Partie serveur :

Choix des technologies :

Nous avons choisi de coder le serveur en Javascript avec Node.js et de faire une base de données NoSql MongoDB. Ces méthodes sont particulièrement utiles pour créer des applications web rapides même avec un grand volume de données. Ces deux solutions ont déjà des bonnes communautés et sont considéré comme l’avenir probable des applications web.

Bien qu’elles ne soient pas encore enseignées à l’eseo, elles représentent un intérêt pédagogique certain. Les quelques pages suivantes vous présenteront la solution ainsi créé et devrait permettre à n’importe qu’elle personne ayant suivant un cursus I1 à l’ESEO de prendre en main ces technologies et de se les approprier.

Création d’un serveur web avec Node.js :

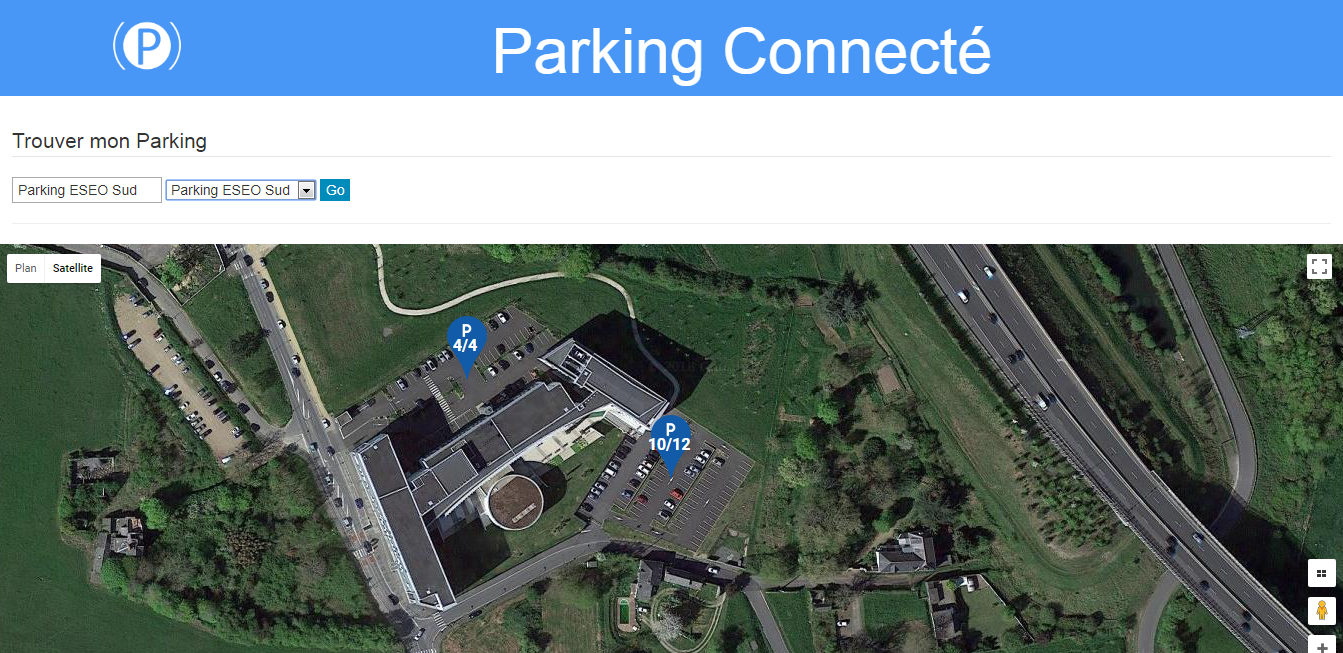
On utilise le framework express pour simplifier la création de nos applications webs.

1. **var** app = express();
3. app.get('/', **function**(req, res) {
4. chargerAccueilPage(req, res);
5. });
7. http.createServer(app).listen(3000);

Ici on associe une fonction de chargement d’accueil au chemin direct ‘/’. Lorsqu’un utilisateur fera une requête ‘/’ au serveur on passera dans la fonction chargerAccueilPage().

Figure 1 : Extrait home.js

Afficher une vue contenant des informations :



Faire communiquer les différents éléments ensemble :

1. **function** chargerAccueilPage(req,res){
2. getData.getEveryIdParking(**function**(err,data) {
3. res.render('home', {data:data});
4. });
5. }

Figure 2 : Extrait de home.js

Fonctionnement :

On appelle la fonction asynchrone getEveryIdParking situé dans getData. Une fois qu’elle aura récupéré et ajouté les informations dans la variable data, on affiche la page ‘home’ dans laquelle on ajoute les informations data sous le nom data, pour les utiliser dans la vue.

