Cabinet médical	**Développement Web avancé** *M2 – MIASHS* A. *Demeure – année 2017/2018*	

Objectifs :

- Implémentation d'un serveur en NodeJS
- Authentication OAuth (via Google, Facebook, …)
- Synchronisation entre plusieurs clients
- Clients Angular 2, IONIC (si possible)
- Mise en place d'une Progressive Web Application (PWA)
- Utilisation de web workers et algorithmes génétiques pour le calcul des trajets les plus courts
- Utilisation d'un composant de cartographie (Google)
- …
- Promises et Observables

Nous allons développer un site de gestion de cabinet médical, côtés serveur et client. Ce cabinet regroupe des **infirmiers** qui doivent rendre des visites à des **patients** pour réaliser des soins. Un **secrétaire** est chargé d'organiser le travail des infirmiers, c'est à dire de leur affecter des patients à visiter.

Nous allons développer dans un premier temps la partie dédiée au secrétaire. L'objectif sera que le secrétaire puisse :

- Ajouter / Modifier / Retirer des infirmiers
- Ajouter / Modifier / Retirer des patients
- Affecter des patients aux infirmiers

Le but du secrétaire est de définir les tournées des infirmiers, il a donc besoin de connaître les adresses des patients, des infirmiers (pour le cas où l'infirmier part directement de chez lui pour commencer sa tournée) et du cabinet médical (les infirmiers doivent y passer avant ou après leur tournée).

Commencez par proposer un modèle de données pour représenter les infirmier, les patients et le cabinet médical. Discutez en avec le professeur avant de passer à son implémentation.

Initialisation du serveur

Dans un premier temps, nous allons développer la partie serveur du cabinet médical. Vous pouvez cloner le dépôt GIT suivant qui contient un squelette de serveur réalisé en NodeJS/Express :

`https://github.com/AlexDmr/M2MIASHS-cabinet.git`

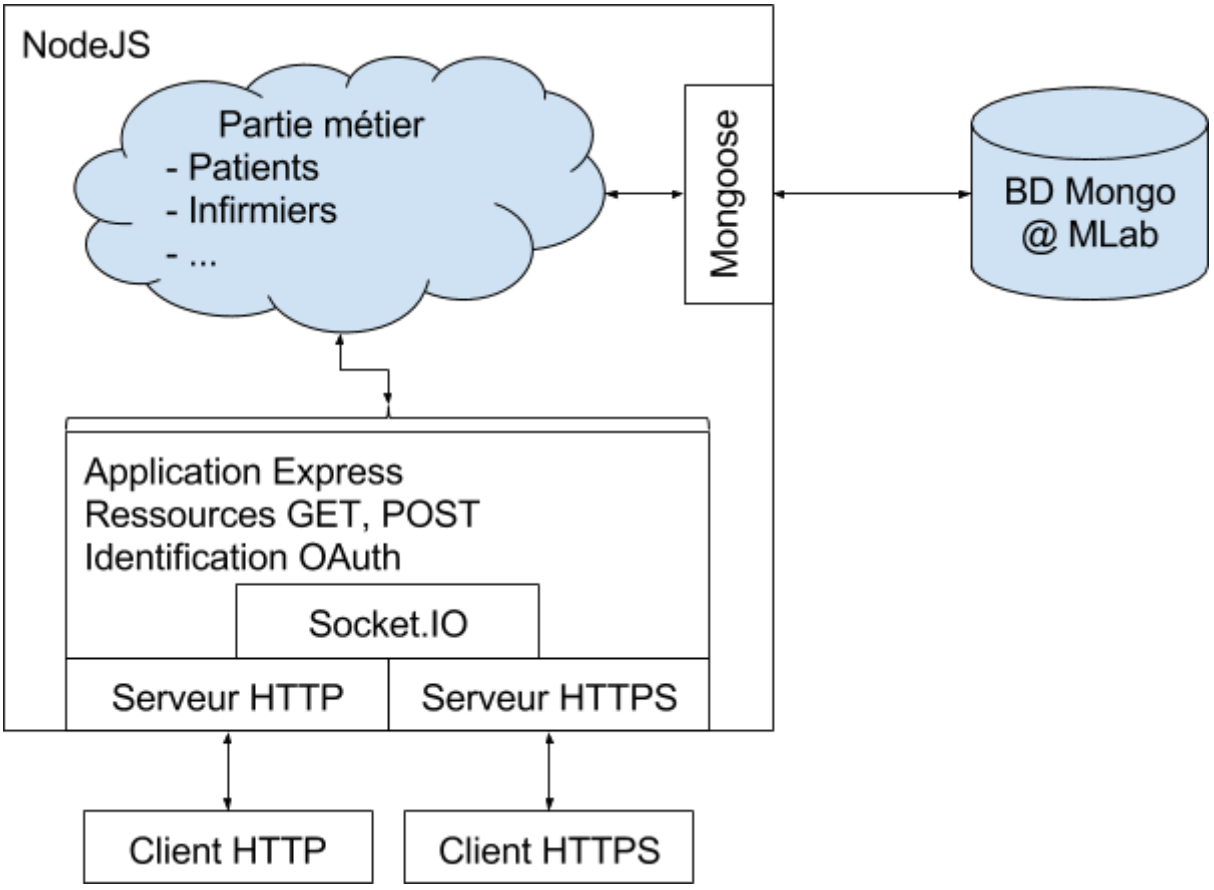
Installez les dépendances (**`npm install`**) puis compilez le projet (**`npm run compile`**).

