Les bases du développement MEAN stack

IB Formation - Lille Juin 2018

Julien Noyer

2018-06-25

Table des matières

Introduction

Lorsque l'on aborde la **MEAN stack** il faut avant tout faire preuve de persévérance et d'une certaine endurance car les outils utilisés pour finaliser un projet en **MEAN stack** sont en pleine évolution et donc absolument pas stabilisés. Et je crois personnellement qu'ils ne sont pas voués à l'être mais plutôt à suivre au fur et à mesure l'évolution des Internets dont l'avenir et plus qu'imprévisible Donc pourquoi se lancer dans la **MEAN stack** ? Est-ce que toute cette communauté qui utilise ces outils est masochiste au point d'apprendre des notions qui seront sans doute obsolètes dans 1 an ? Et bien la réponse est simple : nous sommes développeurs, nous sommes chercheurs, nous sommes passionnés par les nouvelles technologies. Et plus que toutes ces raisons, depuis les début du Web les technologies n'ont fait qu'évoluer, passant de Flash à Wordpress, nous devons suivre les nouveaux usages pour

produire des solutions qui correspondent a leur temps. La **MEAN stack** répond au besoin de pouvoir mettre en place un environnement structuré qui nous permettent d'évoluer dans notre métier, les quelques lignes qui suivent vont tenter de vous en persuadé.

A quoi ça sert le Javascript ?

« A faire de jolies animations et éviter le rechargement des pages. », d'accord mais pas d'accord : parlons-nous de **Javascript** ou de **jQuery** ? J'ai adoré cette bibliothèque à une certaine époque, elle permet de se former sur la gestion des callbacks et ne serait-ce que pour ça il faut lui rendre hommage, c'est quand même open-source ! Mais **Javascript** n'est pas **jQuery** et bien que nous condamnons ses positions politiques, c'est à **Brendan Eich** qu'il faut rendre hommage pour la création de **Javascript**, pas à **Netscape**. Le plus intéressant dans dans cette histoire et de savoir la raison de la création de **Javascript** et l'endroit dans lequel il était initialement prévu : Eich voulait que sont serveur soit plus rapide, donc il à développé son langage pour cette usage. **Netscape** apparaissait et pour développé son navigateur à eu la bonne idée d'embaucher Eich. C'est à ce moment là que le **Javascript** et passé cotés client. L'histoire permet souvent de comprendre le présent, et certaine fois donner des pistes pour l'avenir : **Javascript** est passé du serveur au client pour revenir au serveur mais en conservant sa maîtrise du client. Quel autre langage Web dont l'apprentissage est accessible à tous permet à les mêmes capacités ?

Présentation de la MEAN stack

La traduction la plus juste du verbe mean en français serait « vouloir dire » et je crois que je pourrais terminer ma présentation sur ça. Par acquis de conscience je me doit de prévenir le lecteur que mon interprétation de la **MEAN stack** peut paraitre dissonante et le rassurer sur le fait que ce document présente bien des techniques précises sur les 4 fameux frameworks. Je suis plongé depuis plus de 2 ans dans cette stack et au-delà d'avoir apprit beaucoup de notions spécifiques aux frameworks, elle m'à permit de m'ouvrir à des nouveaux principes de développement et à ne plus avoir peur d'aucuns frameworks, s'il est en **Javascript**, donnez moi deux semaines et je l'intègre à la **MEAN stack**. La mise en place de l'environnement de travail peut être très compliqué selon votre machine, les mises à jours sont fréquentes et dans 6 mois votre code risque de changer en vous mettant dans ce mode de fonctionnement vous répondez aux besoins de l'industrie en vous amusant à apprendre : vous devenez agile.

Environnement de travail

J'écris c'est ligne depuis un MacBook Pro ce qui me pousse à conseiller d'utiliser Linux comme OS dans tous les cas, et si vous préférez dépenser beaucoup d'argent, un Mac et peu importe le modèle (un vieux coucou d'occasion fera l'affaire). Linux parce que de toute façon, un jour ou l'autre, il faut si mettre. Que ce soit pour installer un package ou configurer un VPN, Linux sera le langage commun que vous devrez utiliser. Mac par ce que j'ai connu beaucoup de PC et pas mal d'échecs à l'installation de l'environnement sur des postes de travail Windows et même si je connais des techniques infaillibles, ça reste assez compliquer d'utiliser l'invite de commande… Le cœur des ordinateur Apple est en Linux et la console aussi, ce qui simplifie l'apprentissage. Mais sans vouloir insister, un bon vieux Debian sur un Raspberry Pi et vous le rendrez vert de jalousie. La documentation qui suit propose principalement des solutions pour les OS Apple et Windows, pour Linux j'attends vos propositions avisées pour mettre à jour ce document !

iTerm

Pour optimiser la console de mon Mac, j'utilise **iTerm** parce qu'il permet une plus grande souplesse dans la configuration : colorisation, fenêtre multiple, intégration de git, … Je suis dessus depuis 2018 et je remercie chaleureusement Kevin Loiseleur de me l'avoir présenté ! L'installation ce fait en téléchargeant un DMG sur le site et à en croire mes amis sur Windows, **Cmder** est plus professionnel que l'invite de commande.

PostMan

Lorsqu'il s'agit de tester des routes API, il est préférable d'utiliser un outil comme **PostMan** pour gérer ses requêtes. Honnêtement je n'en connais pas d'autre, j'ai découvert **PostMan** en premier et je n'ai pas changé depuis. Une des options intéressante est par exemple de pouvoir exporter ses requêtes pour les partager avec les membre de sont équipe, c'est très pratique en phase de développement.

NodeJS et NPM

Installation Apple et Windows

Pour les OS « mainstream », vous devez vous rendre sur le site officiel de **NodeJS** pour télécharger la dernière version de **NodeJS** pour votre OS. Lancez l'installateur et suivez tranquillement la procédure, pour Windows il faut souvent redémarrer pour les autres c'est bon !

Installation Linux

On est bien sur Linux, le code qui suit provient sur site **OpenClassroom** :

```
curl-sL https ://deb.nodesource.com/setup_8.x |sudo-Ebash-
sudo apt install -y nodejs
```

Tester votre installation

Une fois installé, vous pouvez taper la commande `node` et `npm` dans votre terminal et tester par exemple le cotre suivant :

```
node -v
npm -v
```

> La version de `node` et de `npm` s'affiche.

MongoDB

Pour que la stack soit complète il faut utiliser **MongoDB**, bien que d'autres solutions de gestions de l'informations peuvent être utilisés. Avec des résultats souvent plus performants.

Installation Windows

Allez sur le site **MongoDB** pour télécharger la version de ****MongoDB** Community Edition** pour télécharger l'installeur est suivez les étapes.

A la fin de l'installation vous devez pouvoir trouver dans votre dossier `Program Files` un dossier `MongoDB` contenant des exécutables qui vous permettent de lancer le serveur `mongod` et le shell `mongo`. Si vous ne trouvez pas ce dossier, vous devez recommencer l'installation et sélectionner l'installation **« custom »** pour sélectionner la racine de votre disque.

Tester votre installation

Une fois installé, vous devez avoir accès au dossier d'installation de **MongoDB** dans lequel vous trouverez les exécutables `mongo.exe` et `mongod.exe`. Double-cliquez sur les deux pour tester votre installation.

> La suite de l'utilisation de **MongoDB** est abordée dans la section tutoriel.

Installation Apple

L'installation la plus simple pour Apple est sans aucuns doutes **Homebrew**, si ce n'est pas déjà fait, ouvrez votre terminal et tapez la commande suivante :

```
1 /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
    ↪ Homebrew/install/master/install)"
```

Une fois **Homebrew** installé - mais aussi pour les autres - mettez à jour **Homebrew** avant de continuer :

```
1 brew update
```

Vous pouvez maintenant installer **MongoDB** avec la commande suivante :

```
1 brew install mongodb
```

Tester votre installation

Une fois installé, vous pouvez taper la commande `mongod` et `mongo` dans votre terminal la commande :

```
1 mongod
```

Et la commande suivante dans un autre terminal :

```
1 mongo
```

> La suite de l'utilisation de **MongoDB** est abordée dans la section tutoriel.

Angular CLI

Nous allons à présent installer une interface de ligne de commande (**CLI**) pour **Angular** pour développer plus rapidement notre application client **Angular CLI**. Ouvrez un terminal et tapez la commande suivante, vous devez être administrateur de votre machine, si la commande ne fonctionne pas, préfixez-la avec `sudo` :

```
1 npm install -g @angular/cli
```


> La suite de l'utilisation de **Angular CLI** est abordée dans la section tutoriel.

Visual Studio Code

Pour finir il faut choisir un éditeur de code, depuis mais premiers pas sur **AngularJS**, j'ai choisi d'utiliser **Visual Studio Code** pour son interface et son accessibilité. J'ai converti pas mal de collègues qui sont conquis à leur tours !

QuickStart MEAN stack

A cette étape, vous devez être d'attaque pour vous lancer dans la **MEAN stack**. Pour tout vous dire, je pense que vous faire tout installer dès le début me permet d'âtre sûr que vous êtes motivés pour continuer ce tutoriel. Le travail qui suit va vous permettre d'aborder toutes les notions qui créées la **MEAN stack** en produisant une application complète qui vous servira de base pour les prochaines.

Etape 1 : Serveur NodeJS

Je le pense pas que la **MEAN stack** soit « disruptive » de quoi que ce soit mais plutôt un nouvel outil. C'est pourquoi je par du principe que le point de départ d'un projet qui stock des informations est le serveur qui les gère. C'est pourquoi nous allons commencer par créer un serveur **NodeJS** que nous ferons évoluer tout au long de ce tutoriel.

Configuration

Pour configurer un nouveau projet vous devez dans un premier temps créer et initier un nouveau dossier `npm` et `git` :

```
mkdir path/to/myNodeServer

cd path/to/myNodeServer

git init && npm init
```

> Lors de la commande `npm init` remplacer l'entré `main` par `server.js`.

Installation des dépendances

Il faut ensuite installer deux dépendances, dont voici les définitions :

- **ExpressJs** : framework pour manipuler le serveur NodeJS
- **NodeMon** : commande permettant de mettre à jour le serveur en cas de modification
- **Body-Parser** : middleware permettant d'extraire les données d'une requête serveur
- **Path** : middleware permettant de définir des adresses de dossiers statiques
- **Ejs** : système de templating en Javascript

Vous pouvez ajouter ces deux dépendance de cette manière :

```
npm install express nodemon body-Parser path --save
```

> La commande `--save` permet d'ajouter les dépendance dans le fichier **`package.json`**.