Les éléments :

1. **var** DatabaseInterrogation = require('./DatabaseInterrogation');
2. **var** getData = **new** DatabaseInterrogation();

a) Retrouver des informations depuis la base de données :

Dans sa version actuelle (8.9.1), Node.js ne supporte pas encore l’import de fonction, nous créons donc un objet getData dont on pourra appeler les fonctions que l’on a mis dans leur .prototype.

On instancie les propriétés de la base de donnée.

1. **var** DatabaseInterrogation = **function**(){};
3. **var** MongoClient = require('mongodb').MongoClient;
4. **var** url = "mongodb://localhost:27017/mydb";
6. **var** GetElementDatabase = require('./functionGetElementDatabase');
7. **var** fonctionElement = **new** GetElementDatabase();
9. DatabaseInterrogation.prototype.getEveryIdParking = **function**(callback){
10. **var** Bool=**false**;
11. MongoClient.connect(url, **function**(err, db) {
12. **if** (err) **throw** err;
13. db.collection("parking").find({}).toArray(**function**(err, result) {
14. **if** (err) **throw** err;
15. **var** callBackString = {};
16. **var** array = [];
17. **if** (result!=**null**){
18. **for**(**var** i=0; i<result.length; i++){
19. **var** idParkings = {}
20. idParkings.id=result[i].\_id;
21. idParkings.nom=result[i].nom;
22. idParkings.coord=result[i].coord;
23. **var** nbPlacesTotal=result[i].places.length;
24. **var** nbPlacesLibres = 0;
25. **for** (**var** j = 0; j < nbPlacesTotal; j++) {
26. **if** (result[i].places[j].disponibilite){
27. nbPlacesLibres++;
28. }
29. }
30. idParkings.places = nbPlacesLibres +"/"+nbPlacesTotal;
31. array.push(idParkings);
32. }
33. }
34. callBackString.idParkings = array;
35. callback(**null**, callBackString);
36. db.close();
37. });
38. });
40. }
42. module.exports = DatabaseInterrogation;

Autres fonctions permettant de mettre en forme les objets reçus.

On va récupérer tous les parkings contenues dans la base de données pour afficher leurs positions sur la page d’accueil.

Pour chaque élément parking on créé un objet contenant les informations voulues.

On renvoies les informations ainsi récupéré dans un objet list.

Node.js étant asynchrone on renvoie un objet callback contenant les informations. Lorsqu’on appelera cette fonction on attendra d’avoir récupérer les informations.

Figure 3 : Extrait de DatabaseInterrogation.js

Exemple de vue avec données récupérées :

Figure 4 : Exemple de code, complet dans home.ejs

1. <!DOCTYPE html**>**
2. **<html>**
3. **<body>**
4. **<h1>**Vous êtes connecté à **<**%- data.nom %**>h1>**
5. **</body>**
6. **</html>**

Structure de la base de données :

Schéma de la base de données :

On notera que ces schémas on été formulé avant la solution, et des changements mineurs ont été effectué. Des parties comme les horaires et les roles, représentant un intérêt moindre on été écarté par manque de temps mais sont encore présent dans la base de données.

Deplus la base de données étant du NoSql, aucun formalisme n’est vraiment représentatif de ce type de base données. On a choisit de les inclure pour tout de même montrer la structure de la base de données à quelqu’un qui a de l’expérience en Sql mais pas en NoSql.

Schéma Conceptuel au formalisme Merise :