Lancez ensuite le serveur avec la commande:

```
npm start
```

Testez ensuite que le serveur fonctionne en y accédant via un navigateur. Le serveur est accessible via le protocole HTTP sur le port 8080 et via le protocole HTTPS sur le port 8443.

Le schéma ci dessous représente les différents éléments qu'il faudra coder pour le serveur. Nous commencerons par implémenter la partie métier et des ressources HTTP pour y accéder. Nous ferons ensuite un lien avec une base de donnée MongoDB hébergée par MLab. Enfin, nous implémenterons une authentification au serveur à l'aide d'OAuth. Il s'agira d'identifier le secrétaire à l'aide d'un compte OAuth (fourni par Google ou Facebook par exemple). Nous implémenterons aussi un serveur Socket.IO lorsque nous développerons la partie client.



Nous allons définir un diagramme de classes définissant des patients et des infirmiers.

Prise en main et début d'implémentation d'un serveur HTTP

- Pour tester vos requêtes HTTP POST vous pouvez utiliser :
  - L'utilitaire CURL si vous travaillez sous UNIX.
  - Le plugin Chrome "Advanced REST client".
  - Tout autre logiciel/plugin que vous connaissez capable de faire des requêtes HTTP GET et POST.
- Ajouter une fonction pour que le serveur puisse adresser les fichiers statiques sur répertoire data. (voir http://expressjs.com/fr/starter/static-files.html). Les ressources devront être accessible à l'adresse /data via le serveur. Par exemple, vous devriez pouvoir accéder au logo du cabinet aux l'adresses suivantes :
  http://localhost:8080/data/logo-cabinet.jpg
  https://localhost:8443/data/logo-cabinet.jpg
- Ajouter une ressource programmatique GET /test qui renvoie le texte "OK tout va bien".
- Ajouter une ressource programmatique GET /testParams qui renvoi un texte contenant la liste des paramètres qui lui ont été passé dans l'adresse. (voir l'attribut query de la requête req, http://expressjs.com/fr/api.html#req.query)
- Modifier GET /testParams de sorte à renvoyer une erreur 400 si le paramètre nom ou le paramètre prénom n'est pas spécifié (méthode status de l'objet réponse http://expressjs.com/fr/api.html#res.status). Si l'erreur est levé, le texte devra mentionner le ou les paramètre manquants.
- Même chose mais avec la ressource POST /addPatient. Notez qu'avec les ressources POST, les variables transmises par le clients se trouvent dans l'attribut body de la requête (voir http://expressjs.com/fr/api.html#req.body). Pour le moment on se contente de renvoyer un texte si tout va bien et une erreur HTTP 400 sinon. Cette ressource reçoit en paramètre un ensemble de données permettant la création d'un nouveau patient, à savoir :

name	Son nom
forName	Son prénom
socialSecurity	Son numéro de sécurité social
patientSex	Son sexe (M ou F)
birthday	Sa date de naissance (AAAA/MM/JJ)
adress	Son adresse