Configuration du package.json

Une fois les dépendance installées, vous devez modifier le fichier **`package.json`** pour intégrer `nodemon` dans la commande `npm start` de la manière suivante :

```
"start" : "nodemon server.js"
```

> Vous pouvez utiliser **NodeMon** dans tous vos projets en l'installant en global.

Votre fichier **`package.json`** doit contenir à présent au minimum les informations suivantes :

```
{
  "name" : "myNodeServer",
  "version" : "1.0.0",
  "main" : "server.js",
  "scripts" : {
    "start" : "nodemon server.js"
  },
  "license" : "MIT",
  "dependencies" : {
    "express" : "^<VERSION>",
    "nodemon" : "^<VERSION>",
    "body-parser" : "^<VERSION>",
    "path" : "^<VERSION>",
    "ejs" : "^<VERSION>"
  }
}
```


> Vous pouvez ajouter des information en suivants les recommandations NPM.

Création d'un serveur NodeJS

Pour configurer votre serveur NodeJS vous devez créer un document nommé `server.js` à la racine de votre dossier `myNodeServer` :

```
cd path/to/myNodeServer

touch server.js
```

Ce fichier contiendra le code pour configurer votre serveur NodeJS :

```
/*
Imports and configuration
*/
  //=> Extern dependencies
  const express = require('express');
  const bodyParser = require('body-parser');
  const path = require('path');
  const ejs = require('ejs');

  //=> Express
  const port = process.env.PORT || 8080;
  const appServer = express();
//

/*
Server initialisation
*/
  const init = () => {
    //=> Ready to listen
    appServer.listen(port, () => console.log(`App is running on port ${
        ↪ port}`) );
  };

  //=> Launch server
  init();
//
```

Une fois le fichier `server.js` enregistré, vous devez ouvrir un invité de commande à la racine du dossier `myNodeServer` pour lancer le serveur :


```
cd path/to/myNodeServer

npm start
```

> Si votre serveur est bien configuré votre invité de commande affiche `Le serveur est lancé sur le port 8080`

Configuration du serveur pour la gestion des routes

Les routes de notre server **NodeJS** ont la même utilités que n'importe quelles routes sur n'importe quelles serveurs, les principes de bases sont les mêmes. Nous allons dans un premier temps utiliser le module **Path** pour définir les dossier statique des vues front.

La variable `path` que nous avons créé sur le serveur permet de définir l'adresse du dossier static de la partie `front`. Vous devez créer un dossier nommé `www` contenant un fichier `index.html` à la racine de votre dossier `myNodeServer` :

```
cd path/to/myNodeServer

mkdir client && cd client

touch index.html
```

Vous devez ensuite ajouter dans le fichier `server.js` l'instruction suivante :

```
/*
Server initialisation
*/
    //=> Use path to add views
    appServer.engine( 'html', ejs.renderFile );
    appServer.set( 'view engine', 'html' );
    appServer.set( 'views', __dirname + '/www' );
    appServer.use( express.static(path.join(__dirname, 'www')) );

    //=> Ready to listen
    ...
//
```

> Nous configurerons plus tard le fichier `index.html` du dossier `client`.

La variable `bodyParser` permet de récupérer les données d'une requête serveur, vous devez le configurer de la façon suivante dans le fichier `server.js` :

```
/*
Server initialisation
*/
    //=> Body Parser
    appServer.use(bodyParser.json({limit : '10mb'}));
    appServer.use(bodyParser.urlencoded({ extended : true }));

    //=> Use path to add views
    ...
//
```

Création des routeurs

Première fonctionnalité de notre Quickstart, la gestion des routes nous permettra de définir ce que le serveur doit renvoyer au client selon l'adresse qui sera appelée. Nous allons commencer par créer un dossier à la racine du serveur et un fichier pour dans ce dossier pour organiser les routes :

```
mkdir routes && cd routes

touch main.router.js
```

Le fichier `main.router.js` nous permet de référencer les différentes class qui gérerons nos routes selon la définition d'un `path` spécifique. Nous allons commencé par gérer la route `front` et la route `api` de la façon suivante :

```
/*
Imports
*/
    const { Router } = require('express');
    const FrontRouterClass = require('./front/front.routes');
    const AuthRouterClass = require('./auth/auth.routes');
//

/*
Define globale routers
- The "mergeParams : true" enable to parse parameters true routers class
*/
    const mainRouter = Router({ mergeParams : true });
```



```
    const apiRouter = Router({ mergeParams : true });
    mainRouter.use('/api/', apiRouter);
//

/*
Define specific routers
*/
    const frontRouter = new FrontRouterClass();
    mainRouter.use('/', frontRouter.init());

    const authRouter = new AuthRouterClass();
    apiRouter.use('/auth', authRouter.init());
//

/*
Export
*/
    module.exports = { mainRouter };
//
```

Ce code configure deux routes : `/` pour afficher les client front et `/api/auth` pour authentifier un utilisateur. Avant de créer les fichier qui sont référencés dans `main.router.js`, nous allons intégrer notre routeur principal dans le fichier `server.js` :

```
/*
Imports and configuration
*/
    ...

    //=> Routes
    const { mainRouter } = require('./routes/main.router');

    //=> Express
    ...
//

/*
Server initialisation
*/
  const init = () => {
    ...

    //=> Server mainRouter
```



```
    appServer.use('/', mainRouter);

    //=> Ready to listen
    ...
  };
//
```

Nous reviendrons régulièrement sur le fichier `server.js` pour ajouter la configuration nécessaire à nos nouveaux composants. A présent, créez dans le dossier `routes` des dossiers pour gérer les routes `auth` et `front` :

```
cd routes

mkdir auth && touch auth/auth.routes.js
mkdir front && touch front.routes.js
```

Commençons pas configurer la route nous permettant d'afficher un client de base, ouvrez le fichier `front.routes.js` et ajoutez-y le code suivant :

```
/*
Imports
- The "mergeParams : true" enable to parse parameters true routers class
*/
    const express = require('express');
    const frontRouter = express.Router({ mergeParams : true });
//

/*
Definition
*/
    class FrontRouterClass {

        constructor() {}

        routes(){
            // Get all paths from "/"
            frontRouter.get( ['/*'], (req, res) => { res.render('index'
                ↪ ) });
        };

        init(){
            this.routes();
            return frontRouter;
```

```
        }
    }
//

/*
Export
*/
    module.exports = FrontRouterClass;
//
```

Pour finir la route `front`, il vous suffit d'ajouter un code de base dans le fichier `index.html` du dossier `www` :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
            ↪ scale=1.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">
        <title>MEAN Quickstart</title>
    </head>

    <body>
        <h1>Hello MEAN</h1>
    </body>
</html>
```

Nous n'allons pas tester tout de suite notre première route, encore un peu de patience ! Le route `front` est simple dans la mesure ou elle n'à pas réellement besoin d'un contrôleur, dans notre cas tout du moins. Un contrôleur dans une route nous permet de séparer les logiques métiers ce qui assure une meilleure compréhension et une meilleure stabilité du code.

Ouvrez à présent le fichier `auth.routes.js` pour commencer la configuration de la route `/api/auth` :

```
/*
Imports
- The "mergeParams : true" enable to parse parameters true routers class
*/
    const express = require('express');
    const authRouter = express.Router({ mergeParams : true });
//
```



```
8
9  /*
10 Definition
11 */
12 class AuthRouterClass {
13
14     constructor({ passport }) { this.passport = passport };
15
16     routes(){
17
18         // Route AUTH
19         authRouter.post('/', (req, res) => {
20             res.json(`HATEAOS for api/auth`)
21         })
22     };
23
24     // Initialize routes
25     init(){
26         this.routes();
27         return authRouter;
28     };
29 };
30 //
31
32 /*
33 Export
34 */
35     module.exports = AuthRouterClass;
36 //
```

Pour le moment, la seul différence entre les routes `front` et `auth` et la façon dont nous renvoyons la réponse : dans la mesure ou **NodeJS** nous servira principalement à créer une api, nous renvoyons des données au format **JSON**.

Vous pouvez à présent tester la route `front` dans votre navigateur à l'adresse `http://localhost:8080` et la route `auth` sur Postman en mode `GET` l'adresse suivante `http://localhost:8080/api/auth`.

Création des services pour les routeur

Avant d'aller plus loin dans le création des routes, nous allons mettre un place un principe de modularisation pour l'envoie des réponses serveur et pour l'analyse des valeurs envoyées par un client vers le

serveur. Nous allons commencé par créer un dossier `services` à la racine du serveur pour un fichier `server.reponse.js` dans ce dossier :

```
cd path/to/myNodeServer

mkdir service && touch services/server.reopnse.js
```

Vous pouvez maintenant ouvrir le fichier `server.reopnse.js` pour y ajouter le code suivant :

```
/*
Service methodes
*/
    const sendApiSuccessResponse = (response, successMessage, data) =>
        ↪ {
        return response.status(200).send({
            message : successMessage,
            err : null,
            data : data,
        })
    }

    const sendApiErrorResponse = (response, errorMessage, error) => {
        return response.status(500).json({
            message : errorMessage,
            error,
            data : null,
        });
    }
//

/*
Export service fonctions
*/
    module.exports = {
        sendApiSuccessResponse,
        sendApiErrorResponse
    };
//
```

Le principe de cette méthode et de permettre une plus grande souplesse dans la gestion des retours de notre API, il sera possible par exemple d'inclure dans chacune des fonctions la gestion des logs ou des retours dans la console. Une fois ce service créé, nous allons l'utiliser dans le route `auth` de la manière suivante :


```
/*
Imports
- The "mergeParams : true" enable to parse parameters true routers class
*/
    const express = require('express');
    const authRouter = express.Router({ mergeParams : true });

    // Add service
    const { sendApiSuccessResponse, sendApiErrorResponse } = require('
        ↪ ../../services/server.response');
//

/*
Definition
*/
    class AuthRouterClass {

        constructor({ passport }) { this.passport = passport };

        routes(){

            // Route AUTH
            authRouter.post('/', (req, res) => {
                // Use service to send response
                sendApiSuccessResponse(res, `HATEAOS for api/auth`,
                    ↪ null)
            })
        };

        // Initialize routes
        init(){
            this.routes();
            return authRouter;
        };
    };
//

