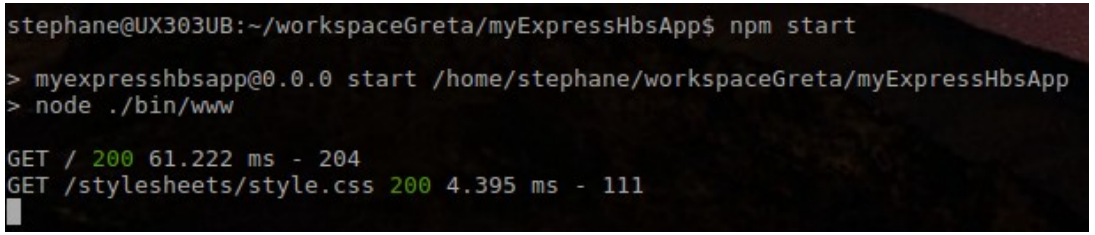
```
stephane@UX303UB:~/workspaceGreta/myExpressHbsApp$ npm start

> 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:*





It Works ! ;-)

- **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 pubic 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 '/users'
└── 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)
```

- **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}}}**.*

- **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>
```

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

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

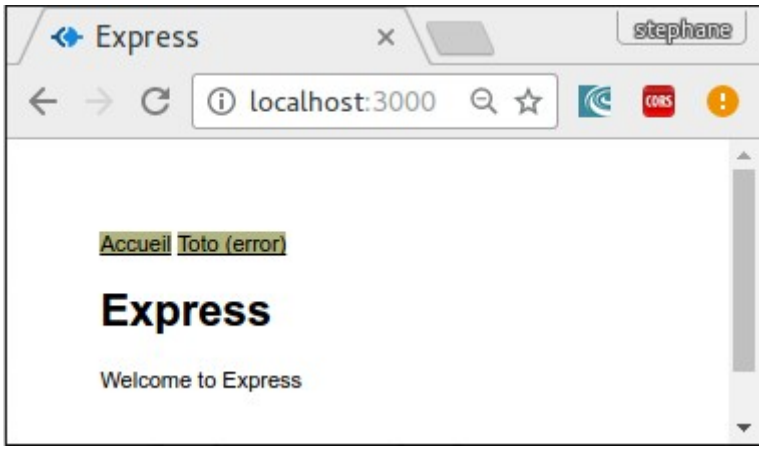
  - *De la même façon nous allons ajouter une barre de navigation (lien href). On crée un fichier **nav.hbs** dans '**partials**' :*

```
<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;
}
```

Debugger une application Node.js via Chrome

- Pour lancer node.js en mode Debug, il faut utiliser l'option :

node --inspect ./bin/www

- Elle peut aussi s'utiliser avec nodemon, module permettant le rechargement des fichiers modifiés dans l'éditeur de code après un CTRL+S
- Installer nodemon :

```
$ npm install -g nodemon
(…)
  └─┬ create-error-class@3.0.2
    └── capture-stack-trace@1.0.1
```

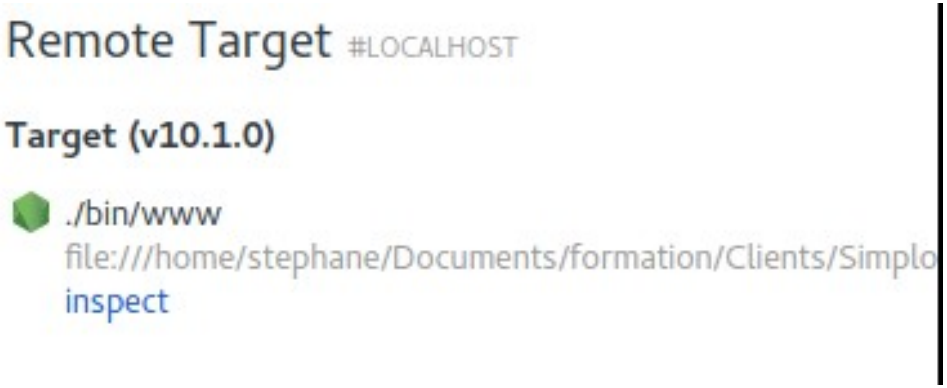
- Puis lancer le Node.js en mode debug :

```
$ nodemon --inspect ./bin/www
```

- **Debugger une application Node.js via Chrome**
  - Pour lancer l'éditeur de code Debug de chrome, lancer "chrome" et dans l'URL tapez :

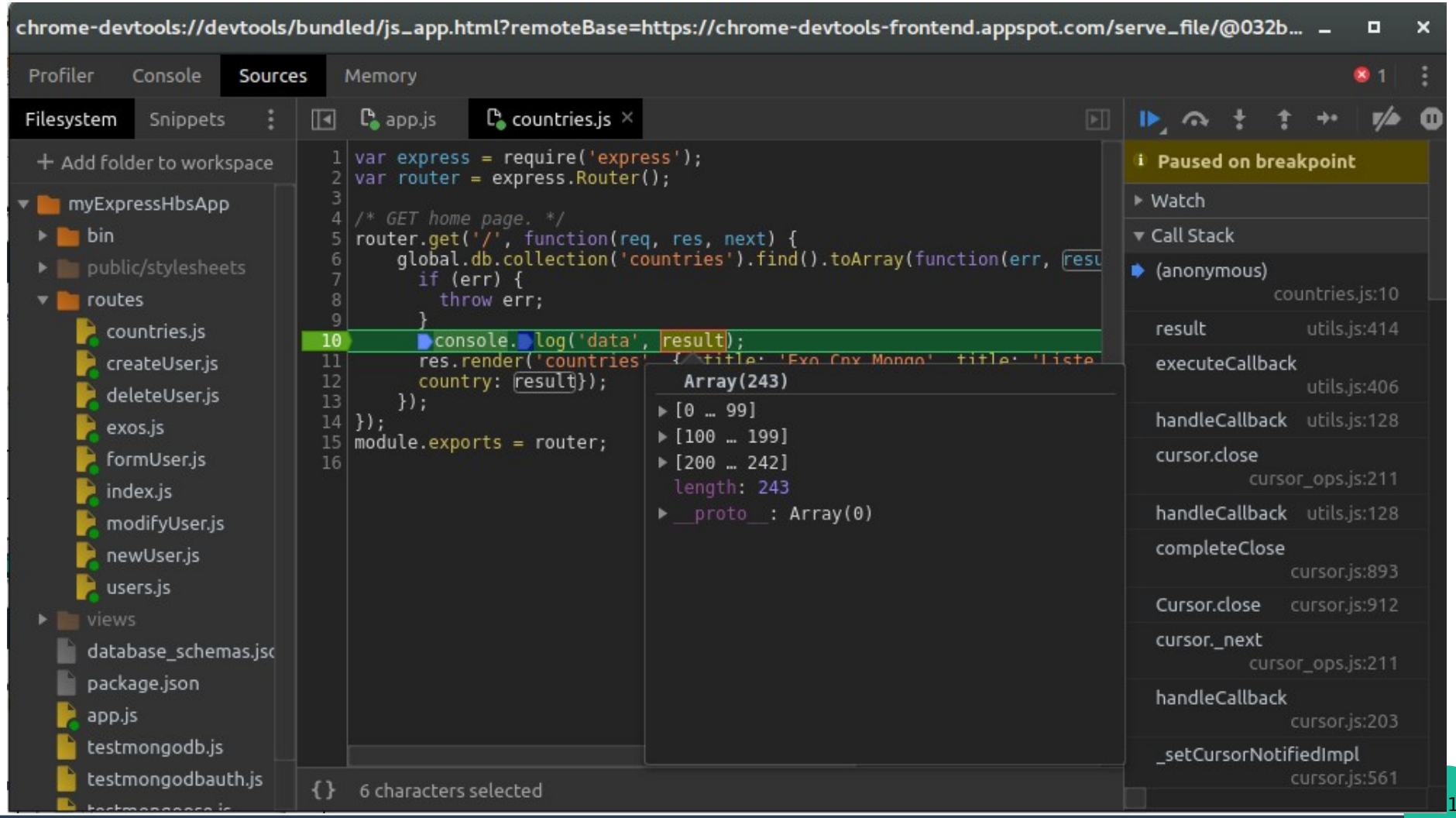
    chrome://inspect

  - Vous devez voir apparaître l'application Node.js
  - Cliquez sur le lien "**Inspect**" il ouvre le debugger de chrome dans une nouvelle fenêtre



Node, Express, MongoDB

- Debugger une application Node.js via Chrome : le debugger s'affiche dans une fenêtre :



- **Exercice : ajouter un footer à notre application**
  - En utilisant les exemples précédents, ajouter un **footer.hbs** donnant le nom de votre entreprise ;-) la date, des logos twitters Facebook, … ce que vous voulez
  - Modifiez les contrôleurs, dans le dossier routes qui font un res.send('…') par une instruction "**res.render('NomTemplate',{objet JSON})** par exemple pour hello.js on va écrire : "**res.render('users', {}) ;**"
  - Faites le aussi pour « users.js » qui devra renvoyer un template "users.hbs".

- **Travaux pratiques NodeJS + Express + HBS**
  - Entraînement à la création d'une application Express avec NodeJS :
    - Créez une application Express avec le CLI express-generator nommée : "**myAppExpress**" :
      $ express --view=hbs myAppExpress
    - Dans cette application ajoutez un dossier "**partials**" dans le répertoire "**views**", puis dans « **app.js** » ajoutez le code permettant de prendre en compte ce dossier "**partials**".
    - Créez les templates : **head.hbs ; nav.hbs ; footer.hbs** (dans partials) et **users.hbs** (dans views).
    - Remplacez le « **res.send()** » par « **res.render('users',{})** » pour charger le fichiers de template pour users.js.
    - Sur le modèle du contrôleur « **users.js** » créez un contrôleur « **countries.js** », une vue « **countries.hbs** » qui affichera la liste des pays dans une liste déroulante. (à faire à la main).

Node, Express, MongoDB

- **Exercice : ajouter un footer à notre application suite et correction :**
  - Correction : **partials/footer.hbs**

```
<footer class="footer">
    <h1> ceci est le footer : &copy; E.I. SMaLL 2018</h1>
</footer>
```

  - Modifiez : **public/stylesheets/style.css**

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

MONGO-DB

par Stéphane MASCARON
Architecte Logiciels Libres

- **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.

