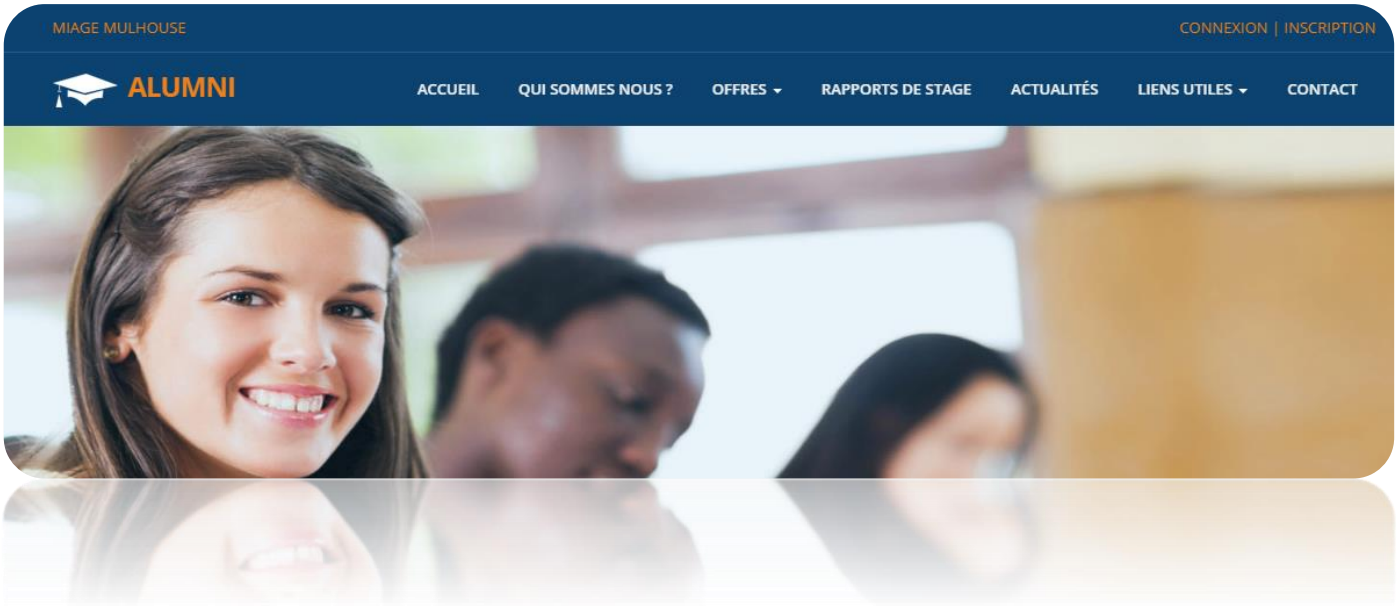
11/6/2018

MANUEL DEVELOPPEUR DU SITE WEB DES ANCIENS MIAGISTES



Réalisé par :

- M. Nabil ELASLAOUI
- M. Akram ELGARDID
- M. Boubacar SOW
- M. Wendpouiré YEREMGANGA

Encadré par :

- M. Yvan MAILLOT

Sommaire

Introduction

Le projet consiste à réaliser un site web avec la technologie JavaScript (Initialement NodeJS, ReactJS et Meteor pour une maintenance évolutive) qui permettra aux anciens miagistes de garder le lien avec la miage (dépose d'offres d'emploi, de stage, d'alternance et diffusion des informations sur les événements comme le rallye, le forum, la remise des diplômes).

Le développement a commencé cette année durant le mois de mai 2018 par A, B, C, D, dans le cadre de notre formation en M1 MIAGE.
Les apports effectués pendant cette période ont été la mise en place d'un noyau de départ qui fonctionne complètement. En effet, maintenant il est possible de renouer le lien avec la miage à travers le site tout en consultant le manuel utilisateur afin de comprendre plus profondément comment il fonctionne.

Il est nécessaire d'avoir acquis les compétences minimales et nécessaires pour développer un site web avec la technologie JavaScript (NodeJS). Ces notions dont nous avons appris avant de commencer à développer le site, plus spécifiquement NodeJS.

Enfin, ce manuel a pour but de faire comprendre comment fonctionne le site, son architecture, sa structure de données afin de corriger/ajouter certaines fonctionnalités.

1. Environnement de développement

1.1. NodeJS

Node.js est une plateforme logicielle libre et événementielle en JavaScript qui utilise la machine virtuelle V8 (Un moteur JavaScript libre et Open Source).
Concrètement, Node.js est un environnement bas niveau permettant l'exécution de JavaScript côté serveur.
L'installation se réalise tout en suivant les instructions sur le site dédié https://nodejs.org/fr/ ou bien en suivant le manuel d'installation.

1.2. Visual Studio Code

Il s'agit d'un éditeur de code multiplateforme, open source et gratuit supportant une dizaine de langages et supporte le débogage, le contrôle du git, ainsi que plusieurs fonctionnalités à découvrir.

1.3. IntelliJ IDEA

L'outil de développement IntelliJ IDEA est un outil spécialement dédié
Java. Il s'avère donc être l'outil central utilisé tout au long du développement de l'application puisqu'il supporte JavaScript ainsi que plusieurs langages.
L'édition du code source, la compilation et toutes autres modifications peuvent être aisément réalisées grâce à cet outil. Ce dernier permet de créer tous les fichiers nécessaires, de modifier les sources et de tester le code.

Une fois l'outil est correctement installé (édition communautaire libre, voir le manuel d'installation); le développement est plus flexible et pratique.

1.4. Win'Design

WinDesign est un outil qui permet de modéliser des systèmes d'information organisationnel et informatique. Afin de couvrir l'ensemble des modélisations, WinDesign se divise en 3 modules:

- Module Business Process: Modélisation des cartographies Métier et SI.
- Module Object: Modélisation UML.
- Module Database: Modélisation des données.