/*
Export
*/
    module.exports = AuthRouterClass;
//
```


Nous devons à présent créer un service essentiel pour mettre en place une logique forte et stable. Lorsqu'un utilisateur va s'inscrire sur l'application, il devra remplir un formulaire contenant des champs spécifiques. Cotés serveur nous devrons récupérer ces champs grâce à **Body-Parser** mais avant de tenter des les enregistrer dans notre base de données, nous devons nous assurer que les champs nécessaires sont bien envoyés. Le service que nous allons créé à présent à cette utilité, il bloque l'utilisateur tant qu'il n'à pas envoyé les bons champs de tel sorte à ne jamais tenter d'inscrire des données dans la base avant d'être sur qu'elle soient présente. Créons à présent le fichier `request.checker.js` dans le dossier `services` :

```
cd path/to/myNodeServer/service && touch request.checker.js
```

Ouvrez le fichier `request.checker.js` pour y ajouter le code suivant :

```
/*
Service methodes
*/
const checkFields = (requiredFieldsArray, bodyParams) => {
    const missedFields = [];
    const extraFields = [];

    // Check missing fields
    requiredFieldsArray.forEach((prop) => {
      if (!(prop in bodyParams)) missedFields.push(prop);
    });

    // Check extra fields
    for (const prop in bodyParams) {
      if (requiredFieldsArray.indexOf(prop) === -1) extraFields.push(
          ↪ prop);
    }

    // Check if there's extra or missing fields
    const ok = (extraFields.length === 0 && missedFields.length === 0);

    // Send objects response
    return { ok, extraFields, missedFields };
  };
//

/*
Export
*/
```



```
29    module.exports = {
30      checkFields,
31    };
32  //
```

Nous pouvons à présent importer ce service dans le fichier `auth.routes.js` de la même façon que le service précédent :

```
1  /*
2  Imports
3  - The "mergeParams : true" enable to parse parameters true routers class
4  */
5      ...
6
7      // Add service
8      const { checkFields } = require('../../services/request.checker');
9  //
```

Toujours dans le fichier `auth.routes.js` nous allons à présent voir l'utilisation de notre service en créant une nouvelle route en `POST` dans la fonction `routes()` pour inscrire un utilisateur :

```
1  // Route AUTH register user
2  authRouter.post('/register', (req, res) => {
3      // Check if body is present
4      if (typeof req.body === 'undefined' || req.body === null) {
           ↪ sendBodyError(res, 'No body data provided') }
5
6      // Check mandatory fields
7      const { miss, extra, ok } = checkFields(['firstName', 'lastName', '
           ↪ email', 'password'], req.body);
8
9      // Check the result
10     if (!ok) {
11         // Error
12         return sendFieldsError(res, 'Bad fields provided', miss, extra)
13     }
14
15     else{
16         // Success
17         return sendApiSuccessResponse(res, 'Provided fields are ok',
               ↪ null)
18     }
19 })
```


Nous indiquons dans cette route que les champs `firstName`, `lastName`, `email` et `password` sont obligatoire et renvoyons un message spécifique en cas d'erreur pu de succès. Vous pouvez tester cet route sur **PostMan** en mode `POST` l'adresse suivante `http ://localhost :8080/api/auth /register`.

> Pensez à bien configurer l'envoie des données en **x-www-form-urlencoded**.

Créer un contrôleur pour la route auth

Les services que nous avons créé nous permettent de nous assurer que les données envoyées par le client sont bien conformes à celles nécessaires pour l'ajout d'un document dans notre collection **MongoDB**, nous devons à présent créer un contrôleur qui nous permettra d'enregistrer les données envoyées. La première étape pour créer ce contrôleur et de définir un model objet qui va nous permettre de structurer les données avant des les enregistrer.

Pour créer notre model `UserModel`, nous allons utiliser le module **Mongoose** qui permet justement de structurer nos données de façon plus efficace, par exemple grâce au fait les les propriétés des modèles seront typées. Ouvrez un terminal, placez-vous à la racine du serveur pour installer **Mongoose** et créer le dossier `models` avec un fichier `user.model.js` à l'intérieur :

```
cd path/to/myNodeServer

npm i -s mongoose

mkdir models && touch models/user.model.js
```

Ouvrez à présent le fichier `user.model.js` et ajoutez le code suivant :

```
/*
Imports
*/
    const mongoose = require('mongoose');
    const { Schema } = mongoose;
//

/*
Definition
*/
    const UserSchema = new Schema({
        firstName : String,
        lastName : String,
        email : String,
```



```
15          password : String,
16          cgu : Boolean,
17      });
18  //
19
20  /*
21  Export
22  */
23      const UserModel = mongoose.model('user', UserSchema);
24      module.exports = UserModel;
25  //
```

> L'avantage de cette méthode est le fait que les modèles objet **Angular** sont similaires.

Le fait de créer notre model avec **Mongoose** nous permet de pouvoir utiliser des fonction sur les objets qui sont propres à **Mongoose** et qui permettent plus facilement d'ajouter, d'éditer ou de supprimer des documents dans une collection **MongoDB**.

Notre model est fait, nous pouvons maintenant créer le contrôleur qui va l'utiliser pour inscrire nos utilisateur, placez vous à la racine du serveur pour créer le fichier `auth.controller.js` dans le dossier de la route `auth` :

```
1  cd path/to/myNodeServer
2
3  touch routes/auth/auth.controller.js
```

Avant de partir bille en tête dans l'enregistrement des informations de nos utilisateurs, posons-nous la question de la valeur qu'elles ont pour eux. Le 25 mai 2018 est une date importante dans la mesure ou enfin, l'Europe à prit les devant dans la mise en place de règles de bons sens pour protéger ses concitoyens : les **RGPD**. Rien de bien nouveau pour le développeur Français qui connaissent les actions de la **CNIL** mais change beaucoup dans la mesure ou aujourd'hui, nous sommes responsable de la sécurité des informations que nous stockons pour nos clients, sur leur serveur.

Partant de ce constat que fait-on ? on stock tout en claire ou on crypte tout ? En matière de cryptage je suis jusqu'au-boutiste, et pour commencer, je considère comme une faute de stocker un mot de passe utilisateur en clair sur une base de données. Je vous présente donc **Bcrypt**, un module NPM très simple d'utilisation et que nous allons utiliser dans notre contrôleur. Ouvrez un terminal à la racine du serveur pour installer la dépendance :

```
1  npm i -s bcrypt
```

Ouvrez à présent le fichier `auth.controller.js` et ajoutez le code suivant :


```
/*
Imports
*/
    const UserModel = require('../../models/user.model');
    const bcrypt = require('bcryptjs');
//

/*
Methods
*/
    // Register new
    const registerUser = (bodyParams) => {
        return new Promise((resolve, reject) => {

            // Check if user already exist
            UserModel.findOne({ email : bodyParams.email }, (error, user
                ↪ ) => {
                if (error) { return reject(error) }
                else if(user){ return reject(`Email already used`) }

                else {
                    // Generate a hash of the password
                    bcrypt.hash(bodyParams.password, 10)
                    .then( hash => {
                        // Define user hashed password
                        const userPasswordHash = hash

                        // Save user
                        UserModel.create({
                            firstName : bodyParams.firstName,
                            lastName : bodyParams.lastName,
                            email : bodyParams.email,
                            password : userPasswordHash,
                            cgu : true,

                        }, (error, user) => {
                            // Check DDB result
                            return error ? return reject(error) :
                                ↪ resolve(user)
                        });
                    })
                    // Catch error when hashing password
```

```
                    .catch( error => reject(error))
                };
            });
        });
    };
//

/*
Export
*/
    module.exports = {
        registerUser,
    };
//
```

Avant d'enregistrer l'utilisateur nous vérifions sont adresse mail, de base nous considérons que les adresse mail sont unique, puis nous créons à un hash du mot de passe et enregistrons les données dans la collection `users` de notre base de données **MongoDB**.Pour finir nous exportons la fonction dans un objet car nous ajouterons d'autres fonctions à exporter.

Nous allons utiliser la fonction `registerUser()` dans le fichier `auth.routes.js`, ouvrez-le et modifiez le de la façons suivante :

```

/*
Imports
*/
...

const { registerUser } = require('./auth.controller');
//

/*
Definition
*/
    class AuthRouterClass {

        constructor({ passport }) { this.passport = passport };

        routes(){
            // Route AUTH register user
            authRouter.post('/register', (req, res) => {
                ...
```



```
21
22                  else{
23                      // Use controller
24                      registerUser(req.body)
25                      .then( apiRes =>  sendApiSuccessResponse(res, 'User
                            ↪  is registrated', apiRes))
26                      .catch( apiErr => sendApiErrorResponse(res, 'Error
                            ↪ during user registration', apiErr));
27                  }
28              })
29          }
30      }
31  //
32  ...
```

DAns un premier temps nous importons la fonction `registerUser()` depuis le contrôleur `auth.controller'.js` puis nous modifions la condition **`else`** pour utiliser la fonction `registerUser()` en lui envoyant le `body` de la `requéte` que nous avons vérifié plus haut en paramètre.

Votre requête `http ://localhost :8080/api/auth` sur **PostMan** doit vous donner les bon résultats, si ce n'est pas le cas vous pouvez au choix reprendre ce tutoriel depuis le début ou télécharger la branch `etape1` sur le répertoire **MEANquickstart**.

Si votre **PostMan** vus donne le sourire, **bravo !** Passez vite à l'étape 2, on va fouiller dans **MongoDB** pour voir comment développer des « idées géantes » !

Etape 2 : Système de gestion de l'information

Si je commençais par vous dire qu'il est fort probable que **MongoDB** n'est sans doute pas la meilleure solution pour votre projet et que vous devriez utiliser le module **mysql** parce que vous avez des connaissance plus fortes en **SQL** et que la plupart de vos clients utilisent ce langage pour gérer leurs informations ? Les développeur **BackEnd** seront rassurés et les **FrontEnd** vont couiner. La **MEAN stack** comme je l'ai imaginé permet de combler les attentes des deux communautés de développeurs dans le mesure ou elle est adaptive et évolutive.

Mes recherches sur la gestion de la donnée ma amener à penser, grâce à **Damien Truffaut** et **Giuseppe Militello** des amis que vous devriez rencontrer, que la donnée n'est rien et que l'information est tout. Avant de choisir un système de gestion il est primordial d'analyser l'information pour en comprendre sont sens et sa portée pour savoir comment la stocker et comment la redistribuer.

SI vous chercher à synchroniser rapidement des données centralisées sur un grand nombre de supports, avant de vous lancer dans **MongoDB**, allez faire un tours du cotés de **CouchDB** et **PouchDB**,

vous m'en direz des nouvelles. Il est probable même que vous ne reveniez plus voir ce tutoriel… Allez, les passionnés, avec moi !

Lancer le server de bases de données MongoDB

Dans un premier temps nous devons lancer un serveur qui rendra disponible nos base de données **MongoDB** car comme n'importe quel système de gestion de l'information il faut un serveur spécifique pour l'héberger.

La commande mongod sur Windows

Vous devez vous rendre dans le dossier d'installation de **MongoDB** dans lequel vous trouverez un exécutable `mongod.exe`, double-cliquez dessus pour lancer le server de base de données `mongod`.

La commande mongod sur Linux et Apple

La commande `mongod` doit être installer dans le path de votre terminal, ouvrez-en un pour taper la commande suivante :