```
$ sudo apt install mongodb
```

  - Dans le répertoire de votre application vous devez ajouter mongodb aux dépendances et dans les **node_modules** :

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

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

- **Accéder à une base de données :**

```
// a écrire dans un fichier testmongodb.js à la racine de votre projet myExpressHdbApp
var dbClient = require('mongodb').MongoClient;
console.log('--> mongoClient : ', dbClient);
var assert = require('assert');
var url = 'mongodb://localhost:27017/gretajs'; // Connection URL
// Use connect method to connect to the server
dbClient.connect(url, { useNewUrlParser: true },
  function(err, client) {
  assert.equal(null, err);
  console.log("Successfully connected to server");
  global.db = client.db('gretajs');
  console.log('global.db : ' , global.db);
  client.close();
});
```

Il faut que vous lanciez la base de données mongoDB :

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

Nous allons pouvoir tester cette connexion dans un terminal :

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

- **Sécuriser la connexion à mongoDB**
  - Pour cela il faut créer un user admin sur la base via le shell:

```
$ mongo
> use admin
Switched to db admin
> db.createUser(
...   {
...     user: "myUserAdmin",
...     pwd: "abc123",
...     roles: [ { role: "userAdminAnyDatabase", db: "admin" }, "readWriteAnyDatabase" ]
...   }
... );
Successfully added user: {
    "user" : "myUserAdmin",
    "roles" : [
        {
            "role" : "userAdminAnyDatabase",
            "db" : "admin"
        },
        "readWriteAnyDatabase"
    ]
}
```

  - Ensuite il faut lancer la base avec l'option "--auth"

```
$ sudo mongod --auth
```

- **Sécuriser la connexion à mongoDB**
  - Maintenant il faut se connecter avec le login et le password défini précédemment :

```
$ mongo -u "myUserAdmin" -p "abc123" --authenticationDatabase "admin"
MongoDB shell version: 3.2.20
connecting to: test
> use gretajs
switched to db gretajs
> db.createUser (
  {
    user:"greta",
    pwd: "azerty",
    roles: [
      {role: "readWrite", db: "gretajs"},
      {role: "read", db: "reporting"}
    ]
  }
)
Successfully added user: {
    "user" : "greta",
    (…)
}
> exit
```

- **Sécuriser la connexion à mongoDB**
  - Maintenant il faut se connecter avec le login et le password défini précédemment :

```
$ mongo -u "myUserAdmin" -p "abc123" --authenticationDatabase "admin"
MongoDB shell version: 3.2.20
connecting to: test
> use gretajs
switched to db gretajs
> db.createUser (
  {
    user:"greta",
    pwd: "azerty",
    roles: [
      {role: "readWrite", db: "gretajs"},
      {role: "read", db: "reporting"}
    ]
  }
)
Successfully added user: {
    "user" : "greta",
    (…)
}
> exit
```

- **Sécuriser la connexion à mongoDB :**

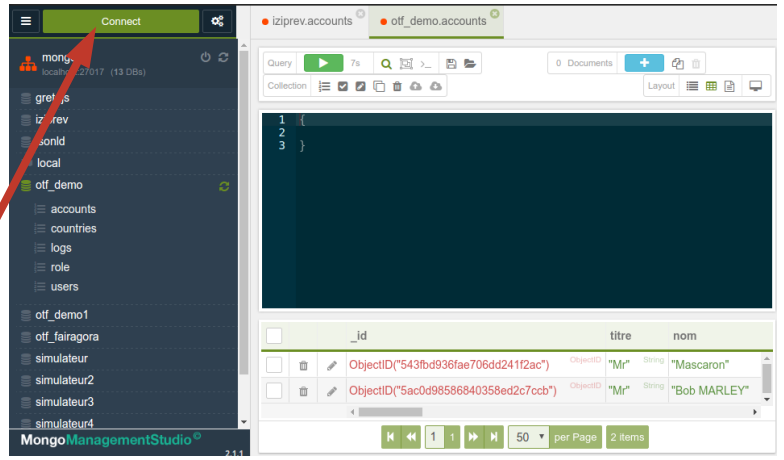
```
var dbClient = require('mongodb').MongoClient;
console.log('--> mongoClient : ', dbClient);
var f = require('util').format;
var assert = require('assert');
var user = encodeURIComponent('greta');
var password = encodeURIComponent('azerty');
const authMechanism = 'DEFAULT';
// formatage de l'URL de connexion
var url = f('mongodb://%s:%s@localhost:27017/gretajs?authMechanism=%s',
            user, password, authMechanism); // Connexion URL
// Connexion effective à la base de donées
dbClient.connect(url, { useNewUrlParser: true },
  function(err, client) {
    assert.equal(null, err);
    console.log("Successfully connected to server");
    global.db = client.db('gretajs');
    console.log('global.db : ' , global.db);
    client.close();
  }
);
```

Nous allons pouvoir tester cette connexion dans un terminal :

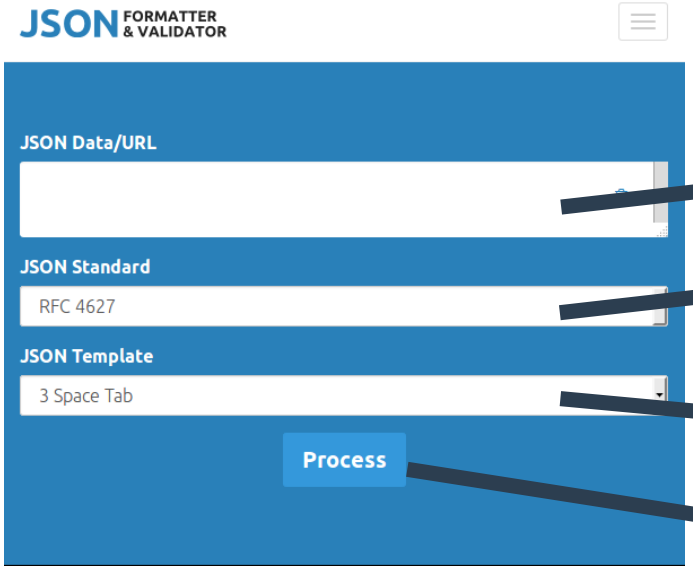
```
$ node ./testmongodbauth.js
Connected successfully to server
(…)
```

Node, Express, MongoDB

- **Outils pour le développement avec mongoDB :**
  - Mongo Management Studio cliquez sur ce bouton pour la configuration de MMS (Création collection exos)



- **JSON Formatter-Validator**



Coller ici le texte en JSON

Format RFC (laissez par défaut)

Indentation du code JSON

Cliquez sur “Process”

Node, Express, MongoDB

- Création de la collection '**exercices**' via le shell :

```
> db.createCollection("exercices")
{ "ok" : 1 }
> show collections
exercices
> db.exercices.insert({libelle: "Exos1"})
WriteResult({ "nInserted" : 1 })
> db.exercices.insert({libelle: "Exos2"})
WriteResult({ "nInserted" : 1 })
> db.exercices.insert({libelle: "Exos3"})
WriteResult({ "nInserted" : 1 })
> db.exercices.insert({libelle: "Exos4"})
WriteResult({ "nInserted" : 1 })
> db.exercices.insert({libelle: "Exos5"})
```

- Voyons les enregistrements qui ont été insérés :

```
> db.exercices.find()
{ "_id" : ObjectId("5babb4981be41ac869f07764"), "libelle" : "Exos1" }
{ "_id" : ObjectId("5babb4af1be41ac869f07765"), "libelle" : "Exos2" }
{ "_id" : ObjectId("5babb4b31be41ac869f07766"), "libelle" : "Exos3" }
{ "_id" : ObjectId("5babb4b51be41ac869f07767"), "libelle" : "Exos4" }
{ "_id" : ObjectId("5babb4bc1be41ac869f07768"), "libelle" : "Exos5" }
>
```

Node, Express, MongoDB

- **Ajoutons une connexion base de données pour une action ‘/exos’ qui listera les enreg. d’exercices :**
  - Créons la connexion dans ‘**app.js**’ :

```
(…)
global.db={};
var mongoClient = require('mongodb').MongoClient;