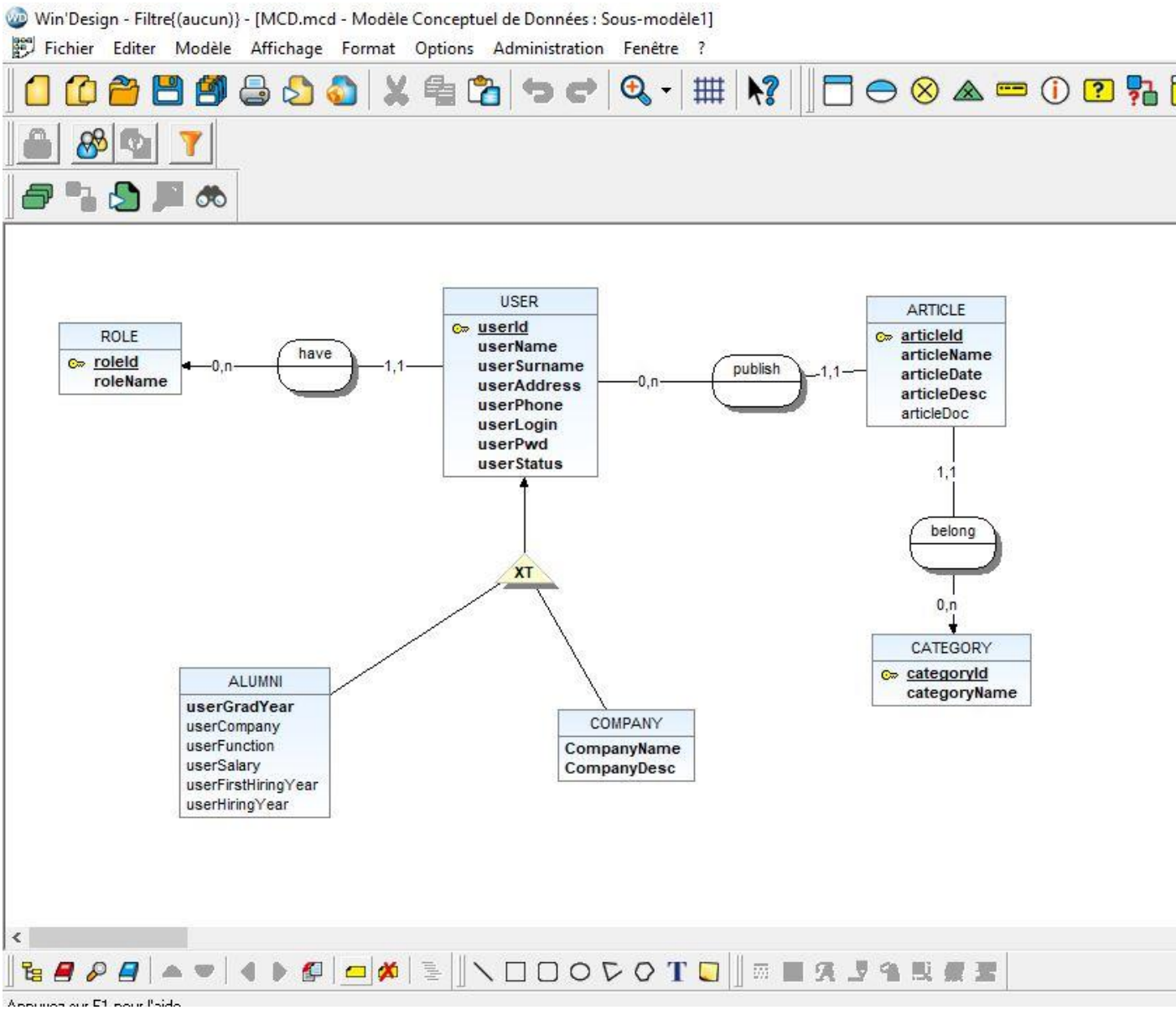
Lors de ce projet, on se limitera au module Database.

2. Base de données

2.1. MCD

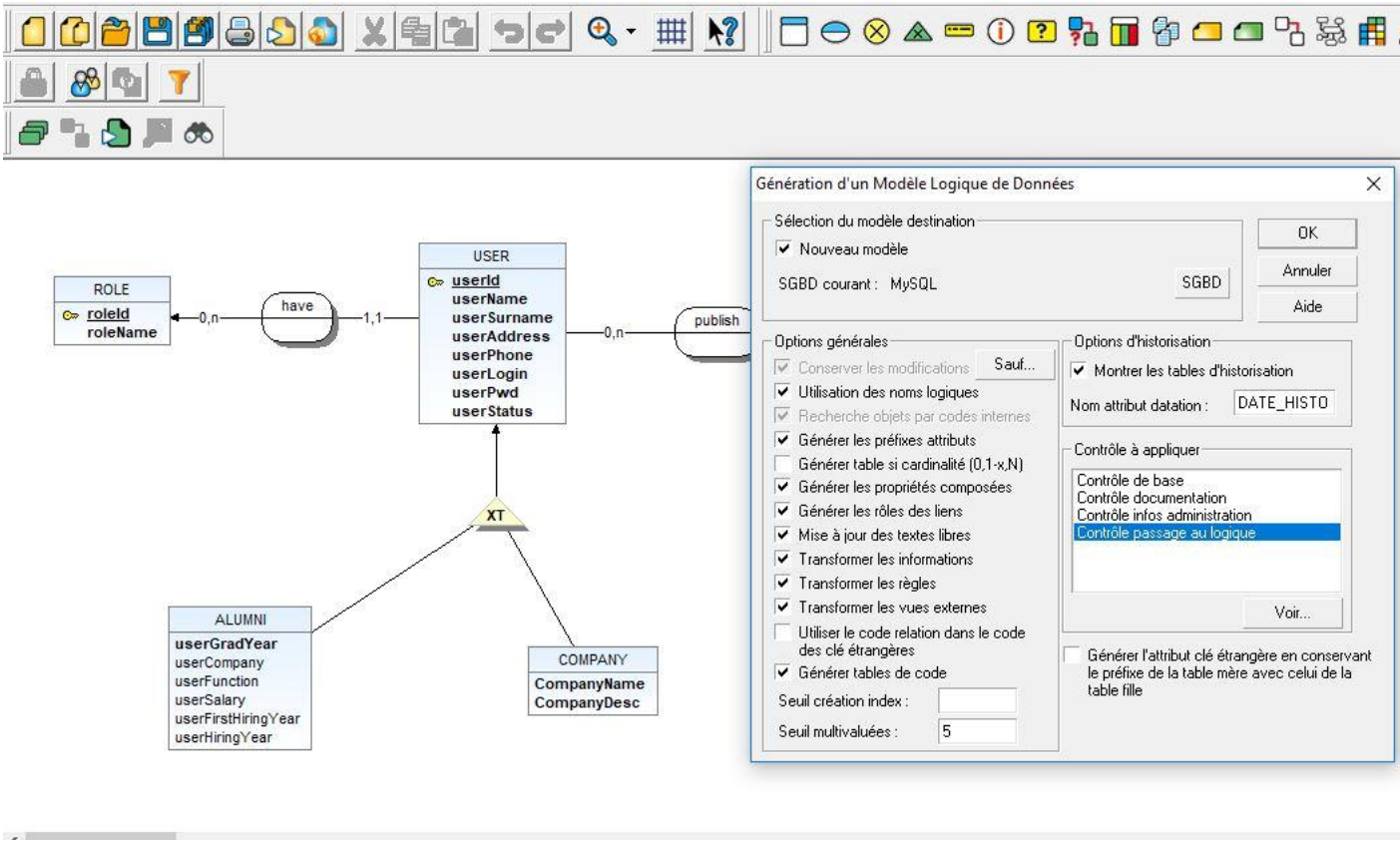
Le modèle conceptuel des données (MCD) a pour but d'écrire de façon formelle les données qui seront utilisées par le système d'information.

Il s'agit donc d'une représentation facilement compréhensible, permettant de décrire le système d'information à l'aide d'entités.