En utilisant un Router dans Express, proposez maintenant une implémentation des ressources suivantes:

- GET api/getNurses : Renvoie un tableau contenant les infirmiers.
- POST api/addOrUpdateNurse : permet de transmettre des variables indiquant les informations nécessaires à la création d'un infirmier.

- POST api/deleteNurse : permet de transmettre la variable contenant l'identifiant d'un infirmier pour le retirer.
- GET api/getPatients : Renvoie un tableau contenant les patients.
- POST api/addOrUpdatePatient : permet de transmettre des variables indiquant les informations nécessaires à la création d'un patient.
- POST api/deletePatient : permet de transmettre la variable contenant l'identifiant d'un patient pour le retirer.

Intégration d'une base de donnée No-SQL Mongo

Nous allons maintenant sauvegarder et charger les patients et les infirmiers dans une base de données de type MongoDB.

Rendez vous à https://mlab.com/. Créez vous un compte et une base de donnée.

- Create new database, choisir un fournisseur, ...
- Create database user(s). Créer un utilisateur avec les droits d'écriture et un autre avec le seul droit de lecture.

Pour interagir avec la base Mongo à partir du serveur NodeJS, nous allons utiliser la bibliothèque Mongoose: http://mongoosejs.com

Pour utiliser Mongoose dans votre projet, ajoutez le à la liste des dépendances :

```
npm install --save mongoose
Npm install --save-dev @types/mongoose
```

Dans le répertoire data (ou sources), nous allons créer un fichier mongo.ts qui sera chargé d'assurer la communication avec la base de données MongoDB. Récupérez sur le site de mlab l'URL d'accès à votre base, elle est de la forme:
`mongodb://<dbuser>:<dbpassword>@STRING.mlab.com:PORT/NOM_DE_VOTRE_BASE`

Peuplement de la base de donnée

Commencez par définir un schéma de données pour les patients et un schéma de données pour les infirmier. Ensuite définissez un modèle pour les patients et un pour les infirmiers (voir http://mongoosejs.com/docs/guide.html).

Exemple:

```
const patientSchema = new Schema({
   name            : String,
   forName         : String,
   socialSecurity  : {type: String, unique: true},
   birthday        : String,
   adress          : String
});
const patientModel = model("patients", patientSchema);

const dataPassportUser = {
   passportId      : {type: String, unique: true},
   name            : String,
   emails          : [String],
   photos          : [String],
   provider        : String
};
const dataNurse = Object.assign( {}, dataPassportUser, {
```

```
   adress           : String,
   patientsSSN      : [String]
});
const nurseSchema = new Schema(dataNurse);
const nurseModel  = model("nurses", nurseSchema);
```

Importez Mongoose et redéfinissez les promesses mongoose avec les promesses du standard ES2105 :

```
import * as mongoose from "mongoose";

(<any>mongoose).Promise = Promise;
```

Etablissez la connection avec votre base à l'aide de la fonction connect de Mongoose:

```
connect(url, {useMongoClient: true})
```

Nous allons maintenant lier la création de patients définie dans le code métier à la création de patients dans la base MongoDB. Pour cela, nous allons nous appuyer sur les Observables (voir http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html) et plus particulièrement une sous classe d'observables que sont les Subjects (voir https://github.com/ReactiveX/rxjs/blob/master/doc/subject.md).

Faites un point avec le professeur pour voir comment mettre ça en oeuvre dans votre projet.

Une fois les observables mis en places, utilisez la méthode findOneAndUpdate des modèles définis pour MongoDB (voir la documentation).

Tester le peuplement de la base en utilisant les API HTTP que vous avez définis.