// Connexion URL
//var url = 'mongodb://greta:azerty@127.0.0.1:27017/gretajs?authMechanism=DEFAULT';
var url = 'mongodb://127.0.0.1:27017/gretajs';
// Utilisation de la methode “connect” pour se connecter au serveur
mongoClient.connect(url, function(err, client) {
  global.db = client.db('gretajs');  //On met en global la connexion à la base
  console.log("Connected successfully to server: global.db initialized");
});
(…)
module.exports = app;
```

  - Pensez à ajouter le require et le use

```
var usersRouter = require('./routes/users');
var exosRouter = require('./routes/exos');
(…)
app.use('/exos', exosRouter);
```

Node, Express, MongoDB

- Creation de la route '**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', {stitle: 'First Cnx Mongo',
                        title: 'Liste déroulante',
                        exos: result});
  });
});
module.exports = router;
```

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

```
<h1>{{ stitle}}</h1>
<h2>Express App : {{title}}</h2><br>
   <select name="exos">
    {{#each exos}}
        <option value="{{this._id}}">{{this.libelle}}</option>
    {{/each}}
</select>
```

Node, Express, MongoDB

- **Résultat de l’appel de l’action ‘exos’ :**

Accueil Toto (error)

First Cnx Mongo

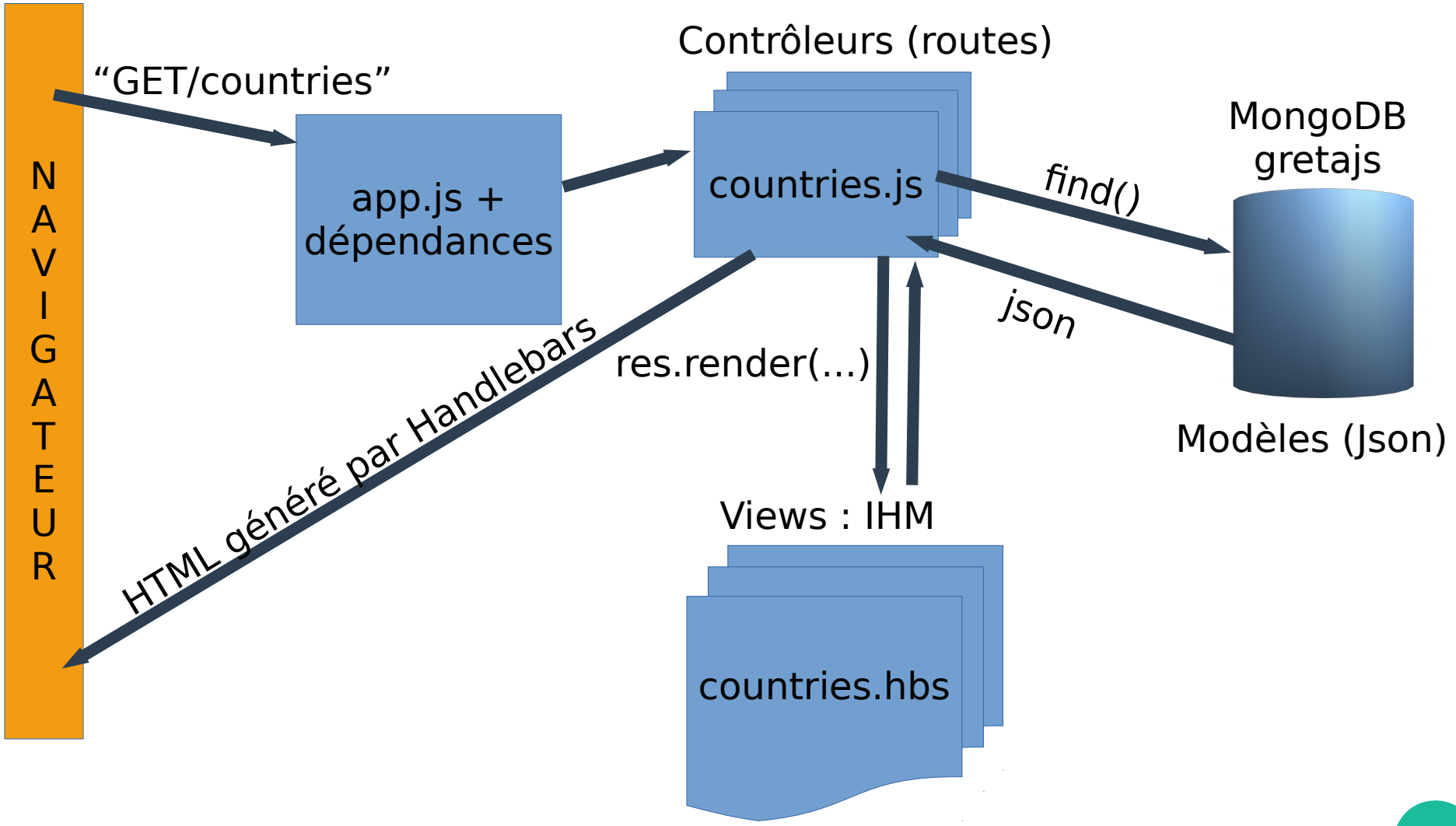
Express App : Liste déroulante

Exos1
Exos1
Exos2
Exos3
Exos4
Exos5

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

Node, Express, MongoDB

- **Schéma architecture Application Express/Handlebars/MongoDB :**



- **Exercice :**
  - En utilisant le code vu précédemment, vous ajouterez une action nommée '**/countries**' un controleur '**countries.js**' et une vue '**countries.hbs**' qui affiche la liste des pays dans un select (liste déourlante).
  - Le fichier countries est téléhargeable ici

```
[{
      "_id":"572377f1e3d6f2d245000001",
      "name":"Afghanistan",
      "code":"AF",
   },
(…)
]
```

Pour importer en ligne de commande un fichier json dans une collection (une seule ligne) : "mongoimport"

```
$ mongoimport -c countries -d gretajs -u greta -p azerty --jsonArray
--file ../countries.json
```

- **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.registerPartials(__dirname + '/views/partials', function () {
    console.log('partials registered');
});

hbs.registerHelper('helper_name', function(...) { ... });
```

  - On peut créer des fonctions qui vont permettre de faire des comparaisons, des conversions, l'intégration de composants.

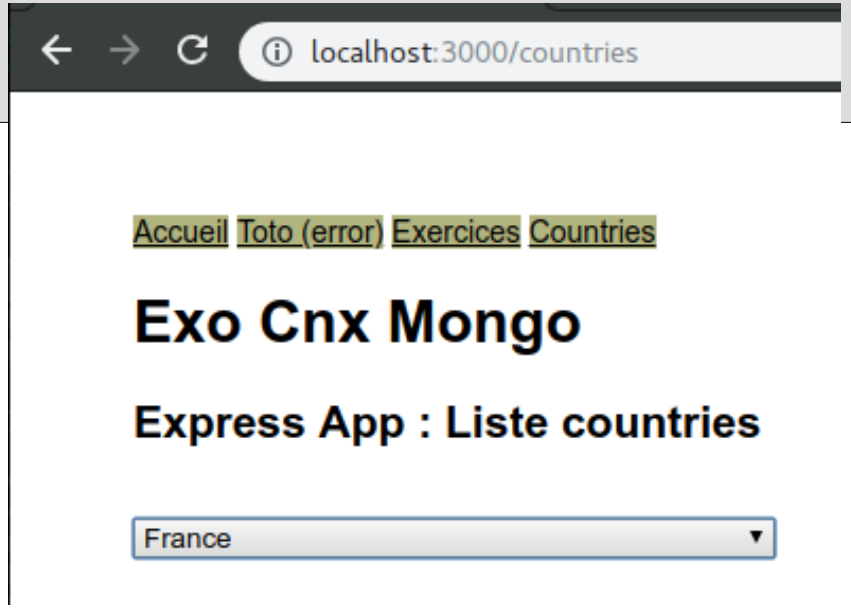
- **Intégration dans notre ‘app.js’**
  - Ajoutons un simple Helper qui va comparer 2 chaînes de caractères :

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

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

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



- **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'OTF² qui contient plusieurs helpers développés au fur et à mesure des besoins.

- **Exercice correction (sans ajouter un helper) mais un opérateur :**
  - On créé la fonction '**isTabEmpty**' dans app.js :

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

  - Code permettant d'appeler l'opérateur 'isTabEmpty' :

```
{{#compare country null operator="isTabEmpty"}}
       <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** ou le **shell** pour stocker des utilisateurs : “**users**”. On créé 1 user via le shell MongoDB (commande mongo) :

```
{
     "name": "MASCARON",
     "firstName": "Stéphane",
     "login": "steph",
     "mdp": "azerty",
     "function": "Free Software Architecte",
     "office": "SMaLL Stéphane MASCARON Architecte Logiciels Libres",
     "date_naiss": "22/04/1970",
     "adresse1": "xxx avenue des freesoftwares",
     "adresse2": "",
     "cp": "40000",
     "city": "MONT-DE-MARSAN",
     "mobile_phone": "0609090909”,
     "home_phone": "0558080808"
}
```

  - Créez un contrôleur « **routes/users.js** », une vue « **views/users.hbs** » qui affiche les données de l’utilisateur. Pensez à charger le module dans app.js et à définir la route (use).