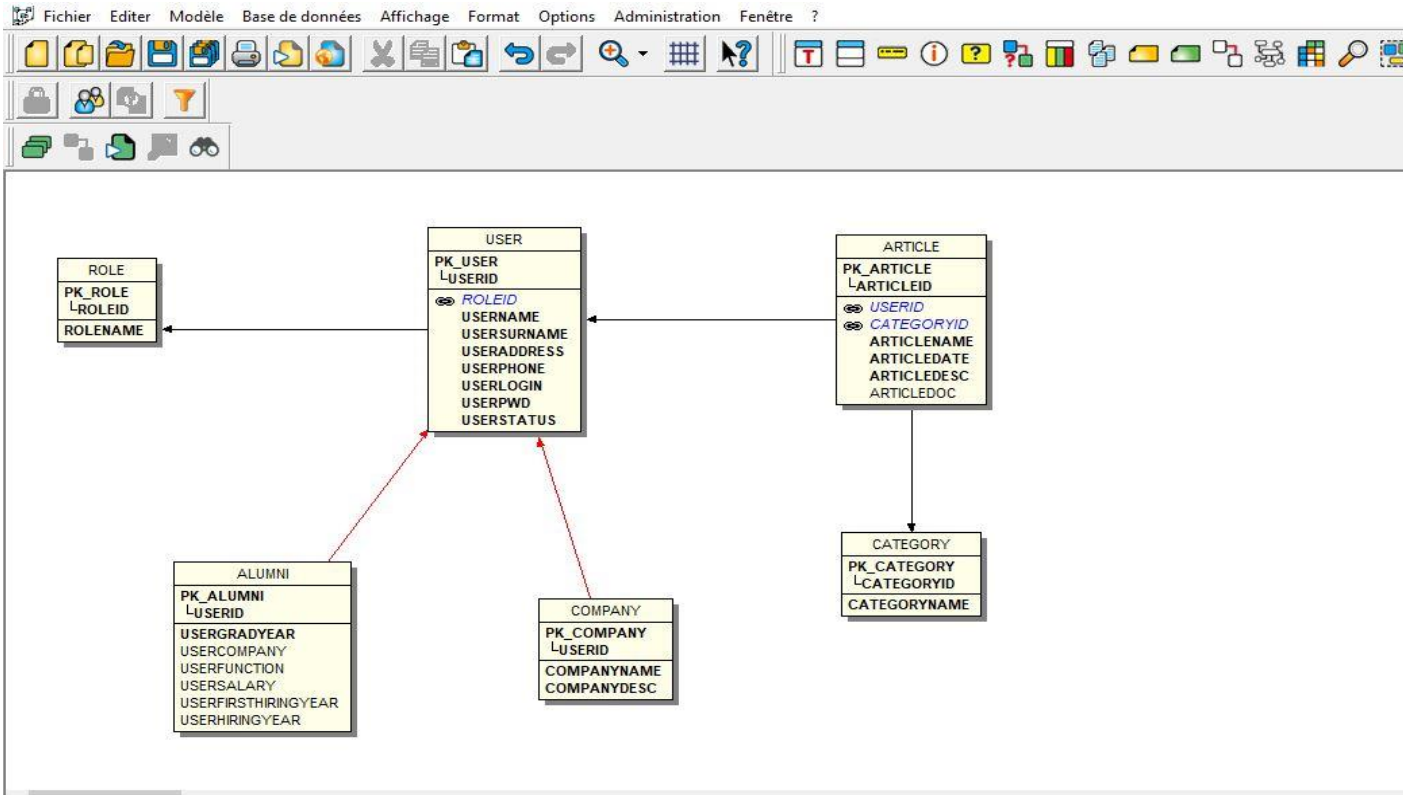


2.2. MLD

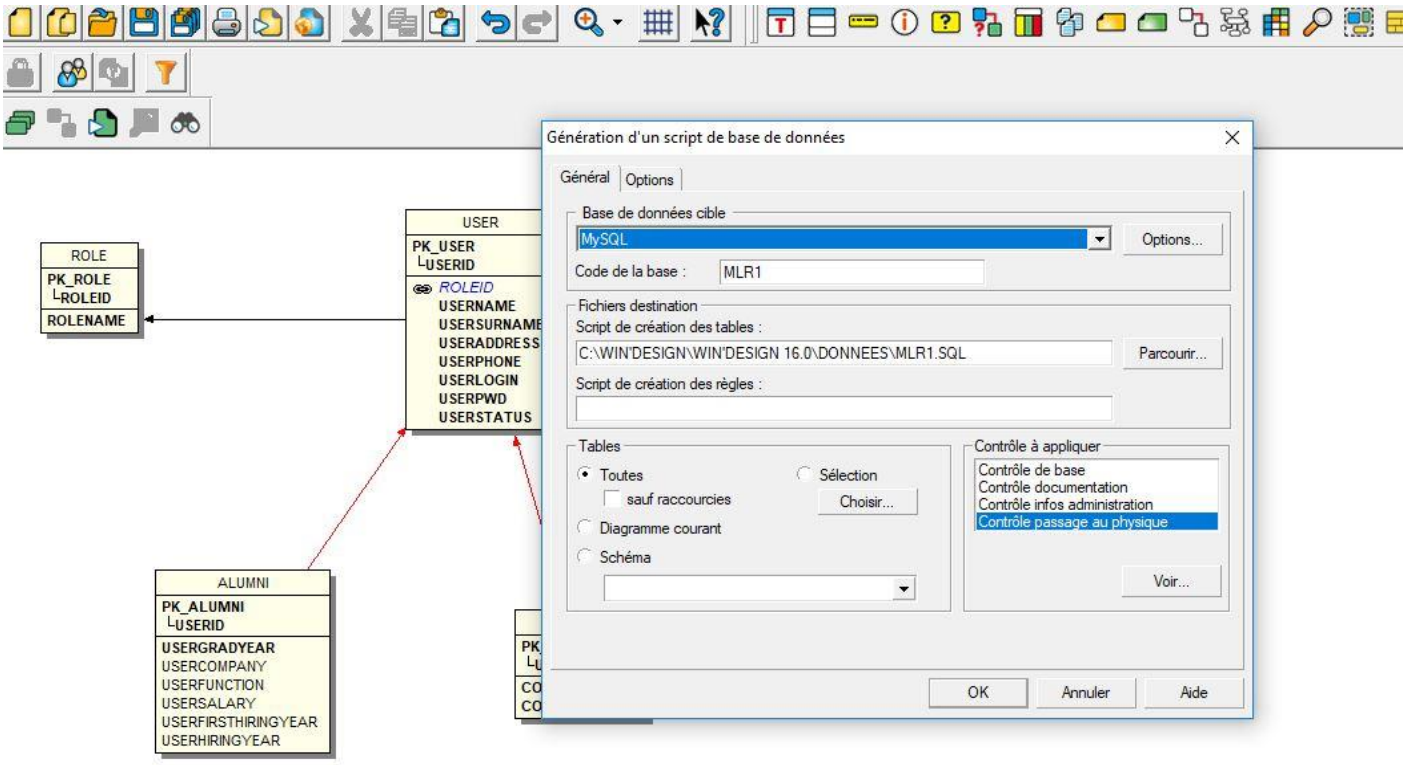
Le modèle logique des données (MLD) consiste à décrire la structure de données utilisée sans faire référence à un langage de programmation. Il s'agit donc de préciser le type de données utilisées lors des traitements. Ainsi, le modèle logique est dépendant du type de base de données utilisé.
Comment générer le modèle logique :


ROLE
roleId
roleName
0,n
have
1,1
USER
userId
userName
userSurname
userAddress
userPhone
userLogin
userPwd
userStatus
0,n
publish
XT
ALUMNI
userGradYear
userCompany
userFunction
userSalary
userFirstHiringYear
userHiringYear
COMPANY
CompanyName
CompanyDesc
Génération d'un Modèle Logique de Données
Sélection du modèle destination
Nouveau modèle
SGBD courant : MySQL
SGBD
OK
Annuler
Aide
Options générales
Conserver les modifications
Sauf...
Utilisation des noms logiques
Recherche objets par codes internes
Générer les préfixes attributs
Générer table si cardinalité (0,1-x,N)
Générer les propriétés composées
Générer les rôles des liens
Mise à jour des textes libres
Transformer les informations
Transformer les règles
Transformer les vues externes
Utiliser le code relation dans le code des clé étrangères
Générer tables de code
Seuil création index :
Seuil multivaluées : 5
Options d'historisation
Montrer les tables d'historisation
Nom attribut datation : DATE_HISTO
Contrôle à appliquer
Contrôle de base
Contrôle documentation
Contrôle infos administration
Contrôle passage au logique
Voir...
Générer l'attribut clé étrangère en conservant le préfixe de la table mère avec celui de la table fille