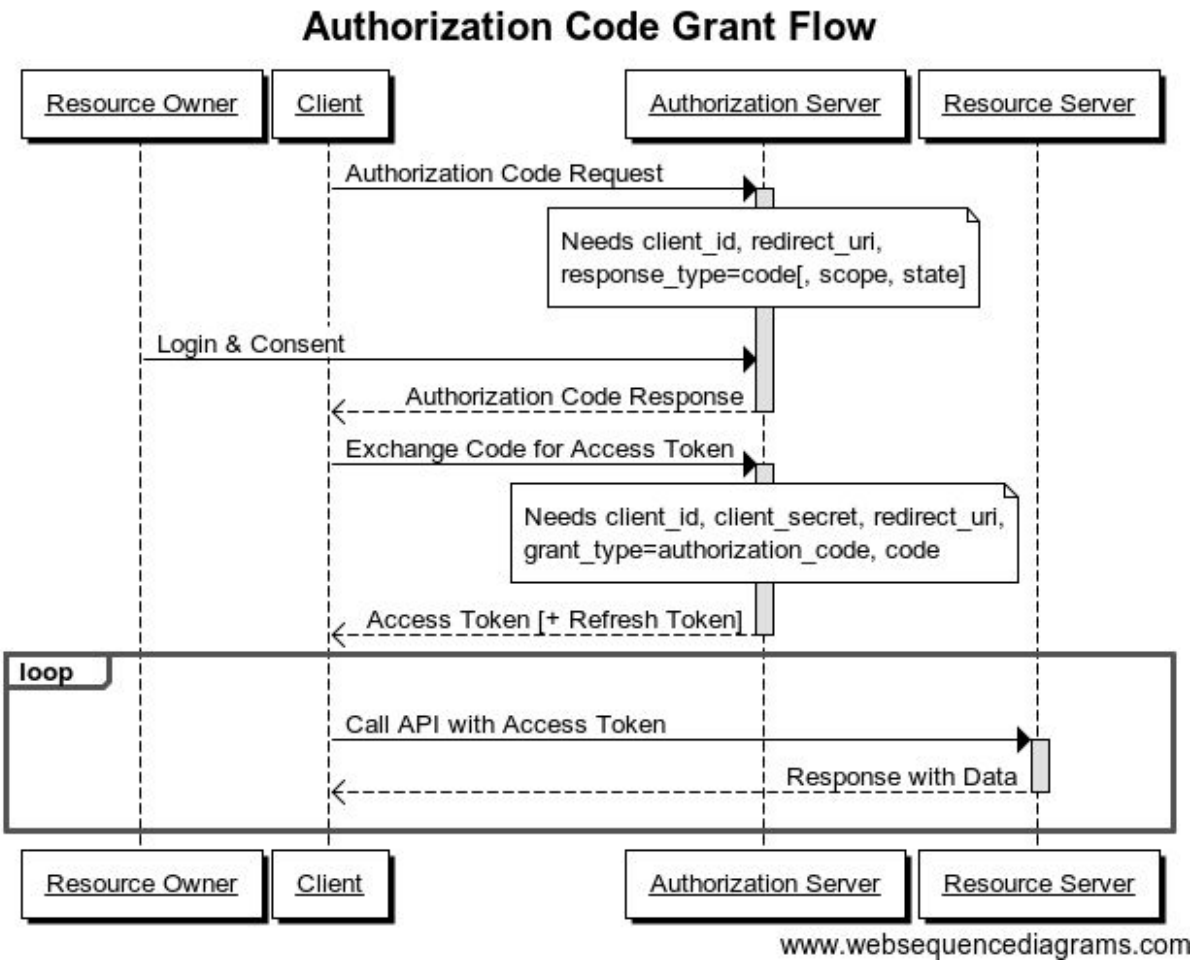
Récupération des données de la base

Nous allons coder la récupération des données à partir de la base MongoDB. Créez une fonction qui sera appelée au lancement du serveur. Cette fonction va être chargé de lire les données représentants les infirmiers et les patients dans la base et de les instancier dans le serveur métier.