```
1 sudo mongod
```

Lancer le shell MongoDB

Le Shell de **MongoDB** va nous permettre dans un premier temps d'exécuter des commandes pour prendre en main la logique de **MongoDB** pour pouvoir plus facilement comprendre l'implémentation dans la **MEAN stack**.

La commande mongod sur Windows

Vous devez vous rendre dans le dossier d'installation de **MongoDB** dans lequel vous trouverez un exécutable `mongo.exe`, double-cliquez dessus pour lancer le shell `mongod`.

La commande mongod sur Linux et Apple

La commande `mongo` doit être installer dans le path de votre terminal, ouvrez-en un pour taper la commande suivante :

```
1 sudo mongo
```


Les bases de données MongoDB

Comme n'importe quel système de gestion de l'information, **MongoDB** fonctionne avec des bases de données. L'originalité des systèmes **NoSQL** c'est que les commandes sont implicites. C'est à dire que la commande pour créer une base de données est la même que celle pour indiquer d'en utiliser une :

```
use mean-quickstart
```

Il est possible de voir la liste des collections de la base de données en tapant la commande suivante :

```
show dbs
```

Pour connaître le nom de la base de données utilisée il suffi de taper la commande :

```
db
```

Les collections MongoDB

MongoDB fonctionne sur le principe de documents et de collection d'objets : un document est un ensemble de données au format **JSON** et une collection est l'équivalent d'une table. Comme pour les bases de données, la création d'une collection est implicite mais il est possible d'en créer des vide avec la commande suivante :

```
db.createCollection('users"')
```

Les documents MongoDB

Les documents en **NoSQL** sont des objets **Javascript** classiques, pour les créer en **MongoDB** il faut utiliser la fonction `insert()` sur la collection désirée.

Créer un document

```
db.posts.insert({
    firstName : "Abdel",
    lastName :"Strauss",
    email :"abdel@strauss.eu",
    password : "doneversaveaclearpassword",
    age : 43,
    cgu : false,
```



```
    skills : [”Javascript”, ”CSS”, ”HTML”]
})
```

> Pour ajouter plusieurs objets il suffi des les inscrire dans un tableau

Rechercher des documents

Pour rechercher des documents il faut utiliser la fonction `find()` de la façon suivante :

```
db.myCollection.find({
    isDone  : false
})
```

Mettre à jour un document

Pour mettre à jour un document il faut utiliser la fonction `update()` de la façon suivante :

```
db.myCollection.update({
    password :”doneversaveaclearpassword”
    },{
        $set :{
            cgu  : true
        }
    })
```

> Le deuxième objet passé en paramètre permet de définir les propriétés à mettre à jour. La fonction `update` permet également d’ajouter des propriétés à un objet.

Supprimer un document

Pour supprimer un document il faut utiliser la fonction `remove()` de la façon suivante :

```
db.myCollection.remove({
    password :”doneversaveaclearpassword”
})
```

> Il est conseiller d’utiliser les ID des objets car sinon toutes les occurrences seront supprimées.

Rechercher dans une collection

La fonction find sur une collection de données permet d'afficher et de filtrer les résultats d'une recherche. Nous allons à présent explorer les options qu'il est possible d'utiliser avec la fonction find(). Pour commencer les tests ci-dessous ; il est recommander de vider votre collection posts à l'aide de la commande drop() pour d'y ajouter de nouveaux documents en tapant la commande suivante :

```
1 db.posts.insert([
2     { "type" : "IMG", "title" : "Une image de sport", "content" : "http
      ↪ ://lorempixel.com/400/200/sports", "tags" : [ "sport", "image
      ↪ " ], "data" : { "author" : "Julien Noyer", "state" : "ONLINE", "
      ↪ likes" : 10 } },
3     { "type" : "QUOTE", "title" : "Lorem ipsum dolor ismet", "content"
      ↪ : "", "tags" : [ "lorem", "image" ], "data" : { "author" : "
      ↪ John Doe", "state" : "DRAFT", "likes" : 0 } },
4     { "type" : "VID", "title" : "The Gladiators", "content" : "
      ↪ P8BKRCpVoug", "tags" : [ "rasta", "video" ], "data" : { "
      ↪ author" : "Julien Noyer", "state" : "DRAFT", "likes" : 0 } },
5     { "type" : "IMG", "title" : "Lorem ipsum dolor ismet", "content" :
      ↪ "http ://lorempixel.com/400/200/people", "tags" : [ "lorem", "
      ↪ image" ], "data" : { "author" : "Carla Santa", "state" : "ONLINE
      ↪ ", "likes" : 30 } },
6     { "type" : "IMG", "title" : "Une image de chat", "content" : "http
      ↪ ://lorempixel.com/400/200/cat", "tags" : [ "chat", "image" ],
      ↪ "data" : { "author" : "John Doe", "state" : "ONLINE", "likes" :
      ↪ 20 } }
7 ])
```

Rechercher parmi les documents

Pour n'afficher que les objets qui correspondent à certaines clefs-valeurs, il faut les indiquer en paramètre de la fonction :

```
1 db.posts.find({"type" :"IMG"})
```

> Il est possible d'ajouter d'autres paramètres dans l'objet : { type :« IMG », title :« … » }

Sélectionner les valeurs à afficher

Il est possible de n'afficher qu'une partie des propriétés des objet rechercher. Il faut ajouter un deuxième objet à la suite du premier dans la fonction `find()` et de préciser les paramètres à afficher

dans le résultat :

```
db.posts.find({"type" :"IMG"}, {title : 1, _id : 0})
```

> Si une seule propriété est défini à 1, toutes les autres sont à 0 sauf _id

Rechercher dans un tableau

Comme pour les champs simple, comme `title` dans nos exemple, il est possible de rechercher dans le tableau de données de nos documents. Mais nous pouvons également définir deux options pour afficher plusieurs combinaisons :

```
db.posts.find({"tags" :{$in :["lorem", "video"]}})
```

> Pour que les deux options n'en forme qu'une seule il faut utiliser `$all` à la place de `$in`

Rechercher dans un objet

Il est également possible de rechercher parmi les propriétés d'un objet situé dans un document, il suffi d'utiliser la méthode objet classique de la façon suivante :

```
db.posts.find({"data.author" : "John Doe"})
```

> Il est possible d'ajouter d'autres paramètres dans la recherche : { « data.author » : « John Doe », « tags » : « lorem » }

Rechercher les valeurs supérieures ou égal à…

Les opérateurs de comparaisons (>, !=, &&, …) n'existent pas dans MongoDB, il est néanmoins possible de filtrer les valeurs supérieures à x de la façon suivante :

```
db.posts.find({"data.likes" : {$gte :20}})
```

Les autres opérateurs de comparaison s'utilisent de la même manière et sont défini de la façon suivante - $gt : plus grand que - $lt : plus petit que - $gte : plus grand ou égal à - $lte : plus petit ou égal à

Trier les résultats

Il est possible de classer dans un ordre croissant ou décroissant un résultat en utilisant la fonction `sort()` de la façon suivante :

```
db.posts.find({"data.likes" : {$gte :20}}, {data :1}).sort({"data.likes"
    ↪  :-1})
```

> Pour afficher les résultats dans un ordre croissant il faut définir la valeur à `1`

Créer, modifier et supprimer un document

Différentes fonctions existent pour manipuler les documents d'une collection, pour les utiliser il suffi de sélectionner la bonne collection de données et d'appliquer les fonctions en respectant leur syntaxe.

Créer un document

Comme vu précédemment, la création d'un document ce fait avec la fonction `insert()` de la façon suivante :

```
db.posts.insert(
    {
        "type"  : "IMG",
        "title"  : "Une image de sport",
        "content"  : "http ://lorempixel.com/400/200/sports",
        "tags"  : [
            "sport",
            "image"
        ],
        "data" : {
            "author" : "Julien Noyer",
            "state" : "ONLINE",
            "likes" : 10
            }
        }
    )
```

> Pour ajouter plusieurs objets il suffi de les inscrire dans un tableau

Ajouter/Modifier une propriété

La fonction `update()` permet de mettre à jour un ou plusieurs objet de la même manière que la fonction `find()`. Il faut appliquer la fonction `update()` en précisant le ou les documents à sélectionner et les propriétés à mettre à jour de la façon suivante :

```
db.posts.update(
    { "type" : "IMG" },
    { $set :{ "state" : "PENDING" } }
)
```

> Pour modifier plusieurs propriétés il suffi de la ajouter dans l'objet `$set` et pour modifier plusieurs objet, il faut ajouter l'option `{ multi : true }` en troisième paramètre de la fonction `update()`

La même fonction `update()` permet également d'ajouter une propriété à un objet, il suffi de le définir en paramètre de la fonction pour qu'il s'ajoute à l'objet.

Supprimer une propriété

Pour supprimer une propriété il faut utiliser la fonction `uptdate()` en utilisant un objet `$unset` qui permet de supprimer la ou les propriété de la façon suivante :

```
db.posts.update(
    { "type" : "IMG" },
    { $unset : { "state" : 1 } }
)
```

> Les options sont paramétrable de la même manière que précédemment

Ajouter une valeur dans un tableau

MongoDB permet l'utilisation d'un `$push` sur un tableau qu'il faut appliquer en paramêtre de la fonction 'update() de la façon suivante :

```
db.posts.update(
    { "type" : "QUOTE" },
    { $push : { "tags" : "ipsum" } }
)
```


> Les options sont paramétrable de la même manière que précédemment

Pour s'assurer qu'il n'y ai pas de doublon dans le tableau, il faut utiliser le paramètre `$addToSet` à la place de `$push` :