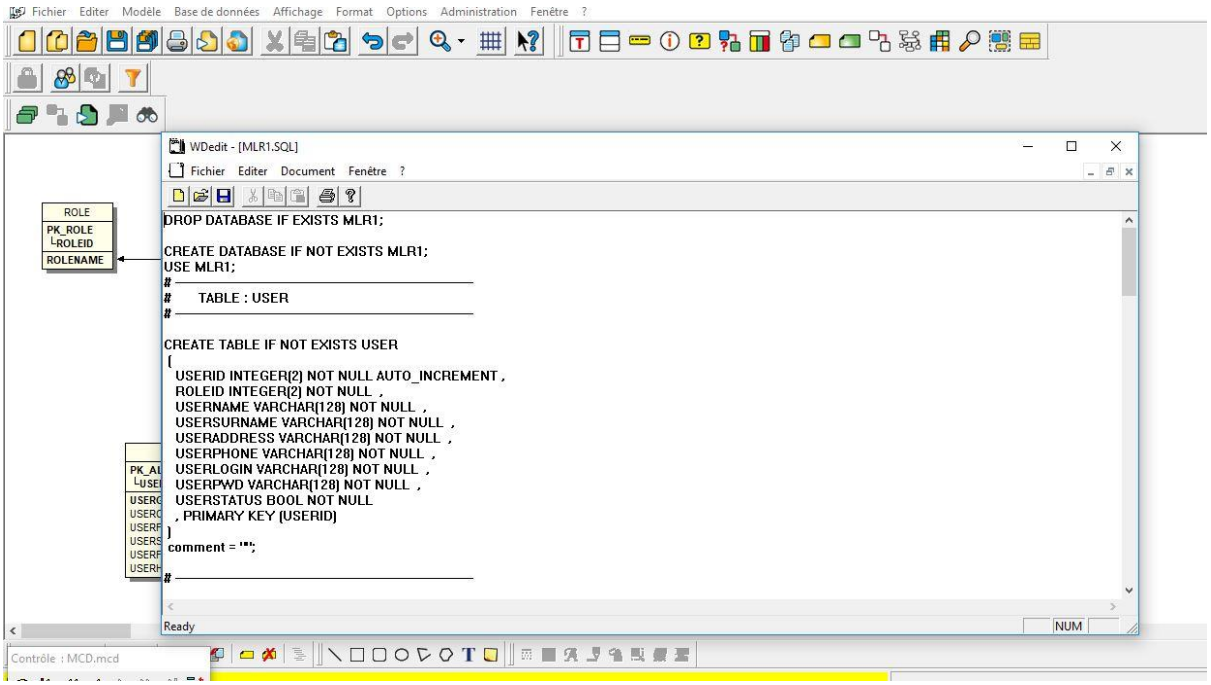
Résultat :


Fichier Editer Modèle Base de données Affichage Format Options Administration Fenêtre ?
ROLE
PK_ROLE
ROLEID
ROLENAME
USER
PK_USER
USERID
ROLEID
USERNAME
USERSURNAME
USERADDRESS
USERPHONE
USERLOGIN
USERPWD
USERSTATUS
ARTICLE
PK_ARTICLE
ARTICLEID
USERID
CATEGORYID
ARTICLENAME
ARTICLEDATE
ARTICLEDESC
ARTICLEDOC
CATEGORY
PK_CATEGORY
CATEGORYID
CATEGORYNAME
ALUMNI
PK_ALUMNI
USERID
USERGRADYEAR
USERCOMPANY
USERFUNCTION
USERSALARY
USERFIRSTHIRINGYEAR
USERHIRINGYEAR
COMPANY
PK_COMPANY
USERID
COMPANYNAME
COMPANYDESC

Passage au modèle physique :



Le script sql généré :



2.3. Structure de données

Cette partie consiste à définir les différents éléments de la structure de données lors de ce projet:

Rôles: **ADMIN**, **MODERATEUR**, **ETUDIANT**, **PROFESSEUR**, **ALUMNI**, **ENTREPRISE**

A chaque rôle, différents droits sont affectés.

Catégories: **Emplois**, **Stages**, **Alternances**, **Actualités**

Chaque catégorie définit un type d'article.

Utilisateurs: **ETUDIANT**, **ALUMNI**, **ENTREPRISE**, **PROFESSEUR**

L'alumni (l'ancien miagiste) peut ajouter ses propres informations (date d'embauche, salaire, entreprise, etc…), le type 'entreprise' est un utilisateur qui représente une entreprise, il peut aussi ajouter une description et un nom de l'entreprise, le professeur peut consulter les rapports de stage de fin d'études et les télécharger ainsi qu'il peut consulter les offres publiées et en ajouter.

2.4. Ajout d'une table à la base de données

Pour ajouter une table à la base de données, on peut passer par la modification du MCD afin de générer le script de la base de données, sauf que cette méthode n'est pas très pratique puisqu'elle va supprimer toutes les données existantes.
Donc pour résoudre ce problème, on peut créer la table en passant directement par le script SQL.

Par exemple: Lors de notre projet, on a ajouté une table **RAPPORT** concernant les rapports de fin d'études tout en le forçant à changer son adresse mail (saisir une adresse personnelle puisque son adresse UHA n'existera plus après sa fin d'études)

```sql
CREATE TABLE `rapport` (
  `RAPPORTID` int(11) NOT NULL,
  `USERID` int(11) NOT NULL,
  `USEREMAIL` varchar(50) NOT NULL,
  `RAPPORTNAME` varchar(100) NOT NULL,
  `RAPPORTDATE` date NOT NULL,
  `RAPPORTDESC` text NOT NULL,
  `RAPPORTDOC` varchar(50) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```sql
208  --
209  -- Index pour la table `rapport`
210  --
211  ALTER TABLE `rapport`
212    ADD PRIMARY KEY (`RAPPORTID`),
213    ADD KEY `FK_RAPPORT_USER` (`USERID`);
214
```

2.5. Ajout d'un champ à une table

Pour ajouter un champ à une table, on peut passer par la modification du MCD afin de générer le script de la base de données, sauf que cette méthode n'est pas très pratique puisqu'elle va supprimer toutes les données existantes.

Donc pour résoudre ce problème, on peut créer la table en passant directement par le script SQL.

```sql
297
298  ALTER TABLE `rapport`
299  ADD sujetRapport varchar(50) NOT NULL;
300
```