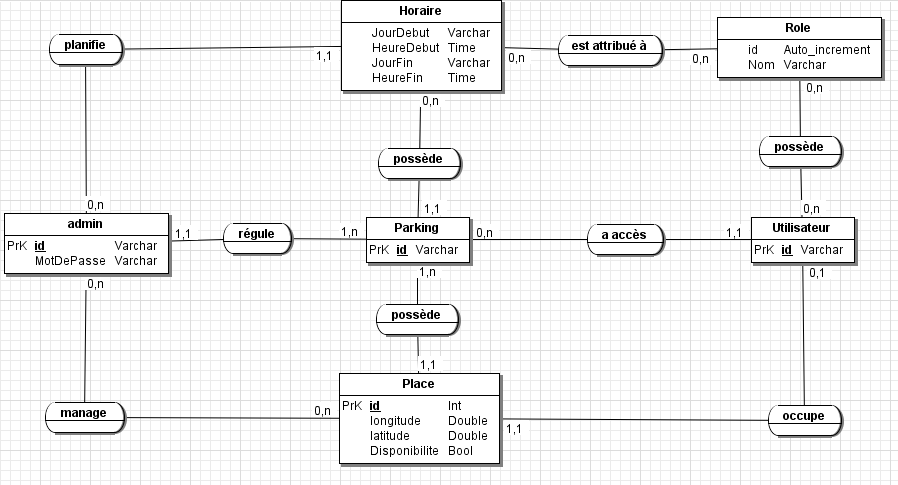
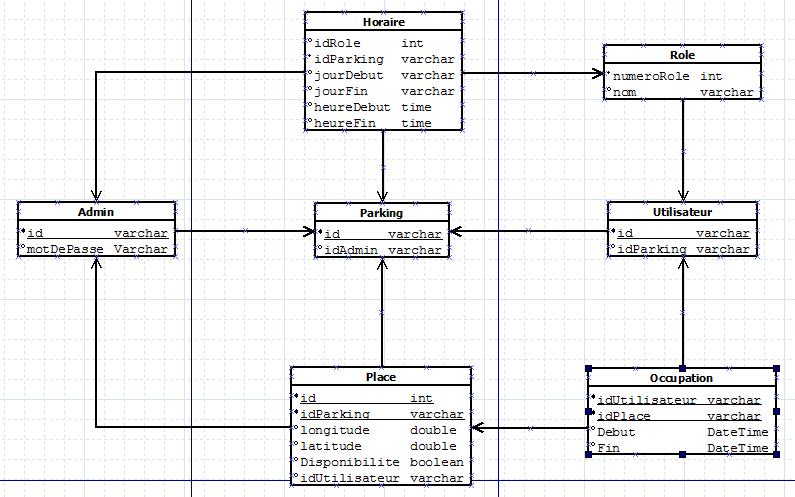


Schéma logique :



Exemple de structure utilisé pour notre base de données, pour un élément dit « parking »

{

"\_id" : ' ',

"id\_lecteur" : "",

"coord" : [0, 0 ],

"nom" : "",

"hash\_mdp" : "",

"etat\_portail" : false,

"places" : [

{

"place\_id" : "",

"addresse\_mac" : "",

"disponibilite" : true,

"utilisateur\_id" : null,

"coord" : [ 0, 0]

}

],

"roles" : [

{

"numero\_role" : 0,

"nom" : "",

"horaires" : [{

"jour\_debut" : "lundi",

"heure\_debut" : "00:00",

"jour\_fin" : "lundi",

"heure\_fin" : "00:00"

}]

}],

"utilisateurs" : [

{

"utilisateur\_id" : "",

"badges" : "",

"place\_id" : null,

"numero\_role" : 0,

"nom" : ""

}]

};

Les bases de données Nosql MongoDB fonctionne en utilisant des documents. A gauche, un document représentant un parking sur notre base de données. On recherche les documents grace au nom des objets.

Afficher une page :

On donne à express le type de vue qu’on utilise et le chemin de nos vues, ici ejs c’est l’une des plus répandues pour Node.js et comme elles se ressemblent énormément ce n’est pas la peine d’expliquer en détail notre choix.

1. app.set('view engine', 'ejs');  //tell Express we're using EJS
2. app.set('views', \_\_dirname + '/views');  //set path to \*.ejs files
4. **function** chargerAccueil(req, res){
5. **if** (req.query.idParking==undefined){
6. chargerAccueilPage(req,res);
7. }**else**{
8. getData.getDataParking(req.query.idParking, **function**(err,data) {
9. **if** (data.bool){
10. res.render('monParking', {data: data});
11. }**else**{
12. chargerAccueilPage(req,res);
13. }
14. });
15. }
16. }

Dans l’exemple à gauche, on vérifie si l’utilisateur a mis en paramètre de l’url ‘idParking’.

-Si oui on charge la vue '/views/monParking.ejs' qui a besoin des données que la fonction getDataParking dans getData renverra.

-Sinon on charge la page d’accueil que l’on a mis dans la fonction chargerAccueilPage() car on l’appelera plusieurs fois.

Fonctionnement du temps réel pour les applications webs : Socket.io

Socket.io est une librairie Node.js qui permet de créer facilement une communication temps réel entre le client et le serveur.

1. io.sockets.on('connection', **function**(socket){
2. socket.on('op', **function**(data){
3. //Notre fonction
4. });
5. });

Ci-dessus l’exemple côté serveur, le programme va écouter s’il reçoit un message intitulé ‘op’ sur les socket client qui sont connecté.

Figure 5 : Exemple de fonctionnement, complet dans home.js

Côté client, sur la vue. On va ouvrir la socket.