```
db.posts.update(
    { "type" : "QUOTE" },
    { $addToSet : { "tags" : "ipsum" } }
)
```

Supprimer un document

Pour rappel, supprimer un document se fait avec la fonction `remove()` de la façon suivante :

```
db.myCollection.remove(
    { title : "Une image de chat" }
)
```

> Il est conseiller d'utiliser les ID des objets car sinon toutes les occurrences seront supprimées

Etape 3 : Client Front

L'application que nous avons créée jusqu'à maintenant nous a permit de faire le point sur les grands principes de la **MEAN stack**. De NodeJS à ExpressJs pour le serveur, à MongoDB pour la gestion des bases de données, nous pouvons passer à présent à une phase nous permettant d'ajouter le « A » à notre application : configurer l'application **Angular** dans la partie client. Nous passons donc au développement MEAN à proprement parlé, l'association de ces quatre frameworks JavaScript au sein d'une seule et même application front et back.

Ajouter une application Angular en tant que client

Vous aller créer un nouveau projet Angular en utilisant un commande **Angular CLI**, ouvrez un terminal puis taper la commande suivante

```
cd path/to/myNodeServer

ng new ANGclient
```

A la fin de l'exécution de cette commande, un projet Angular complet - incluant le dossier `node_modules` - sera présent dans le nouveau dossier `client`. Vous devez à présent vous

rendre dans ce dossier pour compiler l'application Angular, nous afficherons ensuite les fichiers compilés en modifiant le `path` de la vue front dans le fichier `api.js` situé dans le dossier `routes` du serveur.

```
cd ANGclient

ng build -prod --output-path ../www --watch
```

> Nous ajoutons le drapeau `--watch` pour relancer la compilation en cas de modification de l'application Angular.

Nous allons à présent mettre en place les éléments principaux de l'application Angular. Nous allons créer les différents composants de l'application grâce à l'interface de ligne de commande et les configurerons au fur et à mesure.

Création du composant PAGE

Notre application **Angular** sera constituée de plusieurs pages que nous aurons besoin de configurer dans un système de routing. Les solutions proposer par **Angular** pour gérer les routes sont diverses, nous partons du principe pour notre projet que chaque composant qui constituent mes pages doivent être chargés de façon asynchrone pour ne pas ralentir le chargement de l'application dès l'ouverture de la première page.

Ouvrez un terminal à la racine du projet **Angular** pour générer un nouveau composant :

```
cd path/to/myNodeServer/ANGclient

ng g c pages/homePage
```

Puis ouvrez le fichier `home-page.component.ts` pour y ajouter le code suivant :

```
/*
Imports
*/
  import { Component, OnInit } from '@angular/core';
  import { UserModel } from "../../models/user.model";
  import { AuthService } from "../../services/auth/auth.service";
//

/*
Definition
*/
```



```
12    @Component({
13      selector : 'app-home-page',
14      templateUrl : './home-page.component.html',
15    })
16  //
17
18  /*
19  Export
20  */
21    export class HomePageComponent implements OnInit {
22
23      // Variables
24      public registerObject : UserModel = {
25          firstName : ``,
26          lastName : ``,
27          email : ``,
28          password : ``,
29          repeatePassword : ``,
30          cgu : false
31      }
32
33      // Inject AuthService in the class
34      constructor(
35        private authService : AuthService
36      ) { }
37
38      // Create a function to register user
39      public registerNewUser = ( user : UserModel ) => {
40        console.log(`Validated form HOME`, user);
41
42        // Use the service to register user
43        this.authService.userRegister(user)
44        .then( apiSuccess => console.log(apiSuccess) )
45        .catch( apiError => console.error(apiError) )
46
47      };
48
49      ngOnInit() {}
50    }
51  //
```

Nous allons à présent créer un module et un routing pour cette page afin de pouvoir la charger de

façon asynchrone :

```
cd path/to/myNodeServer/ANGclient/src/app/pages/home-page

touch touch home-page.routes.ts && home-page.module.ts
```

Ouvrez le fichier `home-page.routes.ts` pour y ajouter le code suivant :

```
/*
Import
*/
    import { ModuleWithProviders } from '@angular/core';
    import { Routes, RouterModule } from "@angular/router";

    import { HomePageComponent } from "./home-page.component";
//

/*
Definition
*/
    const routes : Routes = [
        {
            path : '',
            component : HomePageComponent
        }
    ]
//

/*
Export
*/
    export const ComponentRouter : ModuleWithProviders = RouterModule.
        ↪ forChild(routes)
//
```

Puis ouvrez le fichier `home-page.module.ts` pour y ajouter le code suivant :

```
/*
Imports
*/
    import { NgModule } from "@angular/core";
    import { ComponentRouter } from "./routes";
    import { AppFormModule } from "../../shared/form-modules/form.
```



```
        ↪ module";
    import { HomePageComponent } from "./home-page.component";
//

/*
Definition
*/
    @NgModule({
        declarations : [ HomePageComponent ],
        imports : [ ComponentRouter, AppFormModule ]
    })
//

/*
Export
*/
    export class HomePageModule {};
//
```

C'est trois étapes sont à suivre pour chaque création de page dans votre application **Angular**, en modifiant bien évidemment le code du contrôleur. Le fichier `home-page.routes.ts` permet de définir le composant à afficher et le fichier `home-page.module.ts` permet d'ajouter des composants spécifiques à la page et de rendre disponible le fichier de routing de la page dans le fichier de routing général de l'application que nous créerons ensuite.

Création d'un model objet

Angular utilise la sur-couche de **Javascript** que l'on appel **TypeScrypt** qui permet comme sont nom l'indique de typer notre code **Javascript**. Nous allons tout au long de notre exploration de **Angular** exploiter ce principe comme par exemple dans le fichier `home-page.component.ts` qui fait référence à un model objet à la ligne :

```
import { UserModel } from "../../models/user.model";
```

Nous allons à présent créer ce model dans un dossier model, ouvrez un terminal et tapez le code suivant :

```
cd path/to/myNodeServer/ANGclient/src/app/

mkdir model && touch models/user.model.ts
```


Ouvrez le fichier `user.model.ts` pour y ajouter le code suivant :

```
/*
Object models in TypeScript are Interfaces
*/
    export interface UserModel {
        firstName : String,
        lastName : String,
        email : String,
        cgu : Boolean,
        password : String,
        repeatePassword? : String,
        _id? : String,
    }
//
```

> Comme vous pouvez le contacter, le model **Angular** est très proche du model **Mongoose**

Création d'un service

Notre model `UserModel` sera utilisé dans plusieurs fichiers ce qui nous permettra de profiter des avantages du typage en **TypeScrypt**. Nous pouvons passez à présentent à la création d'un service, nous allons configurer celui qui est chargé dans le fichier `home-page.component.ts` :

```
import { AuthService } from "../../services/auth/auth.service";
```

Un service en **Angular** et l'équivalent d'un contrôleur pour les routes **NodeJS** : ils vont stocker les fonctions qui interrogent les routes du API du serveur. Ouvrez un terminal dans le dossier de l'application pour générer un service :

```
cd path/to/myNodeServer/ANGclient

ng g s services/auth/auth
```

Ouvrez à présent le fichier `auth.service.ts` et ajoutez-y le code suivant :

```
/*
Imports
*/
  import { Injectable } from '@angular/core';
  import { HttpClient, HttpHeaders } from "@angular/common/http";
  import { UserModel } from "../../models/user.model";
```



```
7  //
8
9  /*
10 Definition
11 */
12   @Injectable({
13     providedIn : 'root'
14   })
15 //
16
17 /*
18 Export
19 */
20   export class AuthService {
21
22     // Define API address
23     private apiUrl = '/api/auth'
24
25     // Inject HttpCLient into the class
26     constructor(
27       private http : HttpClient
28     ) { }
29
30     // Function to register user
31     public userRegister( user : UserModel ) : Promise<any> {
32
33       // Delete repeatePassword property from the user object (for
          ↪ checkFields)
34       delete user.repeatePassword;
35
36       // Header configuration
37       let myHeader = new HttpHeaders().set( 'Content-Type', '
          ↪ application/json')
38
39       // Make an HTTP GET call
40       return this.http.post(`${this.apiUrl}/register`, user, { headers :
          ↪  myHeader })
41       .toPromise().then(this.getData).catch(this.handelError)
42     }
43
44     // Fonction to parse SUCCESS response
45     private getData( response : any ){
46       // return res || {};
```

```
47        return Promise.resolve( response || {} );
48      }
49
50      // Fonction to parse ERROR response
51      private handelError( response : any ){
52        return Promise.reject( response.error );
53      }
54    }
55  //
```

Création du router global

Tous les éléments de notre page `home-page` sont créés, nous allons à présent rendre accessible cette page en créant le fichier global de routing de l'application. Placez-vous à la racine de l'application **Angular** pour créer le fichier `app.routing.ts` :

```
1  cd path/to/myNodeServer/ANGclient/src/app
2
3  touch app.routing.ts
```

Ouvrez le fichier `app.routing.ts` et ajoutez le code suivant :

```
1  /*
2  Import
3  */
4      import { ModuleWithProviders } from '@angular/core';
5      import { Routes, RouterModule } from "@angular/router";
6  //
7
8  /*
9  Define APP routes
10 */
11     const mainRoutes : Routes = [
12         {
13             path : '',
14             loadChildren : './pages/home-page/module#HomePageModule' //
                   ↪ Lazy Load
15         }
16     ];
17 //
18
19 /*
```

```
20 Export
21 */
22     export const AppRouter : ModuleWithProviders = RouterModule.forRoot(
       ↪ mainRoutes);
23 //
```

Comme pour la page `home-page`, pour rendre accessible ce fichier de routing, vous devez l'ajouter dans le fichier `app.module.ts`, ouvrez-le et remplacer le code par le suivant :

```
1  /*
2  Import
3  */
4    // Angular components
5    import { BrowserModule } from '@angular/platform-browser';
6    import { NgModule } from '@angular/core';
7    import { HttpClientModule } from '@angular/common/http';
8
9    // APP components
10   import { AppRouter } from "./app.router"; // AppRouter est un module
11   import { AppComponent } from './app.component';
12 //
13
14 /*
15 Definition
16 */
17   @NgModule({
18     declarations : [ AppComponent ],
19     imports : [
20       BrowserModule,
21       HttpClientModule,
22       AppRouter
23     ],
24     providers : [ ],
25     bootstrap : [ AppComponent ]
26   })
27 //
28
29 /*
30 Export
31 */
32   export class AppModule { }
33 //
```


> Nous ajoutons également le module `HttpClientModule` pour permettre les appels HTTP dans l'application.

Pour finir, nous allons changer le contenu du fichier `app.component.ts` pour y mentionner la directive `router-outlet` qui permet de charger les composants des routes :