- **Insérer l'utilisateur via le Shell MongoDB :**
  - Lancer un shell mongo :

```
$ mongo
```

```
>use gretajs
switched to db gretajs
> db.createCollection('users')
{ "ok" : 1 }
> db.users.insert({name:"MASCARON", firstName: "Stephane", login: "steph", mdp: "azerty",
function: "Free Software Architect", office: "EI SMALL", date_naiss: "22/04/1970",
adresse1: "xxxx rue des logiciels libres", adresse2: "", cp: "40000", city: "MONT-DE-MARSAN",
mobile_phone: "0606060606", home_phone: "0558080808"})

WriteResult({ "nInserted" : 1 })
> db.users.find()
{ "_id" : ObjectId("5babd87c5a4a7fdc4b7e3a9f"), "name" : "MASCARON",
"firstName" : "Stephane", "login" : "steph","mdp" : "azerty",
"function" : "Free Software Architect", "office" : "EI SMALL",
"date_naiss" : "22/04/1970", "adresse1" : "xxxx rue des logiciels libres",
"adresse2" : "", "cp" : "40000", "city" : "MONT-DE-MARAN",
"mobile_phone" : "0606060606", "home_phone" : "0558080808" }
```

  - De la même façon créez votre utilisateur avec votre nom.

Nb : cette collection « users" va nous permettre plus tard de gérer l'authentification dans l'application (login, mdp).

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>
<br>
<select name="users" id="user">
    {{#compare users null operator="=="}}
        <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><br/>
<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>
```

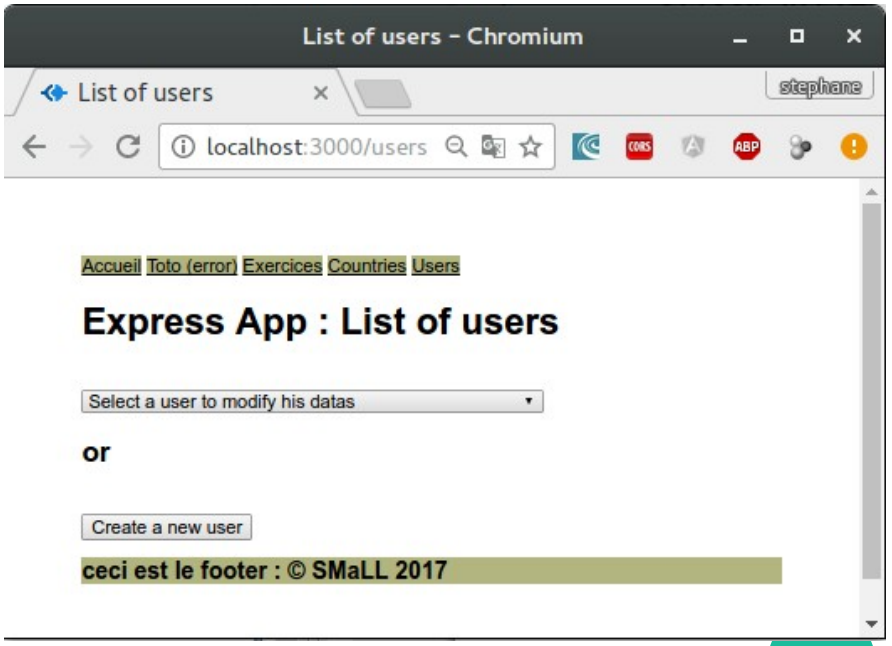
Node, Express, MongoDB

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

```
<nav class="barnav">
    <ul id="nav">
        <li><a href="/">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="/users">Users</a></li>
    </ul>
</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);  // Enregistre 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 methode="POST" action="{{form_action}}" id="formAddUser">
   <input type="hidden" name="id" value="{{user._id}}" />
    name : <input type="text" name="name" value="{{user.name}}" /><br/>
    firstName : <input type="text" name="firstName" value="{{user.firstName}}" /><br/>
    login : <input type="text" name="login" value="{{user.login}}" /><br/>
    mdp : <input type="password" name="mdp" value="{{user.mdp}}" /><br/>
    function : <input type="text" name="function" value="{{user.function}}" /><br/>
    office : <input type="text" name="office" value="{{user.office}}" /><br/>
    date_naiss : <input type="text" name="date_naiss" value="{{user.date_naiss}}" /><br/>
    adresse1 : <input type="text" name="adresse1" value="{{user.adresse1}}" /><br/>
    adresse2 : <input type="text" name="adresse2" value="{{user.adresse2}}" /><br/>
    cp : <input type="text" name="cp" value="{{user.cp}}" /><br/>
    city : <input type="text" name="city" value="{{user.city}}" /><br/>
    mobile_phone : <input type="text" name="mobile_phone" value="{{user.mobile_phone}}" /><br/>
    home_phone : <input type="text" name="home_phone" value="{{user.home_phone}}" /><br/>
    <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>
```

2- modifyUser.hbs :

```
<h1> Express App : {{title}}</h1>
<ul>
    <li>name :.......................... {{user.name}}</li><br/>
    <li>firstName :..................... {{user.firstName}}</li><br/>
    <li>login :......................... {{user.login}}</li><br/>
    <li>mdp :........................... {{user.mdp}}</li><br/>
    <li>function :...................... {{user.function}}</li><br/>
    <li>office :........................ {{user.office}}</li><br/>
    <li>date_naiss :.................... {{user.date_naiss}}</li><br/>
    <li>adresse1 :...................... {{user.adresse1}}</li><br/>
    <li>adresse2 :...................... {{user.adresse2}}</li><br/>
    <li>cp :............................ {{user.cp}}</li><br/>
    <li>mobile_phone :.................. {{user.mobile_phone}}</li><br/>
    <li>home_phone :.................... {{user.home_phone}}</li><br/>
</ul>
```

- 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.

3- formUser.js :

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

/* GET users from _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 onlines.

4- modifyUser.js : 3 Blocs imbriqués, 3 Callbacks !

```
var express = require('express');
var router = express.Router();
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 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 ré-afficher les données modifiée dans la base.

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

- **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();
/* 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 mode --inspect) 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 remaquera 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**.

Démarche de généralisation

- Lorsque nous développons une application Node.js avec Express nous avons a charger par un “**require**” le module contrôleur dans app.js, puis nous avons aussi à définir la route dans un « **app.use(‘/xxxx’, nomContrôleur)** » pour chaque action, nous chargeons donc en RAM tous les contrôleurs au lancement de l’application.

- Nous allons donc factoriser le chargement des contrôleurs et externaliser leur définition dans un fichier JSON que l’on pourrait comparer à un annuaire d’actions (pathname).

- Pour ce faire nous devons réfléchir comment définir la structure de cet annuaire. Nous devons gérer l’action (pathname) comme clé mais également la méthode HTTP (GET, POST) afin de pouvoir gérer les 2 cas.

- **Exemples d'un chargeur de routes :**
  - Ecrire un chargeur de contrôleurs 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.

- **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 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 :

- **Workshop router dynamique :**
  - 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);
```

- **Routeur dynamique (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
```

A T T E N T I O N ! ! !
NB : Pensez à changer dans le fichier /bin/www le nom du fichier app en appdyn

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

```
/* Fonction qui permet d'aguiller 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]
    // On récupère la méthode HTTP : GET ou POST
    type = req.method;
    // [type + path] permet de retrouver le bon contrôleur
    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;
```

Node, Express, MongoDB

- **Routeur dynamique (suite) :**
  - Comment gérer le paramètre “:_id” dans un contrôleur

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

/* Delete one user into database. */
router.route('/:_id').get(function (req, res) {
    // ici on a un élément en plus _id dans l'URL "/deleteUser/5b0d5e95488eaf5161489a1e"
    // on découpe l'url et on ne récupère que le premier élément du tableau "deleteUser"
    // on ajoute devant un "/"
    var path = '/' + req.originalUrl.split('/')[1];  // retourne ‘/deleteUser’ dans path
    var type = req.method;
    console.log('req.originalUrl : ', req.originalUrl);
    global.db.collection('users').remove({_id: new ObjectID(req.params._id)},
        function (err, result) {
            if (err) {
                throw err;
            }
            console.log('createUser: ', result);
            global.db.collection("users").find().toArray(function(err, listusers) {
                res.render(global.actions_json[type+path].view, {
                    title: 'List of users :',
                    users: listusers
                });
            });
        } // fin callback du Delete
    ); // fin de l'insert()
}); // fin de la gestion de la route
module.exports = router;
```

- **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, nous pourrions ajouter le type MIME de retour (html, json, ...), bref, des paramètres supplémentaires qui peuvent être nécessaires.
  - Maintenant dans notre application Express, MongoDB, Handlebars il n’est plus nécessaire d’ajouter du code dans appdyn.js pour ajouter une action !! youpi ;-)
  - Il suffit de compléter et d’ajouter dans le fichier **config_actions.json** une action et un contôleur ...

- **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. (équivalent de hibernate en Java)
  - Pour l’installer, il faut ajouter 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 « testMongoose.js » initialisant une connexion à MongoDB via Mongoose :

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://@127.0.0.1:27017/gretajs', {useNewUrlParser: true},
  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();
});
```