Utiliser la méthode find sur les modèles de patient et d'infirmier créés avec Mongoose. A partir des données recueillies, créer les instances de patients et d'infirmiers dans votre serveur métier. Vous serez sans doute amené à utiliser des Promises. Faites valider par le professeur votre solution avant de passer à l'implémentation.

Identification via OAuth

Nous allons gérer la connection au site à l'aide du protocole d'authentification OAuth. Ce protocole est utilisé par Google, Facebook, Twitter, etc. pour gérer l'accès aux données. Nous allons permettre aux utilisateurs de notre système de s'identifier via leur compte Google ou Facebook.



Pour ce faire, nous allons ajouter les dépendances aux bibliothèques passport et cookie-parser au projet :
`npm install --save cookie-parser passport passport-facebook passport-google-oauth express-session`

Spécifiez à express qu'il faut utiliser cookie-parser et passport:

```
import * as passport from "passport";
import * as cookieParser from "cookie-parser";
```

...

```
let sessionMiddleware = session({
   secret: "thisIsAVerySecretMessage",
   resave: true,
   saveUninitialized: true
});

app.use( cookieParser()        );
app.use( sessionMiddleware     );
app.use( passport.initialize() );
app.use( passport.session()    );
```

Vous trouverez la documentation de la bibliothèque Passport ici : http://passportjs.org/docs

Créez un sous répertoire OAuth qui contiendra les fichiers dédiés à l'authentification. Dans ce répertoire, créez trois fichiers :

- OAuth.ts : Ce fichier définira les fonctions utilisables dans le reste de l'application. Il importera les autres fichiers nécessaires à l'authentification via les services Google et Facebook.
- OAuthFacebook.ts : Ce fichier sera chargé de gérer l'authentification via le service fourni par Facebook. Pour utiliser le service d'authentification Facebook, il faut s'enregistrer en tant que développeur: voir https://developers.facebook.com.
- OAuthGoogle.ts : Ce fichier sera chargé de gérer l'authentification via le service fourni par Google. Pour utiliser le service d'authentification Google, il faut s'enregistrer en tant que développeur: https://console.developers.google.com. Dans la partie Identifiants, créez de nouveaux identifiants OAuth. Dans la partie API, activez les API Google Plus et Google Plus Domain.
- PassportUser.ts : Ce fichier définit les utilisateurs du système.

Dans le fichier OAuth.ts, nous allons gérer les utilisateurs connectés:

```
export type PassportUser = {
   id: string,
   name: string,
   token: any,
   emails: string[],
   photos: string[],
   provider: "facebook" | "google";
};

const passportUsers = new Map<string, PassportUser>();
```

Nous allons configurer la bibliothèque passport pour qu'elle gère l'enregistrement des utilisateurs. Nous définissons une fonction RegisterOAuth qui prend en paramètre une application express. Dans un premier temps, nous allons fournir à la bibliothèque passport les fonctions de sérialisation et désérialisation des utilisateurs :

```
export function RegisterOAuth(app: Application) {
   passport.serializeUser( (user: PassportUser, done) => {
       passportUsers.set(user.id, user);
       done(null, user.id);
   });
```

```
    passport.deserializeUser( (id: string, done) => {
        const user = passportUsers.get(id);
        done(null, user ? user : false );
    });
}
```

Pour que ce module soit bien pris en compte, dans le fichier package.json, nous créons on alias en mettant à jour l'attribut _moduleAliases :

```
"_moduleAliases": {
 "@data": "./appJS/data",
 "@OAuth": "./appJS/OAuth"
}
```

Dans un second temps, nous allons définir les ressources nécessaire à la mise en oeuvre de l'authentification via Google (cela sera similaire pour Facebook, voir la documentation sur http://passportjs.org/docs). Pour cela, nous définissons une fonction initOAuthGoogle qui prend en paramètre un objet de configuration ayant pour clefs l'identifiant de votre application Google ainsi que son code secret. Cette fonction indiquera à Express d'utiliser les mécanisme d'authentification de la bibliothèque PassportJS. Elle définira aussi deux ressources accessibles via un routeur :