```
1  /*
2  Import
3  */
4    import { Component } from '@angular/core';
5  //
6
7  /*
8  Definition
9  */
10   @Component({
11     selector : 'app-root',
12     template : `
13       <router-outlet></router-outlet>
14     `
15   })
16
17 /*
18 Export
19 */
20   export class AppComponent {}
21 //
```

> A cette étape vous pouvez supprimer les fichier `app.componen.css` et `app.componen.html` que nous n'utilisons plus.

La première page de votre application est maintenant terminée ! Nous en ajouterons plusieurs dans l'étape 4 mais vous pouvez dès à présent tester votre application dans votre navigateur !

Création du module user-interface

Nous allons à présent créer notre premier module qui va référencer tous les composants relatifs à l'interface utilisateur et les rendre disponible dans les différentes page de notre application. Ouvrez un terminal et tapez les commande suivantes :

```
1  cd path/to/myNodeServer/ANGclient/src/app
2
```



```
mkdir shared/user-interface && touch shared/user-interface/user-
    ↪ interface.module.ts
```

Vous pouvez à présent créer le composant `my-header` dans le dossier `user-interface` :

```
cd path/to/myNodeServer/ANGclient/

ng g c shared/user-interface/myHeader -is --skip-import
```

Ouvrez le fichier `user-interface.module.ts` pour y ajouter le code suivant :

```
/*
Imports
*/
    // Angular components
    import { NgModule } from "@angular/core";
    import { CommonModule } from "@angular/common"; // To use global
        ↪ Angular common directives
    import { RouterModule } from "@angular/router";

    // Intern components
    import { MyHeaderComponent } from "./my-header/my-header.component"
        ↪ ;
//

/*
Definition
*/
    @NgModule({
        // Import the components
        declarations : [ MyHeaderComponent ],
        imports : [ CommonModule, RouterModule ],

        // Export the components to enable there access from main
            ↪ compoenent
        exports : [ MyHeaderComponent ]
    })
//

/*
Export
*/
    export class UserInterfaceModule {};
```



```
30 //
```

Nous venons de référencer notre nouveau composant `MyHeaderComponent` ainsi que deux modules spécifiques : `CommonModule` nous permet d'utiliser les directives globales de **Angular** dans notre module et `RouterModule` pour pouvoir gérer des redirection dans les composants importés dans `user-interface.module.ts`.

Ouvrez à présent le fichier `my-header.component.ts` pour y ajouter le code suivant :

```
1  /*
2  Imports
3  */
4    import { Component, OnInit, Input } from '@angular/core';
5    import { Router } from "@angular/router";
6  //
7
8  /*
9  Definition
10 */
11   @Component({
12     selector : 'app-my-header',
13     templateUrl : './my-header.component.html',
14     providers : [ UserService ]
15     styles : [`
16       ul{ display : flex; list-style : none; }
17       li :not( :last-child){ margin-right : 1rem; }
18     `]
19   })
20 //
21
22 /*
23 Export
24 */
25   export class MyHeaderComponent implements OnInit {
26
27     // The @Input() decorateur enable to input data to the component
          ↪ from the main component
28     @Input() path : string;
29     @Input() headerTitle : string;
30
31     constructor(
32       private router : Router
33     ) { }
```

```
34
35      ngOnInit() {}
36    }
37  //
```

Vous pouvez maintenant éditer le fichier `my-header.component.html` avec le code suivant :

```
1  <header>
2    <h1>{{headerTitle}}</h1>
3    <nav>
4      <ul *ngIf="path === '/todo'">
5        <li><a [routerLink]="'/todo'">Todo page</a></li>
6        <li (click)="logoutUser()">Logout</li>
7      </ul>
8    </nav>
9  </header>
```

La base de notre header est faite, nous devons maintenant le rendre accessible en important le module `user-interface.module.ts` dans le module de la page `home page`, ouvrez le fichier `module.ts` situé dans le dossier `home-page` pour y ajouter le module `user-interface.module.ts` de la façon suivante :

```
1  /*
2  Imports
3  */
4      // Angular components
5      import { NgModule } from "@angular/core";
6      import { ComponentRouter } from "./routes";
7
8      // APP components
9      import { AppFormModule } from "../../shared/form-modules/form.
           ↪ module";
10
11     //=> Import the user-interface.module
12     import { UserInterfaceModule } from "../../shared/user-interface/
           ↪ user-interface.module";
13
14     import { HomePageComponent } from "./home-page.component";
15 //
16
17 /*
18 Definition
19 */
```

```
20      @NgModule({
21          declarations : [ HomePageComponent ],
22
23          //=> Add the UserInterfaceModule in the iports array
24          imports : [ ComponentRouter, UserInterfaceModule ]
25      })
26  //
27
28  /*
29  Export
30  */
31      export class HomePageModule {};
32  //
```

Vous pouvez terminer cette mise en place en ajoutant dans le fichier `home-page.component.html` le code suivant :

```
1  <!-- Use the my-header.component.ts -->
2  <app-my-header
3      [path]="'/'"
4      [title]="'Welcome on the home page'"
5
6  ></app-my-header>
```

Gestion des formulaires

Dernière phase de l'étape 3, nous devons à présent créer des formulaires pour inscrire et connecter un utilisateur. Nous allons partir du même principe que pour le module `UserInterface` en créant dans un premier temps un dossier `form-module` avec un fichier `form.module` dans le dossier `shared` avec les commandes suivantes :

```
1  cd path/to/myNodeServer/ANGclient/src/app
2
3  mkdir shared/form-module && touch shared/form-module/form.module.ts
```

Vous pouvez à présent créer le composant `register-form` dans le dossier `form-module` :

```
1  cd path/to/myNodeServer/ANGclient/
2
3  ng g c shared/form-module/registerForm -is --skip-import
```

Ouvrez le fichier `form.module.ts` pour y ajouter le code suivant :


```
/*
Imports
*/
    import { NgModule } from "@angular/core";
    import { CommonModule } from "@angular/common";
    import { FormsModule } from '@angular/forms'; // To use [(ngModel)]
    import { RegisterFormComponent } from "./register-form/register-
       ↪ form.component";
//

/*
Definition
*/
    @NgModule({
        declarations : [ RegisterFormComponent ],
        imports : [ CommonModule, FormsModule ],
        exports : [ RegisterFormComponent ]
    })
//

/*
Export
*/
    export class AppFormModule {};
//
```

Nous venons de référencer notre nouveau composant `RegisterFormComponent` ainsi que deux modules spécifiques : `CommonModule` nous permet d'utiliser les directives globales de **Angular** dans notre module et `FormsModule` pour pouvoir utiliser la directive **NgModel** dans notre formulaire.

Ouvrez à présent le fichier `register-form.component.ts` pour y ajouter le code suivant :

```
/*
Imports
*/
  import { Component, OnInit, Input, Output, EventEmitter } from '
     ↪ @angular/core';
  import { UserModel } from "../../../models/user.model";
//

/*
Definition
```

```
10 */
11   @Component({
12     selector : 'app-register-form',
13     templateUrl : './register-form.component.html',
14     styles : []
15   })
16 //
17
18 /*
19 Export
20 */
21   export class RegisterFormComponent implements OnInit {
22     @Input() formObject : UserModel;
23
24     // The @Output() decorator enable component to send data to main
       ↪ component with an event
25     @Output() sendFormData = new EventEmitter();
26
27     // Create a varible to toggle the form
28     public formIsOpen : Boolean = false;
29
30     // Create an object for errors
31     public formError;
32
33     constructor() { }
34
35     // Create a function to reset form error
36     private resetFormError = () => {
37       this.formError = {
38         score : 0,
39         firstName : false,
40         lastName : false,
41         email : false,
42         password : false,
43         repeatePassword : false,
44         cgu : false
45       }
46     }
47
48     // Create a function to manipulate checkbboxe
49     public checkConditions = () => {
50       // Invers value
51       this.formObject.cgu = ! this.formObject.cgu;
```



```
52
53        // Hide error
54        this.formError.cgu = false
55      }
56
57      // Creta a function for the form submission
58      public formSubmission = () => {
59        // Reset errors
60        this.resetFormError()
61
62        // Test mandatory fields : firstName
63        if( this.formObject.firstName.length <= 1 ){
64          this.formError.score ++;
65          this.formError.firstName = true
66        }
67
68        // Test mandatory fields : lastName
69        if( this.formObject.lastName.length <= 1 ){
70          this.formError.score ++;
71          this.formError.lastName = true
72        }
73
74        // Test mandatory fields : email
75        if( this.formObject.email.length <= 1 ){
76          this.formError.score ++;
77          this.formError.email = true
78        }
79
80        // Test mandatory fields : password
81        if( this.formObject.password.length <= 4 ){
82          this.formError.score ++;
83          this.formError.password = true
84        }
85
86        // Test mandatory fields : repeatePassword
87        if( this.formObject.password != this.formObject.repeatePassword )
              ↪ {
88          this.formError.score ++;
89          this.formError.repeatePassword = true
90        }
91
92        // Test mandatory fields : CGU
93        if( !this.formObject.cgu ){
```

```
94          this.formError.score ++;
95          this.formError.cgu = true
96        }
97
98        /*
99        Finale validation
100       */
101         this.formError.score === 0 ? this.sendFormData.emit( this.
              ↪ formObject )  : null;
102       //
103
104     }
105
106     ngOnInit() {
107       // Reset errors when component is loaded
108       this.resetFormError()
109     }
110
111   }
112 //
```

Les techniques mises en place dans ce fichiers permettent er vérifier chacun des champs nécessaires via un objet `formError`. Nous allons à présent éditer le fichier `register-form.component.html` de la façon suivante :