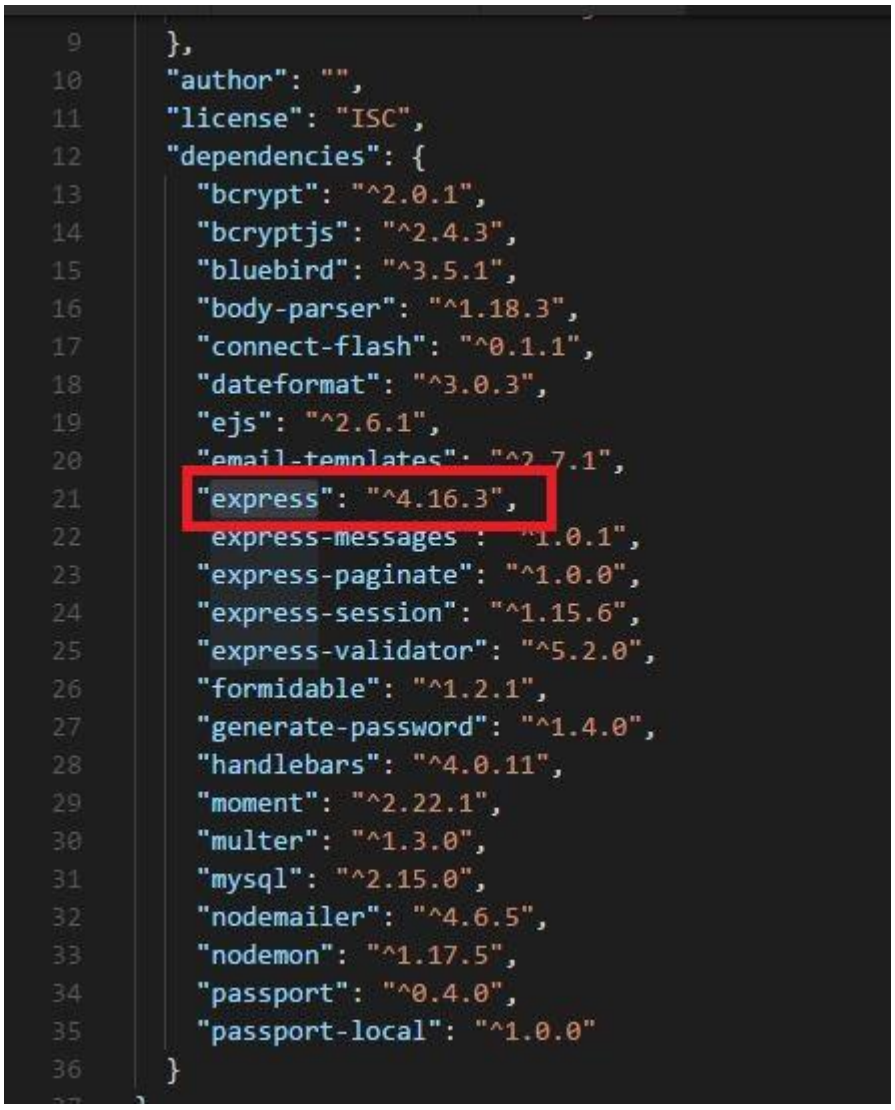
3. Structure du projet

3.1. Express

Express.js est un framework pour construire des applications web basées sur Node.js. C'est de fait le framework standard pour le développement de serveur en Node.js.

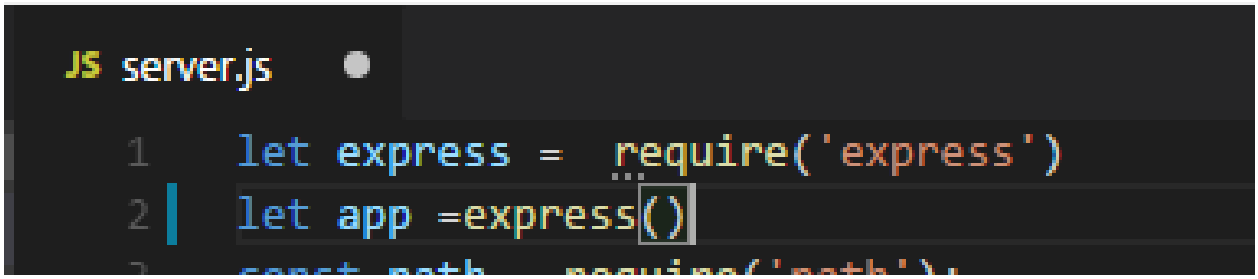
Pour installer Express JS il suffit d'exécuter la commande npm install express.

Après installation il va apparaître dans le fichier package.json



```
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^2.0.1",
    "bcryptjs": "^2.4.3",
    "bluebird": "^3.5.1",
    "body-parser": "^1.18.3",
    "connect-flash": "^0.1.1",
    "dateformat": "^3.0.3",
    "ejs": "^2.6.1",
    "email-templates": "^2.7.1",
    "express": "^4.16.3",
    "express-messages": "^1.0.1",
    "express-paginate": "^1.0.0",
    "express-session": "^1.15.6",
    "express-validator": "^5.2.0",
    "formidable": "^1.2.1",
    "generate-password": "^1.4.0",
    "handlebars": "^4.0.11",
    "moment": "^2.22.1",
    "multer": "^1.3.0",
    "mysql": "^2.15.0",
    "nodemailer": "^4.6.5",
    "nodemon": "^1.17.5",
    "passport": "^0.4.0",
    "passport-local": "^1.0.0"
  }
```

Inclusion d'express en créant un objet ***app*** tout en appelant la fonction **express()** :



```
JS server.js
let express = require('express')
let app =express()
```

Les Routes:

```
// route vers page d'authentification
app.get('/login',(request,response)=>{
    response.render('login')
})

// route vers page à propos
app.get('/apropos',(request,response)=>{
    response.render('front/apropos')
})

// route vers page contact
app.get('/contact',(request,response)=>{
    response.render('front/contact')
})
```

Il suffit d'indiquer les différentes routes (les différentes URL) à laquelle l'application doit répondre. Ici par exemple j'ai créé une route "/login". Une fonction de callback est appelée quand quelqu'un demande cette route.

Ce système est beaucoup mieux fait que nos "if" imbriqués. On peut écrire autant de routes de cette façon qu'on le souhaite.

```
let users = require('./controllers/users');
app.use('/users', users);

let roles = require('./controllers/roles');
app.use('/roles', roles);

let articles = require('./controllers/articles');
app.use('/articles', articles);
let category = require('./controllers/category');
app.use('/category', category);

let company = require('./controllers/company');
app.use('/company', company);

let rapport = require('./controllers/rapport');
app.use('/rapports', rapport);

let rapportStage = require('./controllers/rapport_stage');
app.use('/rapportStages', rapportStage);

let offres = require('./controllers/offres');
app.use('/offres', offres);
```

Si un url commence par **/users** il va le rediriger vers le dossier **controllers** et il va exécuter l'url qui existe dans les fichiers **users.js**. Même chose pour rôles, articles, company, … etc.

Routes dynamiques

Express vous permet de gérer des routes dynamiques, c'est-à-dire des routes dont certaines portions peuvent varier. Vous devez écrire ":nomvariable" dans l'URL de la route, ce qui aura pour effet de créer un paramètre accessible depuis **req.params.nomvariable**

```
107
108  router.get('/delete/:id', ensureAuthenticated, (request, response) => {
109      Role.getOne(request.user.user.ROLEID, function (role) {
110      if (role[0].ROLENAME === 'ADMIN' || role[0].ROLENAME === 'MODERATEUR') {
111          if (request.params.id) {
112              User.delete(request.params.id, function () {
113                  request.flash('success', "User supprimé")
114              })
115          }
116      } else {
117          response.redirect('/')
118      }
119  })
120  response.redirect('/users')
121  })
122  ;
123
```

Dans cet exemple pour supprimer un utilisateur on doit envoyer son id en paramètre (**/delete/:id**).

Remarque : on verra les détails de la fonction après.

Express nous permet d'utiliser des templates pour sortir de cet enfer. Les templates sont en quelque sorte des langages faciles à écrire qui nous permettant de produire du HTML et d'insérer au milieu du contenu variable.

Il existe de nombreux systèmes de templates, on a choisi dans notre projet d'utiliser EJS (Embedded JavaScript). Pour l'installer il suffit d'exécuter la commande **npm install ejs**.

Une fois installé il doit apparaître dans le fichier package.json qu'on a vu toute à l'heure.

```
12
13  /**
14   * Liste les différents utilisateurs avec leurs roles
15   **/
16  router.get('/', ensureAuthenticated, function (req, res) {
17      Role.getOne(req.user.user.ROLEID, function (role) {
18          if (role[0].ROLENAME === 'ADMIN' || role[0].ROLENAME === 'MODERATEUR') {
19              User.Allu(function (users) {
20                  res.render('users/users', {users: users})
21              })
22          } else {
23              res.redirect('/')
24          }
25      })
26  });
27
```

Ce code fait appel à un fichier users.ejs **qui doit se trouver dans un sous-dossier appelé "views/users"**. On transmet à la vue la liste des utilisateurs récupérée de la base de données.