- **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();
});
```

- **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().

- **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/simplonjs', 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 commenter en bas du fichier “appdyn.js” la connexion via le driver natif mondodb.

- **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.

- **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 inséré. On l’ajoute à req.query que l’on passe en paramètre de la méthode “create([req.query],...”

- **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 caractères _id récupérée dans la liste déroulante des utilisateurs.

- **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 comment paramétrer le nom du "Model" (au sens Mongoose) pour l'action considérée...

- **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;
```

- **Exercice Correction :**
  - Nous avons pris le parti de modifier la vue “**users.hbs**” et d’ajouter une dexième 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",
        "modelName" : "Users",
        "view": "users"
     }
}
```

Node, Express, MongoDB

- **Écriture d'un module générique pour lire des données : /routes/finder.js**
  - Illustration avec le contrôleur « finder.js » permettant de requêter la collection passée en paramètre dans le fichier config_actions.json dans la variable : **model**

```
/* ******************************************************
**  Module générique pour faire un "find()" sans filtre *
** ******************************************************/
var express = require('express');
var router = express.Router();
var mongoose = require('mongoose');

/* GET users listing. */
router.get('/', function (req, res, next) {
    var path = "/" + req.originalUrl.split('/')[1];
    var type = req.method;
    global.schemas[global.actions_json[type + path].modelName].find({}, function (err, result) {
        if (err) { throw err;}
        console.log(result);
        if (result.length == 0) result = null;
        res.render(global.actions_json[type + path].view,
                   { title: 'List of results :', data: result }
        );
    });
});
module.exports = router;
```

Node, Express, MongoDB

- **Écriture d'un module générique pour supprimer des données : /routes/delete.js**
  - Illustration avec le contrôleur «delete.js » permettant de supprimer de la collection un enregistrement via son _id :

```
/* ********************************************************
**  Module générique pour faire un "deleteOne()" via l'_id *
** ********************************************************/
var express = require('express');
var router = express.Router();
var mongoose = require('mongoose');
var ObjectId = mongoose.Types.ObjectId;

/* DELETE record from _id into url and into config_actions.json */
router.route('/:_id').get(function (req, res) {
    var path = "/" + req.originalUrl.split('/')[1];
    var type = req.method;
    var model = global.actions_json[type + path].model;
    global.schemas[model].deleteOne({_id: new ObjectId(req.params._id)}, function (err, result) {
        if (err) {
            throw err;
        }
        global.schemas[model].find({}, function (err, result2) {
            console.log("result after delete : ", result2);
            res.render(global.actions_json[type + path].view, {
                title: "List of " + model,
                data: result2
            });
        });
    });
});
module.exports = router;
```

Node, Express, MongoDB

- **Écriture d'un module générique pour lire et liéer des données entre 2 collections « populate » :**
  - Illustration avec le contrôleur «**populate.js** » permettant de lire une collection et de récupérer par une clé externe les données d'un autre collection via l'id par exemple :

```
/* ********************************************************
**  Module générique pour faire un "populate()" via l'_id *
** ********************************************************/
var express = require('express');
var router = express.Router();
var mongoose = require('mongoose');

/* GET users listing. */
router.get('/', function (req, res, next) {
    var path = "/" + req.originalUrl.split('/')[1];
    var type = req.method;
    var model = global.actions_json[type + path].model;
    var pop_ref = global.actions_json[type + path].pop_ref;
    global.schemas[model].find({}).populate(pop_ref).exec(function (err, result) {
        if (err) {
            console.log("error: ", err);
            return handleError(err);
        } else {
            if (result.length == 0) result = null;
            res.render(global.actions_json[type + path].view, { title: 'List of results', data: result });
        }
    });
});

module.exports = router;
```

Node, Express, MongoDB

- **Écriture de la configuration dans le fichier « config_actions.json » pour ces 3 contrôleurs :**
  - Illustration avec les action suivantes en lien avec le schéma de la collection :

```
(...)
   "GET/users": {
        "controler": "finder",
        "view": "users",
        "model": "Users"
    },
     "GET/deleteUser": {
        "controler": "delete",
        "view": "users",
        "model": "Users"
    },
    "GET/userspopulate" :  {
        "controler": "populate",
        "view": "userspopulate",
        "model": "Users",
        "pop_ref": "countries"
    },
(...)
```

```
16          }
17      },
18      "Users": {
19          "collection": "users",
20          "schema": {
21              "_id": "ObjectId",
22              "name": "String",
23              "firstName": "String",
24              "login": "String",
25              "mdp": "String",
26              "function": "String",
27              "office": "String",
28              "date_naiss": "String",
29              "adresse1": "String",
30              "adresse2": "String",
31              "cp": "String",
32              "city": "String",
33              "countries": {
34                  "type": "String",
35                  "ref": "Countries"
36              },
37              "mobile_phone": "String",
38              "home_phone": "String"
39          }
40      }
41  }
```

- **Exercice T P** : Intégrez ces éléments dans votre applications et créez le contrôleur qu'il manque : « **update.js** » qui permet de modifier un enregistrement d'une collection.

  NB : Penser à changer dans les **{{#each }}** des vues le nom de la variables qui contient les données par la variable « **data** » puisque dans les render() nous devons être générique sur les termes nous renvoyons : « **{data: result}** » .

Node, Express, MongoDB

- **WORKSHOP : Intégrer Passport dans notre application exemple.**
- **Le module Passport pour l'authentification**
  - Nous avons presque fini notre framework pour développer des applications qui utilisent :
    - Node.js avec comme modules principaux :
      - Express, Handlebars, MongoDB et 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 JWToken et la sécurisation d'API
    - Article SupInfo sur : JWToken & 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 importer les modules installés à notre App Express, dans **appdyn.js** :

```
(…)
    var fs = require('fs');
    var url = require('url');
    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 le **use** de **body-parser,** ceux concernant passport et la session (ici en bleu) :

```
(…)
    app.use(express.bodyParser());
    app.use(session({
        name: 'sessiongreta',
        secret: 'AsipfGjdp*%dsDKNFNFKqoeID1345',
        resave: false,
        saveUninitialized: false,
        cookie: {secure: false}   // à mettre à true uniquement avec un site https.
    }));
    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 devrons appeler la méthode :

    `passport.authenticate('local')`
  - Dans la fonction de callback 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 dynamicaly
require('./dynamicRouter')(app);
```

- **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/" : {
       "controler": "login",
       "view": "login",
       "collection": "",
       "schema": ""
    },
(…)
```

```
    "GET/index": {
       "controler": "index",
       "view": "index",
       "collection": "",
       "schema": ""
    },
(…)
```

  - 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 authentification' });
});
module.exports = router;
```

- **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 controler qui nécessite une authentification vérifie avant de réaliser le traitement que la requête est authentifiée.

- **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 athentification: exos, users, createUser, modifyUser, formUser, newUser. Ne pas modifier index et login !

- **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 précédemments 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": {
        "controler": "logout"
    }
}
```

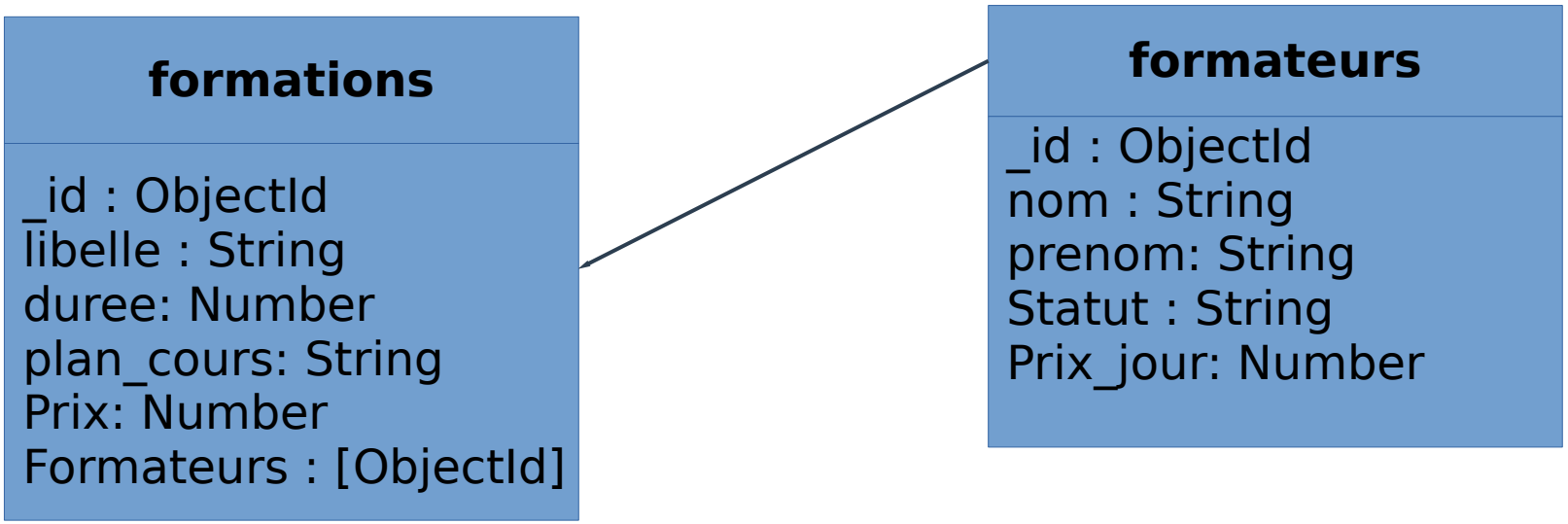
- **Le module Passport pour l'authentification**
  - **Correction Exercice** : Voyon le code de notre controleur de déconnexion :