```
1  <!--
2  Use the *ngIf to show the section when formObject data is receive
3  -->
4  <section *ngIf="formObject">
5    <h2 (click)="formIsOpen = !formIsOpen">Inscription</h2>
6    <!--
7    Use the NgModel to make a tow way data binding
8    -->
9    <form (submit)="formSubmission()"  [ngClass]="{ 'open' : formIsOpen }"
       ↪  >
10     <label for="firstName">Votre prénom <span *ngIf="formError.
         ↪ firstName">Obligatoire</span></label>
11     <input type="text" name="firstName" [(ngModel)]="formObject.
         ↪ firstName" (focus)="formError.firstName = false">
12
13     <label for="lastName">Votre nom  <span *ngIf="formError.lastName">
         ↪ Obligatoire</span></label>
14     <input type="text" name="lastName" [(ngModel)]="formObject.lastName
```



```
        ↪ " (focus)="formError.lastName = false">
15
16      <label for="email">Votre adresse mail  <span *ngIf="formError.email
        ↪ ">Obligatoire</span></label>
17      <input type="email" name="email" [(ngModel)]="formObject.email" (
        ↪ focus)="formError.email = false">
18
19      <label for="password">Votre mot de passe  <span *ngIf="formError.
        ↪ password">Obligatoire</span></label>
20      <input type="password" name="password" [(ngModel)]="formObject.
        ↪ password" (focus)="formError.password = false">
21
22      <label for="repeate-password">Répéter votre mot de passe  <span *
        ↪ ngIf="formError.repeatePassword">Différent</span></label>
23      <input type="password" name="repeate-password" [(ngModel)]="
        ↪ formObject.repeatePassword"  (focus)="formError.
        ↪ repeatePassword = false">
24
25      <input type="checkbox" name="cgu" [(ngModel)]="formObject.cgu" (
        ↪ click)="checkConditions()">
26      Accepter les CGU <span *ngIf="formError.cgu">Obligatoire</span>
27
28      <button type="submit">Inscription</button>
29    </form>
30  </section>
```

La grande nouveauté dans ce document est l'utilisation des **NgModel**, cette directive propose à **Angular** et permet d'envoyer un objet dans le formulaire pour alimenter ses propriété avec la valeurs des inputs. Nous allons voir à présent comment utiliser ce composant et tout deviendra plus clair.

Ouvrez le fichier du module de la page `home page` pour y ajouter le module `form.module.ts` de la façon suivante :

```
1  /*
2  Imports
3  */
4      // Angular components
5      import { NgModule } from "@angular/core";
6      import { ComponentRouter } from "./routes";
7
8      // APP components
9      import { AppFormModule } from "../../shared/form-modules/form.
           ↪ module";
```



```
10      import { UserInterfaceModule } from "../../shared/user-interface/
            ↪ user-interface.module";
11
12      //=> Import the form.module
13      import { AppFormModule } from "../../shared/form-modules/form.
            ↪ module";
14
15      import { HomePageComponent } from "./home-page.component";
16  //
17
18  /*
19  Definition
20  */
21      @NgModule({
22          declarations : [ HomePageComponent ],
23
24          //=> Add the UserInterfaceModule in the iports array
25          imports : [ ComponentRouter, UserInterfaceModule, AppFormModule
                ↪ ]
26      })
27  //
28
29  /*
30  Export
31  */
32      export class HomePageModule {};
33  //
```

Une fois le module ajouté, nous allons utiliser le composant `RegisterFormComponent` dans le fichier `home-page.component.html` :

```
1  <app-register-form
2      [formObject]="registerObject"
3      (sendFormData)="registerNewUser($event)"
4  >
```

Etape 4 : Finalisation

Publier une application MEAN stack

Configurer un VPN

Digital Ocean est un service de cloud computing qui fourni au développeurs d'applications une solution de monitoring de serveurs virtuels. D'autres solutions sont disponibles mais nous choisissons **Digital Ocean** car il fait parti des solutions les plus sollicitées par les développeurs d'applications.

Créer un compte sur Digital Ocean

Vous devez dans un premier temps créer votre compte sur la plateforme - vous pouvez profiter d'une offre de 10$ en cliquant sur ce lien. Remplissez le formulaire et suivez les différentes étapes de validation pour rendre votre compte actif. Une fois activé, vous pouvez accéder à la partie `Droplets` pour configurer votre VPN.

Créer un droplet

Un droplet Digital Ocean est un serveur virtuel dont l'accès est sécuriser en SSH. Plusieurs options sont disponible, nous choisirons la configuration suivante :

- One-click apps : NodeJS (dernière version)
- Size : 5$/mo
- Datacenter : Amsterdam

> Cette configuration est suffisante pour notre projet mais vous pouvez en choisir une plus appropriée si votre projet en à besoin.

Configurer un utilisateur SSH

Une fois votre droplet créé, vous aurez la possibilité de vous connecter dessus en SSH via votre invité de commande. La première étape de la configuration de votre VPN consiste à créer un utilisateur en lui attribuant les droit d'accès `sudo` afin de pouvoir supprimer l'utilisateur `root` ajouté au droplet automatiquement à la création (cette opération est permet de sécuriser votre VPN). Ouvrez une fenêtre d'inviter de commande et taper la commande suivante :

```
1 ssh root@ip_du_droplet
```


> Vous devez accepter la connexion en tapant `yes` puis `ENTER` et renseigner le mot de passe que vous aurez reçu par mail. Lors de votre première connexion il vous sera demandé de créer un nouveau mot de passe pour le compte `root`.

Une fois cette commande exécuté vous devez renseigner le mot de passe du profil `root` que vous avez reçu à la création de votre droplet. En cas de succès, vous êtes connecté en tant qu'administrateur sur votre VPN, vous pouvez donc taper les commandes suivantes pour créer un nouvel utilisateur :

```
1 adduser server-admin
```

> A la fin de cette commande il vous sera demandé de définir un mot de passe et de le confirmer. Les autres informations demandées sont optionnelles.

Une fois ce nouvel utilisateur créé, vous devez lui donner les droit `sudo` pour qu'il devienne administrateur du serveur en tapant la commande suivante :

```
1 usermod -aG sudo server-admin
```

> Pour vérifier le statut de l'utilisateur, taper la commande `id server-admin` (sudo doit se trouver dans groups).

Vous pouvez maintenant changer de profil utilisateur en tapant la commande suivante :

```
1 su - server-admin
```

> Pour vérifier votre identité tapez la commande `whoami` (`server-admin` doit s'afficher dans la console)

Ajouter une clés de connexion SSH

Pour assurer un minimum de protection sur votre votre VPN, vous allez associer au compte que vous avez créer une clés SSH. Cette clés vous permettra ensuite de vous connecter automatiquement à votre VPN sans renseigner votre mot de passe. Vous allez tout d'abord créer un dossier qui recevra la clés SSH :

```
1 mkdir ~/.ssh
2
3 chmod 700 ~/.ssh
4
5 nano ~/.ssh/authorized_keys
```


> Ces commandes créées un dossier nommé `.ssh` configuré en écriture puis créées et ouvrent un fichier `authorized_keys` dans l'éditeur `nano`.

Une fois `nano` ouvert, vous devez y indiquer votre clés ssh. Vous devez au préalable disposer d'une clés SSH sur votre machine, pour savoir si vous en avez une, ouvrez une nouvelle fenêtre d'invité de commande et tapez la commande suivante :

```
pbcopy < ~/.ssh/id_rsa.pub
```

> Cette commande affiche (ou copie) votre clés SSH.

Une fois la clés SSH copiée, retournez dans le fenêtre de l'éditeur `nano` (celle du serveur) et collez la clés SSH. Pour sortir et valider la modification du fichier `authorized_keys`, tapez les commandes suivantes :

```
// Sortir du fichier
ctrl + X

// Valider la modification
"Y"

// Quitter nano
ENTER
```

Si votre clés SSH est configurée correctement, vous devez modifier les droits d'accès au dossier `.ssh` pour empêcher toutes modifications du fichier `authorized_keys`.

```
chmod 600  ~/.ssh/authorized_keys
```

Ne devons à présent vérifier notre configuration en quittant le serveur et en nous re-connectant avec le nouvel utilisateur que nous avons créé. Tapez deux fois la commande `exit` puis testez votre connexion :

```
ssh server-admin@ip_du_droplet
```

> Si votre configuration est réussie votre mot de passe ne vous sera pas demandé.

En cas de problème dans votre configuration, vous devez supprimer le profil `server-admin` et re-commencer les étapes. La suppression d'un utilisateur ce fait de la façon suivante :

```
sudo userdel server-admin
```


Supprimer le compte root

Si votre connexion SSH est configurée correctement, vous pouvez à présent supprimer les droits d'accès du compte `root` pour ajouter une protection supplémentaire. Tapez la commande suivante pour éditer le fichier de configuration :

```
sudo nano /etc/ssh/sshd_config
```

Tapez les touches `ctrl` + `W` pour rechercher le terme `PermitRootLogin` et changer `yes` par `no`.

```
PermitRootLogin no
```

Pour sortir et valider la modification du fichier `sshd_config`, tapez les commandes suivantes :

```
// Sortir du fichier
ctrl + X

// Valider la modification
"Y"

// Quitter nano
ENTER
```

Pour que votre nouvelle configuration soit prise en compte, tapez la commande suivante pour redémarrer le serveur :

```
service ssh restart
```

Configurer du VPN avec un accès https gratuit (oui, oui, 100% gratuit)

Après avoir créer notre Droplet, nous pouvons à présent nous y connecté pour installer plusieurs dépendances que nous utiliserons pour exécuter notre application MEAN. La méthode suivante est inspiré d'une vidéo de **Jason Lengstorf** sur la configuration d'un VPN Digital Ocean.

Git et NodeJS

Avant toutes choses, nous allons installé Git et configurer NodeJS sur notre VPN. Connectez vous à votre serveur et tapez les commande suivantes :

```
sudo apt-get install git
```



```
2
3 curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
4
5 sudo apt-get install nodejs
```

> Ces commandes installent Git sur le serveur et mettent à jour NodeJS.

Configurer les modes de connexion

Nous allons à présent autoriser sur le serveur les connexions `OpenSSH`, `http`, `https` et activer le firewall.