Pour afficher les informations dans la vue on utilise **<%= %>**

```
<% for(user of users) { %>
<tr>
    <td></td>
    <td><%= user.USERNAME %></td>
    <td><%= user.USERSURNAME %></td>
    <td><%= user.USERPHONE %></td>
    <td><%= user.USERLOGIN %></td>
    <td><%= user.ROLENAME %></td>
    <td>
```

Express et les middlewares :

Express est fourni avec une quinzaine de middlewares de base, et d'autres développeurs peuvent bien entendu en proposer d'autres via NPM. Les middlewares livrés avec Express fournissent chacun des micro-fonctionnalités. Il y a par exemple :

3.1.1. Express-session

Permet de gérer des informations de session (durant la visite d'un visiteur).

3.1.2. Express-validator

Permet de gérer la validation des champs.

Tous ces middlewares communiquent entre eux en se renvoyant jusqu'à 4 paramètres :

- **err**: les erreurs
- **req**: la requête du visiteur
- **res**: la réponse à renvoyer (la page HTML et les informations d'en-tête)
- **next**: un callback vers la prochaine fonction à appeler.

Utiliser les middlewares au sein d'Express

Concrètement, il suffit d'appeler la méthode **app.use()** pour utiliser un middleware. On peut les appeler les uns à la suite des autres.

```
// Middleware session
app.use(express.static('public')) // pour acceder aux dossiers dans public prefix assets
app.use(bodyParse.urlencoded({extended : false}))
app.use(bodyParse.json())

app.use(session({
    secret: 'azeazazas',
    resave: false,
    saveUninitialized: true,
    cookie: { secure: false }
}));

// Express Messages Middleware
app.use(require('connect-flash')());
app.use(function (req, res, next) {
    res.locals.messages = require('express-messages')(req, res);
    next();
});

// Express Validator Middleware
app.use(expressValidator({
    errorFormatter: function(param, msg, value) {
        var namespace = param.split('.')
            , root    = namespace.shift()
            , formParam = root;

        while(namespace.length) {
```

3.1.3. Passport

Passport est un middleware d'authentification pour Node.js. Extrêmement flexible et modulaire, Passport peut être intégré discrètement à n'importe quelle application Web basée sur Express. Un ensemble complet de stratégies prend en charge l'authentification en utilisant un nom d'utilisateur et un mot de passe, Facebook, Twitter … etc.

Configuration passport

Pour l'installer passport il faut exécuter la commande suivante **npm install passport**

```
56  // Passport Config
57  require('./config/passport')(passport);;
58  // Passport Middleware
59  app.use(passport.initialize());
60  app.use(passport.session());
61
```

Ici, on requiert passport et l'initialisation avec son middleware d'authentification de session, directement dans notre application Express.

Implémentation de l'authentification locale

Il est temps de configurer notre stratégie d'authentification en utilisant **Passport-local**. Allons de l'avant et installons-le. Exécutez la commande suivante:

```
npm install passport-local --save
```

passport.js

```
const LocalStrategy = require('passport-local').Strategy;
const User = require('../models/User');
const db = require('../config/db');
const bcrypt = require('bcrypt');

module.exports = function(passport){
    // Local Strategy
    passport.use(new LocalStrategy({
            usernameField: 'USERLOGIN',
            passwordField: 'USERPWD'
        },
        function(username, password, done) {
            db.query('SELECT * FROM user WHERE USERLOGIN=?',[username],(err,results,fields)=>{
                if (err){done(err);}
                if (results.length === 0){
                done(null,false);
            }else {
                const hash = results[0].USERPWD.toString();
                bcrypt.compare(password, hash, function(err, response){
                    if (response === true){
                        if(results[0].USERSTATUS === 1){
                            return done(null,{user:results[0]});
                        }else {
                            return done(null,false);
                        }
                    }else {
                        return done(null,false);
                    }
                        }
                    });
                }
            })
            }
        ));

        passport.serializeUser(function(user, done) {
            done(null, user);
        });

        passport.deserializeUser(function(user, done) {
            done(null, user);
        });
    }
```

Examinons ce code. D'abord, nous exigeons le passport-local Strategy. Ensuite, nous indiquons à Passport d'utiliser une instance de LocalStrategy ce dont nous avons besoin. Là, nous utilisons simplement une requête sql pour trouver un enregistrement basé sur le nom d'utilisateur.

Si un enregistrement est trouvé et que le mot de passe correspond, le code ci-dessus renvoie l'objet **user**. Sinon, il revient false.

Remarque : Passport utilise toujours user et password alors que nous dans notre base de données en utlise **USERLOGIN** et **USERPWD**, donc pour que passport comprend notre login et password on a ajouté ces deux lignes:

```
passport.use(new LocalStrategy({
        usernameField: 'USERLOGIN',
        passwordField: 'USERPWD'
    },
```

```
34
35  // Login Process
36  router.post('/login', function (req, res, next) {
37      passport.authenticate('local', {
38          successRedirect: '/',
39          failureRedirect: '/login',
40          failureFlash: 'adresse email ou mot de passe incorrect'
41      })(req, res, next);
42  });
43
44  // logout
```

Dans le contrôleur users, il y a notre post, avec la méthode **[passport.authenticate]** (http://www.passportjs.org/docs/authenticate/) qui tente de s'authentifier avec le username et password, dans ce cas 'local' nous redirigera vers '/login' (page d'authentification) si cela échoue. Sinon, il nous redirigera vers la route '/' (page d'accueil).

```
380  // Access Control
381  function ensureAuthenticated(req, res, next) {
382      if (req.isAuthenticated()) {
383          return next();
384      } else {
385          req.flash('danger', "vous n'etes pas connecter");
386          res.redirect('/login');
387      }
388  }
```

Dans le contrôleur users il y a la méthode **ensureAuthenticated**, cette fonction permettant de vérifier si l'utilisateur est authentifié dans ce cas on va le rediriger vers la page souhaité (return next()), sinon on va le rediriger vers la page d'authentification.

3.1.4. Bcrypt

Pour hacher le mot de passe on va utiliser la **bcrypt** pour l'installer il suffit d'exécuter la commande suivante :

npm install bcrypt

```
bcrypt.genSalt(10, function (err, salt) {
    bcrypt.hash(USERPWD, salt, function (err, hash) {
```

Cette fonction permet aussi de comparer deux mots de passe.

```
}else {
    const hash = results[0].USERPWD.toString();
    bcrypt.compare(password, hash, function(err, response){
        if (response === true){
            if(results[0].USERSTATUS === 1){
                return done(null,{user:results[0]});
            }else {
                return done(null,false);
            }
        }else {
```

3.1.5. Nodemailer

Nodemailer est le module qui permet d'envoyer des emails. Il s'agit du fichier **/config/emailing.js**

```
const nodemailer = require('nodemailer'),
    //creds = require('./creds'),
    //creation de l'objet transporter
    transporter = nodemailer.createTransport({
        host: 'smtp.gmail.com',
        port: 587,
        secure: false,
        auth: {
            user: 'miagemulhousetest@gmail.com',
            pass: 'miagemulhousetest123',
        },
        tls:{
            rejectUnauthorized: false // false dans un environement de developpement
        },
    }),
    EmailTemplate = require('email-templates').EmailTemplate,
    path = require('path'),
    Promise = require('bluebird');
```

Pour le développement et les tests, un compte gmail de test a été créé. Les identifiants du compte:

- **Login** : miagemulhousetest@gmail.com
- **Mot de passe** : miagemulhousetest123

L'envoie de mail se fait grâce au code suivant :

```
    let MonUser = [
        {
            username: USERNAME,
            usersurname: USERSURNAME,
            email: USERLOGIN,
        }
    ]
    require('../config/emailing')('welcome', MonUser);
```

La variable MonUser doit contenir la propriété email qui contient l'adresse email du récepteur.