```
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;
```

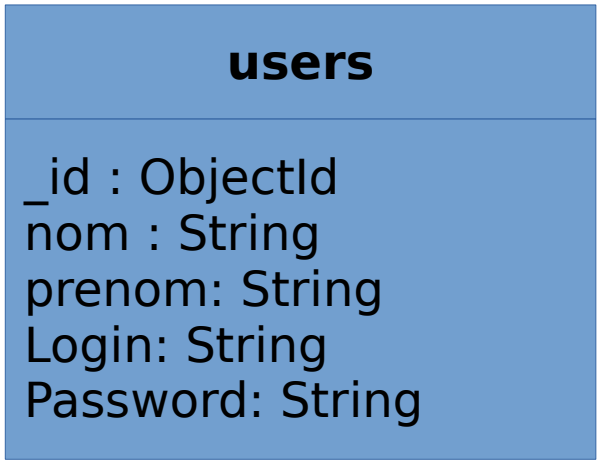
Node, Express, MongoDB

- **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 « formateurs » avec son nom, prenom, login, mdp, liste formations (tableau JS [] de chaîne).
    - Créez les écrans, les contrôleurs, les vues pour gérer les formations et les formateurs.

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



NB: On utilisera la fonctionnalité **populate de Mongoose** pour créer les relations entre les collections et récupérer la liste des formateurs dans la collection formations.



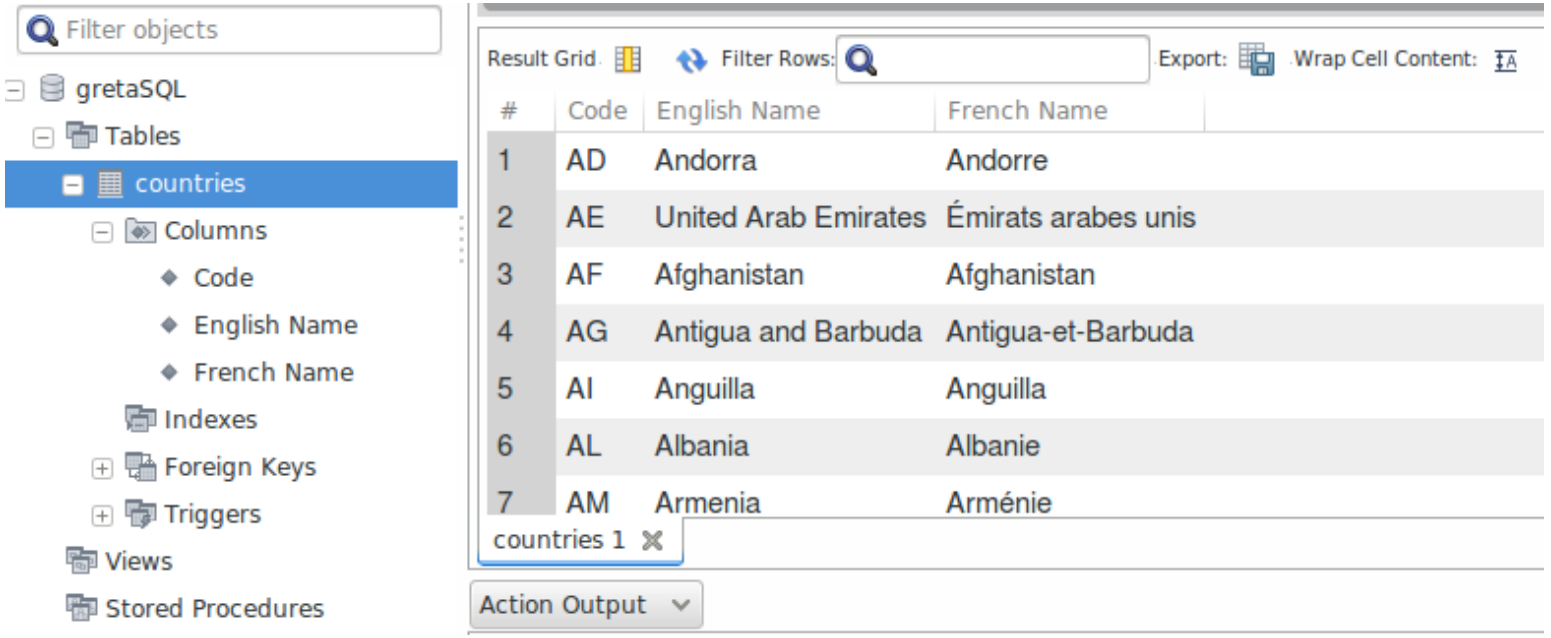
Node, Express, MongoDB

- **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 "controleurs" et non dans app.js.
  - Vous intègrerez 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 !

- **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** ou **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.

- **Ajouter une table dans notre base de données : la table “countries”**
  - Dont les données en csv sont récupérables ici :

    http://mascaron.net/greta/countries_gretasql.csv
  - Vous pouvez utiliser l’importation de données pour la table countries via un logiciel de gestion de votre SGBD : pour MySQL il existe **MySQL Workbench**



Node, Express, MongoDB, et Baseq SQL

- **Pilote MySQL : le module “mysql”**
  - Installez le module pour votre base de données :

```
$ npm install mysql   // driver natif MySQL
```

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

```
var mysql = require('mysql');

var config = {host: 'localhost', user: 'stephane', password: 'azerty', database: 'gretaSQL'};

var connection = mysql.createConnection(config);
connection.connect(function(err) {
    if (err) { console.error('error connecting: ' + err.stack); return; }
    console.log('connected as id ' + connection.threadId);
});

connection.query('SELECT * from countries', function (error, results, fields) {
    if (error) throw error;
    console.log('The first record is: ', results[0]); // results is an array of records
});

connection.end(function (err) {
    console.log('connection mysql closed !');
});
```

- **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
  }});
```

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

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

var sequelize = new Sequelize('gretaSQL', 'stephane', 'azerty', {
  host: 'localhost',
  dialect: 'mysql',

  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);
    });
```

- **Nous allons ajouter la capacité de connexion à la base MySQL pour notre Express App:**
  - Pour ce faire nous avons déjà intégré les modules via : **npm install --save sequelize mysql**.
  - 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('gretaSQL', 'stephane', 'azerty', {
  host: 'localhost',
  dialect: 'mysql',
  pool: { max: 5, min: 0, idle: 10000 }
});
```

- **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

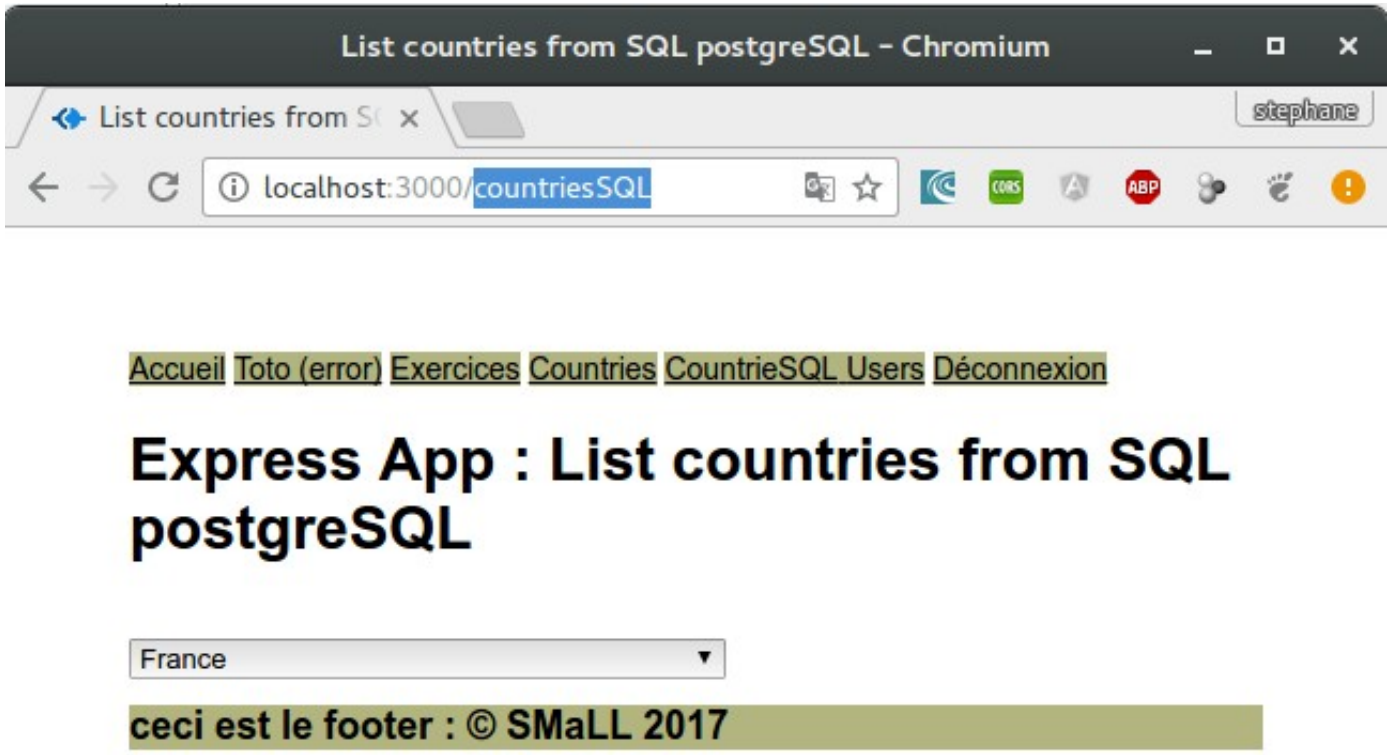
- **Avant de pouvoir tester il faut ajouter la route dans notre config_actions.json :**

```
(…)
"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">CountrieSQL</li>
        <li><a href="/users">Users</a></li>
        <li><a href="/logout">Déconnexion</a></li>
    </ul>
</nav>
```

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



- **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'excution d'une requête :
      - **Le nom du contrôleur** est déjà paramétrée 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.