```
1 sudo ufw allow OpenSSH
2
3 sudo ufw allow http
4
5 sudo ufw allow https
6
7 sudo ufw enable
```

> A la fin de ces commandes vous devez taper la touche `Y` puis `ENTER` pour valider les changements, vous pouvez vérifier la configuration en tapant la commande `sudo ufw status`.

Configurer l'accès https

Nous allons installer `Let's Encrypt` qui va nous permettre de configurer gratuitement un accès sécurisé `https` pour notre application. Lorsque vous êtes connecté au serveur, tapez les commandes suivante :

```
1 sudo git clone https://github.com/letsencrypt/letsencrypt /opt/
    ↪ letsencrypt
2
3 cd /opt/letsencrypt
```

> A la fin de ces commandes vous aurez installé `Let's Encrypt` dans un dossier et serrez placé à l'intérieur.

Une fois `Let's Encrypt` installé, nous pouvons configurer le nom de domaine en tapant la commande suivante :

```
1 ./letsencrypt-auto certonly --standalone
```


> Cette commande installe sur votre serveur le système de configuration d'adresse https de `Let's Encrypt`.

A la fin de l'installation, une fenêtre s'ouvre et vous demande de renseigner votre adresse mail, puis une seconde pour accepter les TOS et une dernière pour renseigner l'adresse de l'application. A la fin de ces trois étapes, tapez sur la touche `ENTER` pour valider votre adresse.

> Lorsque le certificat arrive à expiration vous pouvez le renouveler en tapant la commande `./letsencrypt-auto renew`.

Configurer le nom de domaine

Pour réaliser cette opération, vous devez disposer d'un nom de domaine qui pointe sur l'adresse IP du droplet. Nous devons à présent configurer la redirection des utilisateurs vers le répertoire de notre application. Pour réaliser cette configuration, nous allons utiliser NGINX en tant que proxy inverse. Pour installer NGINX tapez la commande suivante :

```
sudo apt-get install nginx
```

Nous pouvons à présent utiliser NGINX pour nous assurer que toutes les requêtes soient servies en SSH :

```
sudo nano /etc/nginx/sites-enabled/default
```

> Cette commande ouvre `nano` pour éditer le fichier `sites-enabled/default`.

Une fois dans `nano`, tapez les touche `CTRL` + `K` pour vider le document et copiez-collez le code suivant :

```
# HTTP — redirect all traffic to HTTPS
server {
    listen 80;
    listen [::]:80 default_server ipv6only=on;
    return 301 https://$host$request_uri;
}
```

> Pour sauvegarder le fichier taper les touche `CTRL + X`, `Y` pour enregistrer et `ENTER` pour valider.

Nous allons à présent crypter les requêtes grâce à la commande `dhparam` :

```
sudo openssl dhparam -out /etc/ssl/certs/dhparam.pem 2048
```


Nous devons maintenant créer un fichier de configuration SSL, tapez la commande suivante pour créer et ouvrir un fichier nommé `ssl-params.conf` :

```
sudo nano /etc/nginx/snippets/ssl-params.conf
```

Dans `nano`, copiez-collez le code suivant :

```
# See https://cipherli.st/ for details on this configuration
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
ssl_prefer_server_ciphers on;
ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
ssl_ecdh_curve secp384r1; # Requires nginx >= 1.1.0
ssl_session_cache shared:SSL:10m;
ssl_session_tickets off; # Requires nginx >= 1.5.9
ssl_stapling on; # Requires nginx >= 1.3.7
ssl_stapling_verify on; # Requires nginx => 1.3.7
resolver 8.8.8.8 8.8.4.4 valid=300s;
resolver_timeout 5s;
add_header Strict-Transport-Security "max-age=63072000;
    ↪ includeSubDomains; preload";
add_header X-Frame-Options DENY;
add_header X-Content-Type-Options nosniff;

# Add our strong Diffie-Hellman group
ssl_dhparam /etc/ssl/certs/dhparam.pem;
```

> Pour sauvegarder le fichier taper les touche `CTRL + X`, `Y` pour enregistrer et `ENTER` pour valider.

Pour finir la configuration, nous devons éditer le fichier `sites-enabled/default` pour que notre nom de domaine utilise le protocole SSL :

```
sudo nano /etc/nginx/sites-enabled/default
```

Une fois dans `nano`, ajoutez à la fin du fichier le code suivant :

```
# HTTPS — proxy all requests to the Node app
server {
    # Enable HTTP/2
    listen 443 ssl http2;
    listen [::]:443 ssl http2;
    server_name app.example.com;

    # Use the 'Lets Encrypt certificates
```



```
9       ssl_certificate /etc/letsencrypt/live/VOTRE-NOM-DE-DOMAIN.COM/
            ↪ fullchain.pem;
10      ssl_certificate_key /etc/letsencrypt/live/VOTRE-NOM-DE-DOMAIN.COM/
            ↪ privkey.pem;
11
12      # Include the SSL configuration from cipherli.st
13      include snippets/ssl-params.conf;
14
15      location / {
16          proxy_set_header X-Real-IP $remote_addr;
17          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
18          proxy_set_header X-NginX-Proxy true;
19          proxy_pass http://localhost:8080/;
20          proxy_ssl_session_reuse off;
21          proxy_set_header Host $http_host;
22          proxy_cache_bypass $http_upgrade;
23          proxy_redirect off;
24      }
25  }
```

> Penser à modifier le nom de domaine et pour sauvegarder le fichier taper les touche `CTRL + X`, `Y` pour enregistrer et `ENTER` pour valider.

Avant de redémarrer le serveur nous allons vérifier notre configuration en tapant la commande `sudo nginx -t`, un code similaire doit alors s'afficher :

```
1  nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
2  nginx: configuration file /etc/nginx/nginx.conf test is successful
```

Vous pouvez à présent redémarrer le serveur NGINX pour que vos modifications soient prises en compte :

```
1  sudo systemctl start nginx
```

Charger l'application sur le VPN

A ce stade, tous nos éléments sont prêts à être mit en ligne sur votre VPN. Nous devons répéter certaines des opération que nous avons fait en local sur le VPN et mettre en place in système que exécute notre application en permanence.

Cloner le repo sur le VPN

Nous allons t cloner la branche `master` de notre repo GitHub :

```
sudo git clone https://github.com/userName/repoName.git --branch master
    ↪  MEANquickstart

cd MEANquickstart && sudo npm install
```

> Ces commandes clonent votre application dans un dossier nommé app puis installent les dépendances.

Installer MongoBD sur le VPN

Nous allons à présent configurer MongoDB sur le VPN, il faut premièrement récupérer le package de MongoDB de la façon suivante :

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
    ↪ EA312927

echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2
    ↪ multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.
    ↪ list

sudo apt-get update
```

> Cette première étape permet de configurer l'installation de MongoDB.

Uns fois le package récupéré, nous pouvons installer MongoDB :

```
sudo apt-get install -y mongodb-org
```

Nous allons configurer MongoDb en tant que service en commençant par créer un fichier qui nous permettra de le configurer :

```
sudo nano /etc/systemd/system/mongodb.service
```

Une fois nano ouvert, copiez-collez le code suivant :

```
[Unit]
Description=High-performance, schema-free document-oriented database
After=network.target
```



```
4
5  [Service]
6  User=mongodb
7  ExecStart=/usr/bin/mongod --quiet --config /etc/mongod.conf
8
9  [Install]
10 WantedBy=multi-user.target
```

> Pour sauvegarder le fichier taper les touche `CTRL + X`, `Y` pour enregistrer et `ENTER` pour valider.

Il ne reste plus qu'à lancer le service pour activer MongoDb :

```
1 sudo systemctl start mongodb
```

> Vous pouvez vérifier le statut de votre service avec la commande suivant : `sudo systemctl status mongodb`.

Installer et configurer PM2

Lorsque nous travaillons en local, nous lançons la commande `npm start` pour lancer notre application. Dans le cadre de la mise en production de notre application sur le VPN, nous devons installer un système qui permet de gérer la mise en route de notre application. Nous allons installer **PM2** sur notre server, **PM2** est un sytème de gestion qui rendra notre application active en permanence :

```
1 sudo npm install -g pm2
2
3 pm2 start server.js
```

> A la fin de ces commande vous devez voir s'afficher un tableau indiquant que l'instance `server` est `online`.

Nous allons à présent configurer PM2 pour qu'il puisse redémarrer si le serveur redémarre en tapant la commande suivante :

```
1 pm2 startup systemd
```

> Cette commande permet permet d'afficher une commande à exécuter pour valider la configuration.

Copiez la commande qui s'affiche et exécuté la :

```
1 sudo env PATH=$PATH :/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup
    ↪ systemd -u server-admin --hp /home/server-admin
```


> A la fin de cette commande la configuration de PM2 est terminée.

Références

- **MEAN stack** From Wikipedia, the free encyclopedia
- **Javascript** From Wikipedia, the free encyclopedia
- **jQuery** Write less, do more
- **Brendan Eich** From Wikipedia, the free encyclopedia
- **Netscape** From Wikipedia, the free
- **iTerm** Terminal emulator for macOS
- **PostMan** Postman Makes API Development Simple
- **Cmder** Portable consoleemulator for Windows
- **NodeJS** JavaScript runtime built on Chrome's V8 JavaScript engine
- **OpenClassroom** Installer Node.js
- **MongoDB** For giant Idea
- **Homebrew** The missing package manager for macOS
- **CLI** Commande Line Interface
- **Angular** One framework. Mobile & desktop.
- **Angular CLI** A command line interface for Angular
- **AngularJS** Angular first version
- **Visual Studio Code** Free. Open source. Runs everywhere.
- **ExpressJs** Infrastructure Web minimaliste, souple et rapide pour Node.js
- **NodeMon** Nodemon reload, automatically.
- **Body-Parser** Node.js body parsing middleware.
- **Path** Provides utilities for working with file and directory paths
- **Ejs** Embedded JavaScript templating
- **package.json** Specifics of npm's package.json handling
- **JSON** From Wikipedia, the free encyclopedia
- **Mongoose** Elegant mongodb object modeling for node.js
- **RGPD : par où commencer, CNIL** Les 4 actions principales à mener pour entamer votre mise en conformité
- **CNIL** Commission nationale de l'informatique et des libertés
- **Bcrypt** Lib to help you hash passwords
- **mysql** NPM module to deal with MySQL and NodeJS
- **SQL** From Wikipedia, the free encyclopedia
- **Damien Truffaut, LinkedIn** Sr Consultant - Internet Technologies
- **Giuseppe Militello** Intervenant en développement Web

- **CouchDB**Seamless multi-master sync, that scales from Big Data to Mobile
- **PouchDB** The Database that Syncs
- **NoSQL** Being non-relational, distributed, open-source and horizontally scalable.
- **JSON : JavaScript Object Notation** From Wikipedia, the free encyclopedia
- **Digital Ocean** Simplicity at scale
- **Jason Lengstorf** Awesome MongoDB tutorial
- **PM2** Advanced, production process manager for Node.js
- **TypeScrypt** Javascript that scale