Ensuite un exemple de fonction qui émet un message et une autre fonction qui reçoit

Figure 6 : Extrait pageAdmin.ejs, exemple de socket.io côté client

1. <script>
2. **var** socket = io.connect();
3. **function** ouvrirPortail(actualPortail){
4. socket.emit('op', { idPortail: actualPortail, idControleur:'<%- data.idControleur %>' });
5. //alert("<%- data.id %>"+<%- data.id %>);
6. }
7. jQuery(**function**($){
8. socket.on('changementEPidControleur=<%- data.idControleur %>', **function**(data){
9. **if** (document.getElementById("position\_"+data.idPortail)!=**null**){
10. **if** (data.position==0){
11. document.getElementById("position\_"+data.idPortail).innerHTML = "fermé";
12. }**else** **if** (data.position==1){
13. document.getElementById("position\_"+data.idPortail).innerHTML = "ouvert";
14. }**else** **if** (data.position==2){
15. document.getElementById("position\_"+data.idPortail).innerHTML = "ouverture";
16. }**else** **if** (data.position==3){
17. document.getElementById("position\_"+data.idPortail).innerHTML = "fermeture";
18. }
19. }
20. });
21. });
22. </script>

Serveur UDP avec un appareil Raspberry :

On instancie une entité socket utilisant UDP avec le port que l’on souhaite utilisé et l’adresse IP de notre serveur.

Figure 7 : Extrait home.js

1. **var** PORT = 33333;
2. **var** HOST= '172.17.1.43'
3. **var** dgram = require('dgram');
4. **var** serverUDP = dgram.createSocket('udp4');
5. serverUDP.bind(PORT, HOST);
7. **function** sendEncrypt(address, port, message, password){
8. **var** encryptedMessage = encrypt(message, password);
9. serverUDP.send(encryptedMessage,port,address,**function**(error){
10. **if**(error){
11. console.log('Error');
12. }**else**{
13. console.log('Data sent !!!');
14. }
15. });
16. }

Chiffrement de données :

Figure 8 : Extrait de home.js

1. **var** crypto = require('crypto'),
2. algorithm = 'aes-128-ecb',
3. password = 'U93GiADMRixL0agL4iLu2Q==';
5. **function** encrypt(text, pwd){
6. **var** cipher = crypto.createCipher(algorithm,password)
7. **var** crypted = cipher.update(text,'utf8','base64')
8. crypted += cipher.**final**('base64');
9. console.log(crypted);
10. **return** crypted;
11. }
13. **function** decrypt(text, pwd){
14. **var** newText = "" + text;
15. **var** decipher = crypto.createDecipher(algorithm, password);
16. **var** decrypted = decipher.update(newText, 'base64', 'utf8');
17. decrypted += decipher.**final**('utf8');
18. console.log(decrypted);
19. **return**(decrypted);
20. }

On instancie notre objet crypto avec l’algorithme que l’on souhaite utilisé et le mot de passe qui nous servira à chiffre nos messages.

Notre serveur est en utf8, mais les messages que l’on envoie et reçoit sont en binaire ‘base64’.

On fait passer les messages dans le siphon de chiffrement et déchiffrement de la librairie crypto.

Guide de l'installation du programme fourni :

Installation des éléments :

La base de données :

Télécharger sur le site officiel <https://www.mongodb.com> "Community server" (sans inscription) et lancer l'éxecutable.

Documentation d'installation complète à https://docs.mongodb.com/manual/administration/install-community/

Vous devez créer les dossiers /data/db à la racine d'où vous lancez l'application. Par exemple on a dû créer les dossiers C:\data\db sachant que notre application server Mongod.exe se situe dans C:\Program Files\MongoDB\Server\3.6\bin.

Localiser et démarrer l'executable Mongod.exe pour démarrer votre serveur

Node.js :

Télécharger la version recommandée pour la plus part des utilisateurs sur le site officiel https://nodejs.org/en/

*Les commandes suivantes sont a réalisé sur la console de votre ordinateur*

Installation des librairies node.js :

Aller dans le dossier de votre projet pour installer les librairies, ou ajouter -g entre install et l'élément dans les commandes ci-dessous pour les installer globalement.

*npm install ejs*

*npm install mongodb@2.2.33* Certains changement ont eu lieu dans cette librairie, prenez cette version si vous voulez faire marcher le projet que nous vous avons donner.

*npm install socket.io*

Ajout des éléments exemples sur la base de données :

*node demo\_create\_mongo\_db.js* Création de la collection

*node insert\_Donnees\_de\_test.js* Ajout des documents

Démarrage de l'application :

*node home.js*