- **GET /auth/google** : Cette ressource va gérer l'indirection vers la page d'authentification de Google.
- **GET /auth/google/callback** : Cette ressource est appelée après l'authentification Google. Elle redirige vers la ressource / si l'authentification a réussie et vers /login si elle a échouée.

```
import {Router} from "express";
import * as passport from "passport";
import {OAuth2Strategy as GoogleStrategy} from "passport-google-oauth";
…

export function initOAuthGoogle(config: {GOOGLE_CLIENT_ID: string, GOOGLE_CLIENT_SECRET: string}):
Router {
    const {GOOGLE_CLIENT_ID, GOOGLE_CLIENT_SECRET} = config;
    passport.use(new GoogleStrategy({
            clientID: GOOGLE_CLIENT_ID,
            clientSecret: GOOGLE_CLIENT_SECRET,
            callbackURL: `/auth/google/callback`
        },
        (accessToken, refreshToken, profile, done: (err: any, user: PassportUser) => any) => {
            const emails = profile.emails || [];
            const photos = profile.photos || [];
            const user: PassportUser = {
                name: profile.displayName,
                id: profile.id,
                token: accessToken,
                emails: emails.map(val => val.value),
                photos: photos.map(val => val.value),
                provider: "google"
            };
            getOrCreateUser(user); // Vous devez implémenter cette fonction
            done(null, user);
        }
    ));
```

```
    const routerGooglePassport: Router = Router();

    routerGooglePassport.get("/auth/google",
        (request, response, next) => {
            passport.authenticate("google", {scope: ["profile", "email"]})(request, response, next);
        }
    );

    routerGooglePassport.get(
        "/auth/google/callback",
        (request, response, next) => {
            passport.authenticate("google", {
                    failureRedirect: "/login",
                    successRedirect: "/"
                }
            )(request, response, next);
        }
    );

    return routerGooglePassport;
}
```

Appelez ensuite cette fonction avec vos paramètres d'identifications Google. L'appel à cette fonction peut se faire dans le fonction RegisterOAuth du fichier OAuth.ts. Notez que initOAuthGoogle renvoie un Router express qu'il faut donc intégrer à l'application...

Pour utiliser le processus d'authentification, il faut paramétrer Express de la façon suivante :

```
Dans OAuth.ts :
// Check if client is authentified
export function checkIsAuthentified(statusIfFailed: number, ressourceIfFailed: string |
JSON) {
   return (req, res, next) => {
       if (req.isAuthenticated()) { // if user is authenticated in the session, carry on
           next();
       } else { // Error 401 if not authentified
           res.status(statusIfFailed);
           if (typeof ressourceIfFailed === "string") {
               res.redirect(ressourceIfFailed);
           } else {
               res.json({error: "YOU ARE NOT LOGGED IN"});
           }
       }
   };
}



Dans server.ts :
const IdentifiedOrLogin = checkIsAuthentified(401, "/login.html");

// Utilisez ensuite IdentifiedOrLogin conditionner l'accès aux ressource "/" et "/data"
par exemple, vérifiez que cela fonctionne.

// Static files
app.get("/login.html", (req, res) => {
   res.sendFile( path.join(__dirname, PATH_TO_LOGIN_HTML) );
});


Avec login.html un fichier permettant de se connecter via Google, Facebook, etc.
Par exemple :
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>Cabinet médical</title>
</head>
<body>
    <h1>Cabinet médical</h1>
    <section>
        <h2>Identifiez vous via une de ces options</h2>
        <ul>
            <li><a class="logo google"   href="/auth/google"  >Google</a></li>
            <li><a class="logo facebook" href="/auth/facebook">Facebook</a></li>
        </ul>
    </section>

</body>
</html>
```

Quelques références :

- http://www.bubblecode.net/fr/2016/01/22/comprendre-oauth2/
- https://zestedesavoir.com/articles/1616/comprendre-oauth-2-0-par-lexemple/

Trucs et Astuces

In case you encounter problems when running “npm start”, if NodeJS says something like : Cannot find @data/Patient

Then modify you package.json file with:

```
"_moduleAliases": {

        "@data": "./appJS/data"

},
```