- **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 cette varaible "**sql_query**", comme on les charge déjà pour récupérer le controler, la vue, le modèle Mongoose, nous pouvons les ajouter à l'objet requête "**req**", en ajoutant une propriété, 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.

- **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.

- **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.

- **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}}.

- **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**”.

- **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);});
(…)
```

- **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 postgreSQL',
                    result: datas
                });
            }) // SQL query error
            .catch(function(err) {
                console.log('error select', err);
            });
        // Not authentified 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.

Node, Express, MongoDB et Bases SQL

- **Exercice : utiliser selectByIdSQL.js pour une autre table : “companies” :**

```
CREATE TABLE public.companies
( -- Identifiant en auto incrément smallserial
  id smallint NOT NULL DEFAULT nextval('companies__id_seq'::regclass),
  name text, -- Nom de l'entreprise
  age smallint, -- nombre d'année d'existence de l'entreprise
  adress1 text, -- première ligne d'adresse de l'entreprise
  adress2 text, -- deuxième ligne d'adresse de l'entreprise
  postal_code character(5), -- Code postal de l'entreprise
  city text -- nom de la ville dans laquelle est installée l'entreprise
)
WITH (
  OIDS=TRUE
);
ALTER TABLE public.companies
  OWNER TO <votre_user>;
     COMMENT ON COLUMN public.companies.id IS 'Identifiant en auto incrément smallserial ';
     COMMENT ON COLUMN public.companies.name IS 'Nom de l''entreprise';
     COMMENT ON COLUMN public.companies.age IS 'nombre d''année d''existence de l''entreprise';
     COMMENT ON COLUMN public.companies.adress1 IS 'première ligne d''adresse de l''entreprise';
     COMMENT ON COLUMN public.companies.adress2 IS 'deuxième ligne d''adresse de l''entreprise';
     COMMENT ON COLUMN public.companies.postal_code IS 'Code postal de l''entreprise';
     COMMENT ON COLUMN public.companies.city IS 'nom de la ville dans laquelle elle est installée';
```

- **Exercice suite : SQL pour injecter des données dans la table companies.**

```
INSERT INTO companies VALUES (3,'SAS Machin Truc',5,'20 rue des bidules','','40000','MONT-DE-MARSAN');
INSERT INTO companies VALUES (4,'SARL Huile de Code',3,'236 route de Canenx',
                              'Parc techno.SO WATT!-La Fabrik', '40000','MONT-DE-MARSAN');
INSERT INTO companies VALUES (5, 'SMALL : Stéphane MASCARON Architecte Logiciels Libres',23,
                              '260 avenue des arènes', '','40090','SAINT-PERDON');
INSERT INTO companies VALUES (6, 'SP Dev : Stéphane PONTEINS',10,'Rue des angles V2','','40100','DAX');
INSERT INTO companies VALUES (7, 'Agence Esens : Laurence MARTIN',15,'13 Rue Léon Bidulle','',
                              '40000','MONT-DE-MARSAN');
```

NB : Utilisez le **code SQL du slide précédent** pour créer une nouvelle table qui va stocker des entreprises, que l'on va appeler "**companies**". A partir des exemples Sequelize précédents, créer un affichage de toutes les "**companies**" et un affichage d'une seule companie filtrer par son "**id**". Puis ajoutez des données dans la nouvelle table companies via les **INSERT** ci-dessus.

Vous devrez créer les actions dans le config_actions.json avec les bons paramètres (view, sql_query et controler)

Node, Express, MongoDB et Bases SQL

- **Sequelize permet également de créer des schémas comme Mongoose.**
  - A partir d'un schéma il est possible d'obtenir une vue objet de notre base de données comme le ferait Hibernate en Java.
  - Pour créer un schéma Sequelize, voici un exemple avec notre table countries.

```
module.exports = function(sequelize, DataTypes) {
  return sequelize.define('countries', {
    code: {
      type: DataTypes.TEXT,
      allowNull: false
    },
    libelle_us: {
      type: DataTypes.TEXT,
      allowNull: false
    },
    libelle_fr: {
      type: DataTypes.TEXT,
      allowNull: true
    }
  }, {
    tableName: 'countries'
  });
};
```

- **Question : Si j'ai une base de données existante de 50 tables pour créer les Schémas je dois le faire à la Main ?**
  - Oui ! ;-) Meuh non !! ;-) il existe un module appelé :

    Sequelize-Auto
  - Installez le en global (-g) dans votre ordinateur

    ```
    npm install -g sequelize-auto
    ```
  - Testez son installation dans un terminal

    ```
    $ sequelize-auto -o "./models" -d gretajs -h localhost -u steph -p 5432 -x azerty -e postgres
    ```

NB : Attention si vous n'avez pas accès, il suffit de créer un lien, raccourcis vers /usr/bin/ de sequelize-auto

- **Dans notre base PostgreSQL nous avons 2 tables : “countries” et “companies” :**
  - Nous allons générer les Schémas via ce module sequelize-auto, pour ce faire placez vous dans le dossier du projet :

```
$ sequelize-auto -o "./models" -d gretajs -h localhost -u steph -p 5432 -x azerty -e postgres
```

  - Cette commande a créée un sous-dossier dans le répertoire du projet nommé “models” qui contient les fichier JavaScript des modèles :

```
stephane@UX303UB:~/workspaceGreta/gretaexos2/models$ ll
total 16
drwxrwxr-x  2 stephane stephane 4096 mars  21 14:42 ./
drwxrwxr-x 11 stephane stephane 4096 mars  21 14:40 ../
-rw-rw-r--  1 stephane stephane  718 mars  21 14:42 companies.js
-rw-rw-r--  1 stephane stephane  378 mars  21 14:42 countries.js
```

Node, Express, MongoDB et Bases SQL

- **Voici le modèle “companies” généré par sequelize-auto**

```
module.exports = function(sequelize, DataTypes) {
  return sequelize.define('companies', {
    _id: { type: DataTypes.INTEGER,
           allowNull: false,
           defaultValue: "nextval(companies__id_seq::regclass)"
    },
    name: { type: DataTypes.TEXT,
            allowNull: true
    },
    age: { type: DataTypes.INTEGER,
           allowNull: true
    },
    adress1: { type: DataTypes.TEXT,
               allowNull: true
    },
    adress2: {
      type: DataTypes.TEXT,
      allowNull: true
    },
    postal_code: {
      type: DataTypes.CHAR,
      allowNull: true
    },
    city: {
      type: DataTypes.TEXT,
      allowNull: true
    }
  }, {
    tableName: 'companies'
  });
};
```

Node, Express, MongoDB et Bases SQL

- **Nous devons écrire un chargeur de modèles Sequelize :**
  - Il faut instancier un objet pour chaque schémas générés, comme on l'a fait pour les schémas mongoose. Ajouter ce code dans "**appdyn.js**" :

```
GLOBAL.modelsSeq = {};
// Loader Sequelize models into GLOBAL.modelsSeq
fs.readdirSync(__dirname + '/models')
    .filter(function(file) {
        return (file.indexOf(".") !== 0);   // on filtre les fichiers ils doivent contenir un . (.js)
     .forEach(function(file) {
        var model = GLOBAL.sequelize.import(path.join(__dirname + '/models', file));
        GLOBAL.modelsSeq[model.name] = model;
        console.log('file read : ' + file);
    });
(…)
```

  - Il nous reste maintenant à utiliser un model, comme ils sont stockés dans la variable "**GLOBAL.modelsSeq**", il est possible de l'utiliser dans un contrôleur pour réaliser une requête.

- **Ecrire un contrôleur qui utilise un model Sequelize pour faire une requête dans la base SQL : “selectAllFromModelSQL.js” :**

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

/* GET companies list into a web page. */
router.route('/').get(function(req, res, next) {
    if ((req.session.passport) && (req.session.passport.user != null)) {

        GLOBAL.modelsSeq["companies"].findAll().then(function(datas) {
            res.render(req.message.view, {
                title: 'List from SQL postgreSQL',
                result: datas
            });

        });
    } else res.redirect('/');
});
module.exports = router;
```

NB : Pensez à modifier la déclaration de Sequelize dans appdyn.js avec PostgreSQL si vous n’avez pas de champ createAt, updateAt il faut désactiver les “timestamps”.