'**welcome**' est le nom du template du mail de bienvenue.

3.1.6. Formidable

Formidable est un module Node.js pour l'analyse des données issues des formulaires, y compris le téléchargement de fichiers (multipart / form-data.).

```
const express = require('express');
const router = express.Router();
const path = require('path');
const url = require('url');

const uploadDir = path.join(__dirname, '/..', 'public/uploads/');
var formidable = require('formidable');
var fs = require('fs');
```

```
router.post('/doAdd', ensureAuthenticated, function (req, res) {

    var form = new formidable.IncomingForm()
    form.multiples = true
    form.keepExtensions = true
    form.uploadDir = uploadDir
    form.parse(req, function (err, fields, files) {
        var nomFichier = '';
        if (files.ARTICLEDOC.size !== 0) {
            var cheminFichier = files.ARTICLEDOC.path.split('\\');
            nomFichier = cheminFichier[cheminFichier.length - 1];
        } else {
            var cheminFichier = files.ARTICLEDOC.path.split('\\');
            if (cheminFichier[cheminFichier.length - 1] === 'fileTodelete') {
                fs.unlink('public/uploads/' + cheminFichier[cheminFichier.length - 1], function (err) {
                    if (err) {
                        console.error(err.toString());
                    } else {
                        console.warn('le fichier ' + cheminFichier[cheminFichier.length - 1] + ' a été supprimé');
                    }
                });
            }
        }
        if (err) res.render('articles/add/'+fields.CATEGORYID, {errors: err})
        Article.create(fields.USERID, fields.CATEGORYID, fields.ARTICLENAME, new Date(), fields.ARTICLEDESC, nomFichier, function () {
            req.flash('success', "Article ajouté avec succès !")
            res.redirect('/articles/'+fields.CATEGORYID)
        })

    })
    form.on('fileBegin', function (name, file) {
        if (file.name !== '') {
            var name = 'fichier';
            const [fileName, fileExt] = file.name.split('.')
            file.path = path.join(uploadDir, `${name} ${new Date().getTime()}.${fileExt}`);

        } else {
            var name = 'fileTodelete';
            file.path = path.join(uploadDir, name);
        }
```

3.2. Le serveur

Avec Node.js, on n'utilise pas de serveur web HTTP comme Apache par exemple. En fait, c'est à nous de créer le serveur.

Node.js est **monothread**, contrairement à Apache. Cela veut dire qu'il n'y a qu'un seul processus, qu'une seule version du programme qui peut tourner à la fois en mémoire.

En effet, il ne peut faire qu'une chose à la fois et ne tourne donc que sur un noyau de processeur.

Mais il fait ça de façon ultra efficace, et malgré ça il est quand même beaucoup plus rapide !

Cela est dû à la nature "orientée évènements" de Node.js. Les applications utilisant Node ne restent jamais les bras croisés sans rien faire. Dès qu'il y a une action un peu longue, le programme redonne la main à Node.js qui va effectuer d'autres actions en attendant qu'un événement survienne pour dire que l'opération est terminée.

Exemple de script serveur :

Cette partie de code permet d'envoyer à la vue les informations de l'utilisateur authentifié ainsi la liste des catégories et des actualités.

server.js

```
app.use(function (req,res,next) {
    Category.catMenu(function (cats) {
        /**
         * renvoyer à la vue la liste des catégories à mettre dans le menu
         */
        global.catMenu= cats;
    })

    /**
     * renvoyer à la vue l'actualité
     */
    Category.catActu(function (actu) {
        global.actu= actu;

    })

    /**
     * envoyer un boolean (isAuthenticated renvoyé par la fonction isAuthenticated()) à la vue
     * pour tester si un utilisateur est authentifié ou pas
     */
    res.locals.isAuthenticated = req.isAuthenticated();
    /**
     * renvoyer pageCourante à la vue qui sert à activer la rubrique courante class="active"
     */
    app.locals.pageCourante = req.path
```

server.js

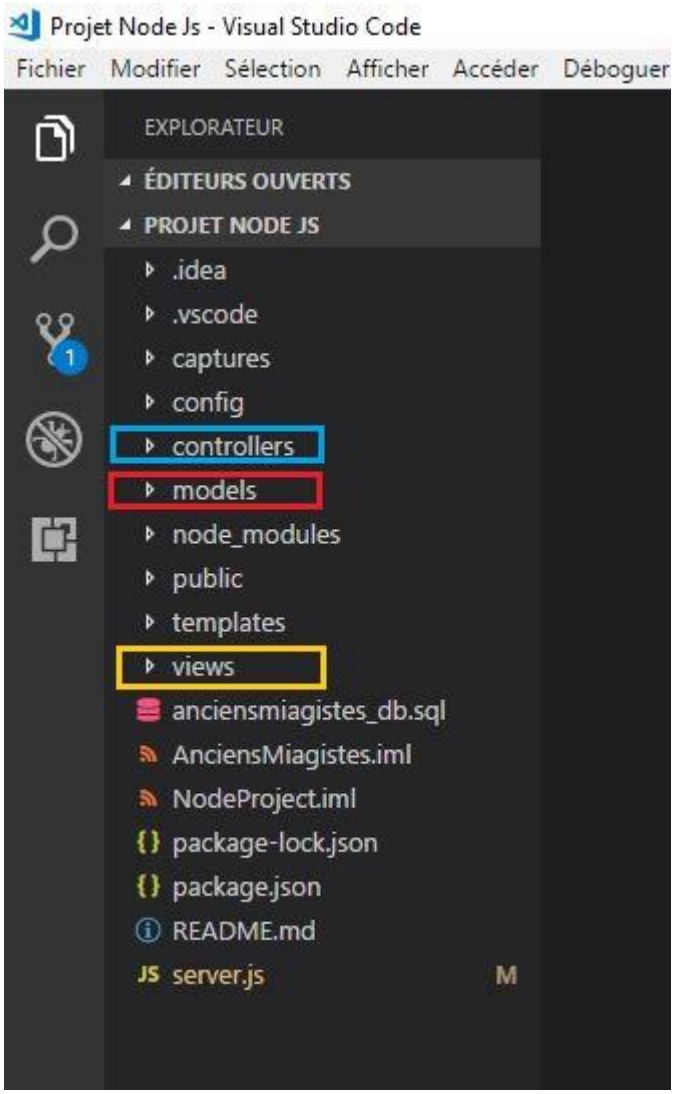
```
    res.locals.isAuthenticated = req.isAuthenticated();
    /**
     * renvoyer pageCourante à la vue qui sert à activer la rubrique courante class="active"
     */
    app.locals.pageCourante = req.path
    /**
     * renvoyer à la vue les informations de l'utilisateur authentifié
     */
    if (req.user != undefined){
        res.locals.nom = req.user.user.USERNAME;
        res.locals.prenom = req.user.user.USERSURNAME;
        res.locals.idUser = req.user.user.USERID;
        res.locals.adresse = req.user.user.USERADDRESS;
        res.locals.phone = req.user.user.USERPHONE;
        res.locals.email = req.user.user.USERLOGIN;
        /**
         * récupérer le role de l'utilisateur authentifié et le renvoyer à la vue
         */
        Role.getOne(req.user.user.ROLEID,function (role) {
            app.locals.role= role[0].ROLENAME
        })
        /**
         * renvoyer à la vue la liste des catégories
         */
        Category.all(function (cats) {
            app.locals.cats= cats;
        })

    }
    next();
})
```

3.3. Modèle MVC

Modèle-vue-contrôleur ou **MVC** est un motif d'architecture logicielle, composé de trois types de modules ayant trois responsabilités différentes : les modèles, les vues et les contrôleurs.