- Créer une action dans le config_actions.json

```
  },
   "GET/companiesModel": {
       "controler": "selectAllFromModelSQL",
       "view": "companiesSQL"
  },
```

- Puis ajouter une lien dans la barre de navigation

```
<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">CountrieSQL</a></li>
        <li><a href="/countrySQL/FR">Country by code SQL</a></li>
        <li><a href="/companiesSQL">CompaniesSQL</a></li>
        <li><a href="/companySQL/7">Company by _id SQL</a></li>
        <li><a href="/companiesModel">Company by Model</a></li>
    </ul>
</nav>
```

- **Si vous avez l'erreur suivante :**
  - CreateAt and modifyAt not exist, vous devez modifier la configuration de l'objet sequelize dans appdyn.js (ajoutez le code en bleu)

```
// configuration des paramètres de la connexion SQL Sequelize
var sequelize = new Sequelize('gretajs', 'steph', 'azerty', {
    host: 'localhost',
    dialect: 'postgres',
    define: {
        timestamps: false
    },

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

- **Voyons comment réaliser des jointures avec Sequelize :**
  - Il existe dans les API des fonctions pour réaliser des relations entre les tables :
    - Lorsque vous appelez une méthode telle que **Companies.belongsTo (Countries)**, vous dites que le modèle **Companies** (le modèle **sur lequel la fonction est appelée**) est la source et le modèle **Countries** (le modèle passé comme argument) **est la cible**.
    - Voyons un exemple de code de jointure entre les deux schémas. Pour ce faire nous allons créer un nouveau contrôleur que l'on va appeler :

      selectFromModelWithJoin.js

- **Contrôleur selectFromModelWithJoin.js**

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

/* GET liste of companies with link to countries. */
router.route('/').get(function(req, res, next) {
    if ((req.session.passport) && (req.session.passport.user != null)) {

        GLOBAL.modelsSeq["Companies"].findAll({
            include: [{
                model: GLOBAL.modelsSeq["Countries”],
                keyType: GLOBAL.db.Sequelize.INTEGER
            }]
        }).then(function(datas) {
             console.log(‘Data from Sequelize Model Jointure : ‘, datas);
             res.render(req.message.view, {
                 title: 'List from SQL postgreSQL',
                 result: datas
            });
        });

    } else res.redirect('/');
});
module.exports = router;
```

Node, Express, MongoDB et Bases SQL

- **Modifier appdyn.js pour rendre global la variable Sequelize et sequelize via une variable que l'on appelera db :**

```
/** todo : ici initialiser la connexion à la base postgreSQL via Sequelize */
// Chargement du module sequelize
GLOBAL.db = {};
var Sequelize = require("sequelize");
db.Sequelize = Sequelize;

GLOBAL.modelsSeq = {};
// configuration des paramètres de la connexion SQL Sequelize
var sequelize = new Sequelize('gretajs', 'steph', 'azerty', {
    host: 'localhost',
    dialect: 'postgres',
    define: {
        timestamps: false
    },

    pool: {
        max: 5,
        min: 0,
        idle: 10000
    }
});
db.sequelize = sequelize;
```

Node, Express, MongoDB et Bases SQL

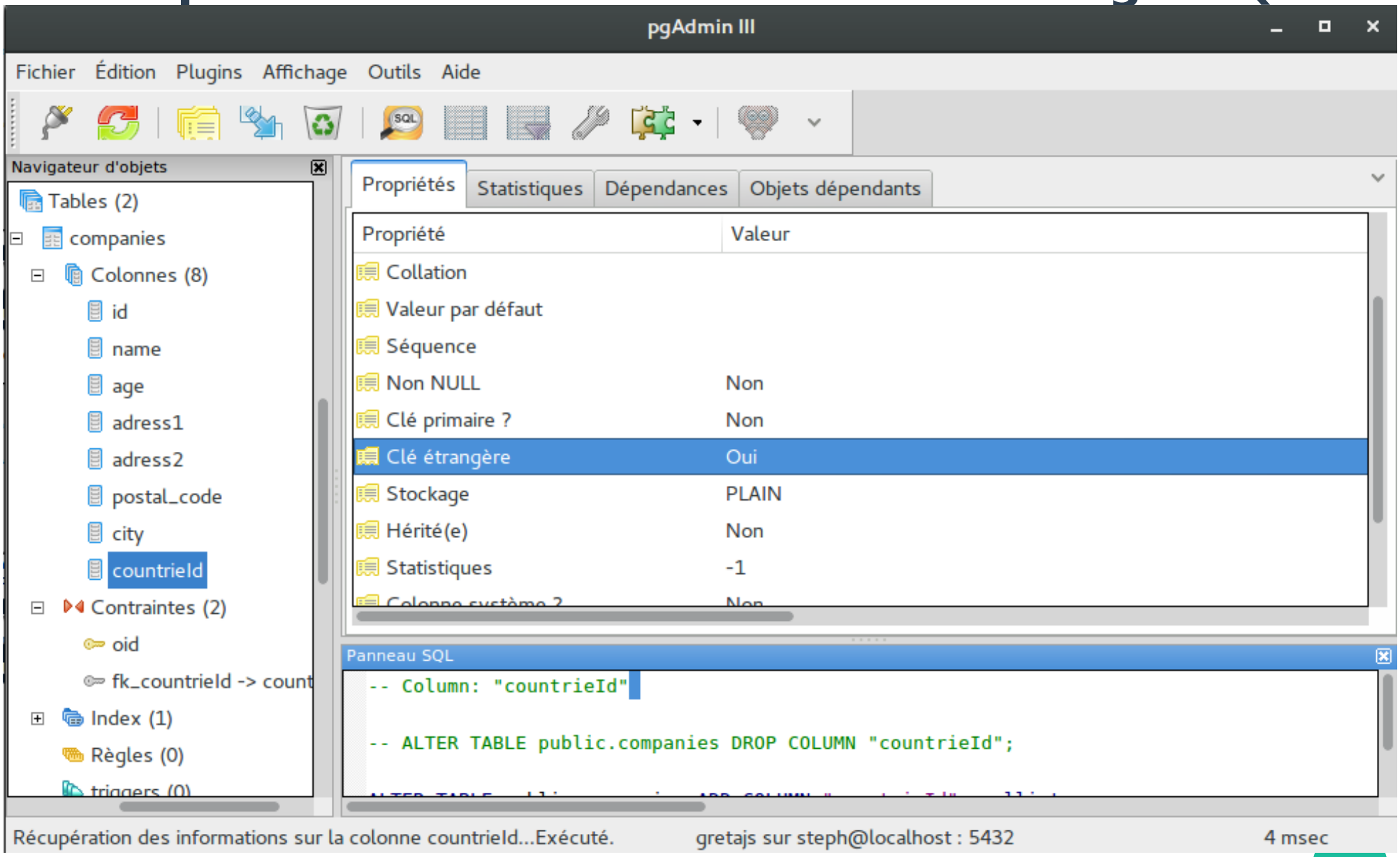
- **Il nous faut créer la relation avec les outils de Sequelize, on va dans appdyn.js ajouter du code :**

```
(…)
// Loader Sequelize models into GLOBAL.db
fs.readdirSync(__dirname + '/models')
    .filter(function(file) {
// Pour exclure du chargement un fichier dans le dossier && (file !== "index.js");
        return (file.indexOf(".") !== 0);
    })
    .forEach(function(file) {
        var model = sequelize.import(path.join(__dirname + '/models', file));
        db[model.name] = model;
        console.log('file read : ' + file);
    });
// CREATION DE L'ASSOCIATION
GLOBAL.modelsSeq["Companies"].belongsTo(GLOBAL.modelsSeq["Countries"], {
    foreignKey: "countrieId",
    keyType: GLOBAL.db.Sequelize.INTEGER
});
(…)
```

NB : Cela suppose que vous avez bien une clé étrangère dans companies nommée : “countrieId”

Node, Express, MongoDB et Bases SQL

- **Propriété de la base de données PostgreSQL :**



- **Remarques :**
  - Il ne faut pas qu'il y ait dans les schémas Sequelize de clés étrangères pour que les associations créées par les fonctions comme belongsTo(), hasOne(), ...
  - Soit vous supprimez les déclarations de champs de clés étrangères dans les schémas soit vous les placer en commentaires :

```
(…)
        city: {
            type: DataTypes.TEXT,
            allowNull: true
        }
        /*,countrieId: {
                    type: DataTypes.INTEGER,
                    allowNull: true,
                    references: {
                        model: 'countries',
                        key: 'id'
                    }
        }*/
    }, {
        tableName: 'companies'
    });
};
```

- **Ressources :**
  - Deux projets sur github pour récupérer des codes sources qui implémente les exemples :

    Avec le driver natif node-mongodb

    - https://github.com/Stephanux/gretaexos.git

    Avec Mongoose à la place du driver node-mongodb

    - https://github.com/Stephanux/gretaexos2.git

- **Allez plus loin :**
  - Je vous conseille de regarder le code source d'un Framework Node.js basé sur cette philosophie de factorisation et de fichier de configuration JSON.
    - https://github.com/huiledecode/OTF-