- Un modèle (Model) contient les données à afficher.
- Une vue (View) contient la présentation de l'interface graphique.
- Un contrôleur (Controller) contient la logique concernant les actions effectuées par l'utilisateur.



Exemple minimale complet "affichage des rôles" :

3.3.1. Le contrôleur

roles.js

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

/**
 * include du modele Role
 */
let Role = require('../models/Role')

    /**
    * Liste les différents roles
    **/
router.get('/',ensureAuthenticated,(request,response)=>{
    Role.getOne(request.user.user.ROLEID,function (role) {
        if(role[0].ROLENAME === 'ADMIN'){
            Role.all(function (roles) {
                response.render('roles/roles',{roles:roles})
            })
        }else{
            response.redirect('/')
        }
    })
})
```

Cette méthode permet de vérifier si l'utilisateur est authentifié avec la méthode **ensureAuthenticated** décrite au-dessus, après s'il est authentifié en vérifier son rôle s'il est Admin on va appeler la méthode all dans le modèle Rôle pour récupérer la liste des rôles ensuite on va le rediriger vers la vue **roles** en renvoyant la liste des rôles, sinon on va le rediriger vers la page d'accueil.

3.3.2. Le modèle

roles.js | Role.js

```
let connection = require('../config/db')
let moment = require('../config/moment')

class  Role {

    constructor(row){
        this.row = row
    }

    // getters
    get ROLENAME(){
        return this.row.ROLENAME
    }
    get ROLEID(){
        return this.row.ROLEID
    }
    // permet de récupérer la liste des roles en les stockant dans le callback cb
    static all(cb){
        connection.query('SELECT * FROM role',(err,rows)=>{
            if (err) throw err
            cb(rows.map((row) => new Role(row)))
        })
    }
```

3.3.3. La vue

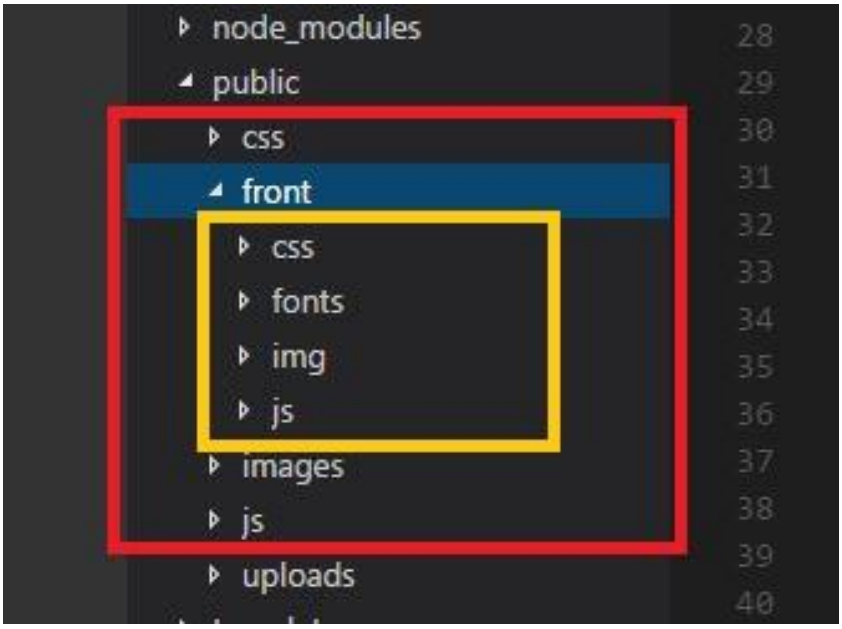
roles.js | Role.js | roles.ejs

```
<thead>
  <th>
    <input type="checkbox" id="checkall" />
  </th>
  <th>Nom du role</th>
  <th>Modifier</th>
  <th>Supprimer</th>
</thead>
<tbody>
  <%  for(role of roles) { %>
    <tr>
      <td></td>
      <td>
        <%= role.ROLENAME %>
      </td>
```

Parcourir la liste des rôles renvoyer par le contrôleur et afficher le nom de chaque rôle.

4. Interface utilisateur

Dans notre projet on a utilisé deux templates une pour l'espace admin et l'autre pour les utilisateurs. On a mis les dépendances des deux templates séparément dans le dossier public.



5. Perspectives

- Duplication de rôles

Cette partie concerne l'affectation de plusieurs rôles à un seul utilisateur, pour le faire on peut procéder de plusieurs façons:

- Créer de nouvelles colonnes dans la table **user** (par exemple ROLEIDSEC pour un rôle secondaire et voire plus…)

- Modifier les cardinalités entre les tables rôle et user pour qu'un utilisateur puisse avoir plusieurs rôles.

- Créer un nouveau **rôle composé** c'est à dire une valeur de plusieurs rôles (par exemple: au lieu de '**MODERATEUR**' on aura '**MODERATEUR PROFESSEUR**' ou bien '**MODERATEUR PROFESSEUR NOUVEAUROLE**' et ainsi de suite.

Le changement ici lors de la condition concernant la vérification de rôle se fera d'une façon à tester si un rôle (qui a certains droits) est bien compris dans la valeur du **rôle composé**.

- Affecter des droits à un rôle

```
/ valide une inscription
outer.get('/valide/:id', ensureAuthenticated, function (req, res) {
   Role.getOne(req.user.user.ROLEID, function (role) {
      if (role[0].ROLENAME === 'ADMIN' || role[0].ROLENAME === 'MODERATEUR') {
         User.Valide(req.params.id, function (users) {
            User.Allu(function (users) {
               res.render('users/users', {users: users})
            })
         })
         User.getOne(req.params.id, function (users) {
            for (user of users) ;
            let MonUser = [
               {
                  username: user.USERNAME,
                  usersurname: user.USERSURNAME,
                  email: user.USERLOGIN,
               },
            ]
            require('../config/emailing')('validation', MonUser);
         })
      } else {
         response.redirect('/')
```

Pour affecter le droit de validation des inscriptions à un nouveau role il suffit d'ajouter le nom du role dans la condition.

Par exemple :

```
if (role[0].ROLENAME === 'ADMIN' || role[0].ROLENAME === 'MODERATEUR' || role[0].ROLENAME === 'Nouveau role')
```

- Captcha

Le Captcha comme il est bien connu, c'est un test de défi-réponse utilisé en informatique, pour s'assurer qu'une réponse n'est pas générée par un ordinateur et que finalement il s'agit bien d'un utilisateur humain.

Ceci existe sur NodeJS grâce au module "**node-captcha**" ainsi qu'il est possible d'intégrer le **Google reCAPTCHA** avec NodeJS.

Liens de l'installation ainsi que la documentation du module **node-captcha**:
https://www.npmjs.com/package/node-captcha
Comment utiliser le **Google reCAPTCHA** avec NodeJS: https://codeforgeek.com/2016/03/google-recaptcha-node-js-tutorial/

- Unit.js

Unit.js est une bibliothèque de tests unitaires open source dédiée au langage de programmation JavaScript.
Il est recommandé de l'utiliser avec NodeJS en **TDD** (Test-driven development ou bien le développement piloté par les tests)

Liens de l'installation ainsi que la documentation: http://unitjs.com/guide/quickstart.html
Petit cours pour débutant: https://hackernoon.com/unit-testing-node-js-38cf2b7e1a41

- NoSQL

Un passage aux bases de données nosql est très recommandé vu leur nature asynchrone partagée, la facilité d'utiliser des objets JavaScript JSON avec la structure du document JSON mongoDB (l'une des bases de données NoSQL les plus populaires).

Liens de l'installation ainsi que la documentation: https://www.npmjs.com/package/nosql
Petit cours pour débutant: https://zestedesavoir.com/tutoriels/312/debuter-avec-mongodb-pour-node-js/