

Université de Nice Sophia Antipolis

IoT & Réseaux

Rapport de projet

Meteo



Enseignants : Monsieur G.MENEZ

Étudiants :Alaedine KAROUIA, Valentin BORDY, Guilhem FABRE,Yohann LAURENDEAU, Mathieu BIRGER

Introduction :	4
Architecture:	5
La partie Back-end :	6
Partie script d'insertion (écoute du broker):	9
Partie esp32 :	9
Interface utilisateur :	9
Usability de l'application :	9
Use Case:	10
Utilisateur Invité:	11
Utilisateur connecté	18
Intégration des données:	20
MongoDB	20
Collection data	21
Collection ESP_USER	22
Collection USER_DATA	24
Bonus:	25
Script NodeJS, l'écoute du topic MQTT	25
API externe	26
L'ESP32	27
Schéma de la station météo	27
Code	28
Doc capteurs et initialisation	28
Paramètres utilisateur	29
La configuration du client MQTT:	31
L'envoi des données	31
Système LED	32
Simulation de données	32
Sécurité:	33
Sécurité des flux de données:	33
JWT JSON Web Tokens:	33
Sécurité de la Connexion:	35
Sécurité du VPS:	36
Pare-feu avec UFW et Fail2Ban	36
SSL/TLS	37
Reverse proxy	38
Broker MQTT	39
Mise en place et déploiement sur le VPS	39
Mise en place du VPS	40
Apache (LAMP)	41

Node et PM2	41
MongoDB	41
Certificat SSL/TLS avec certbot	42
Mise en place du reverse proxy	42
Broker MQTT Mosquitto	43
Déploiement	43
L'application web	44
L'API node/express avec pm2	45
Le script node d'écoute du broker MQTT	46
Le script de simulation d'esp	47
Améliorations futures pour le déploiement	47
Bonus: Installation et lancement du projet en local	48
Api nodeJS	48
Application Web VueJS	49
Script d'insertion/écoute du broker MQTT et script de simulation	49
Environnement de travail pour l'ESP32	50
Amélioration	51

Introduction :

Notre projet a pour but de créer une application web sur la météo qui permettra aux utilisateurs d'observer les données météorologiques grâce à une API météo dans les villes en France métropolitaine. De plus, l'utilisateur peut se connecter et ajouter des Esp « Electronic stability program » à une certaine adresse afin d'obtenir les données météo depuis cette localisation et les partager aux utilisateurs.

				devenu météo	
birger	mathieu.birger	21907552@un	TD01	1	Installation/livraison de l'objet, Architecture ESP
bordy	valentin.bordy	21904041@un	TD01	1	Sécurité, Déploiement, Integration de donnees
fabre sauterey	guilhem.fabre-	21605108@un	TD01	1	Usability de l'application, Analyse fonctionnelle
karouia	alaedine.karou	21611332@un	TD01	1	Sécurité, Usability de l'application
laurendeau	yohann.lauren	21709669@un	TD01	1	Architecture, Documentation

Architecture:

Une bonne architecture de notre application est un élément important afin de mieux comprendre les les objectifs de notre application, de mieux travailler le côté code derrière, de vérifier les fonctionnalités ou d'approfondir nos services.

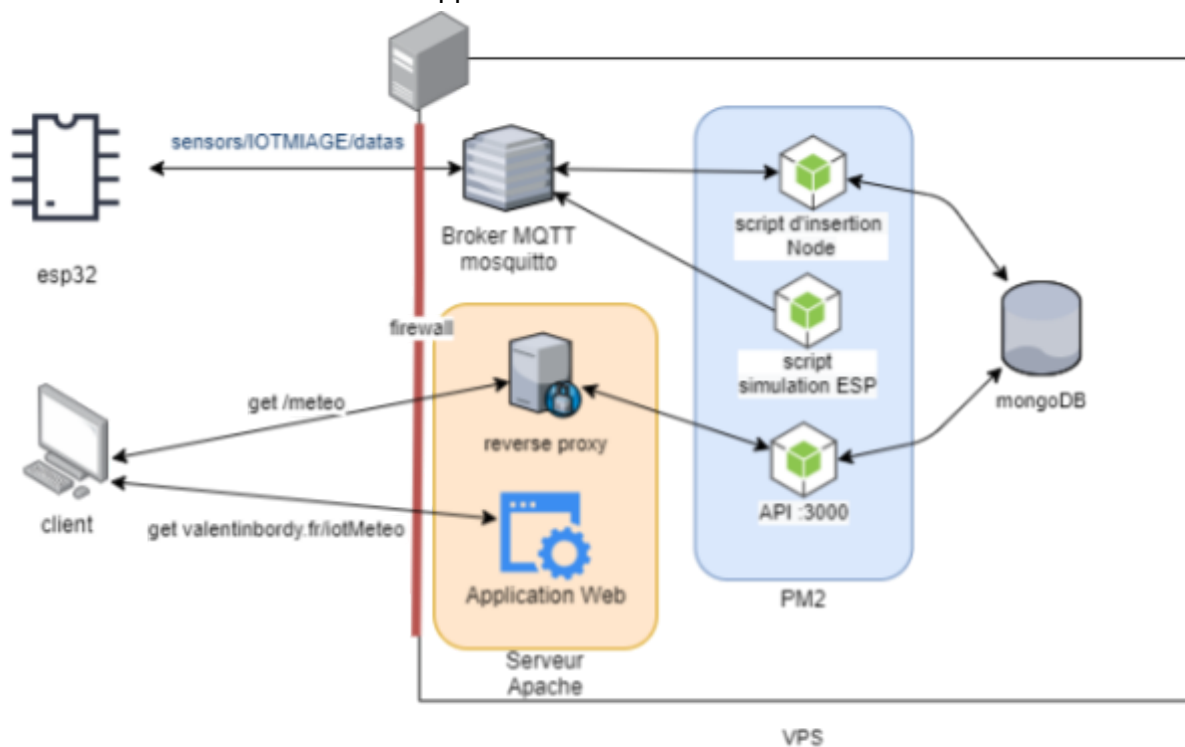
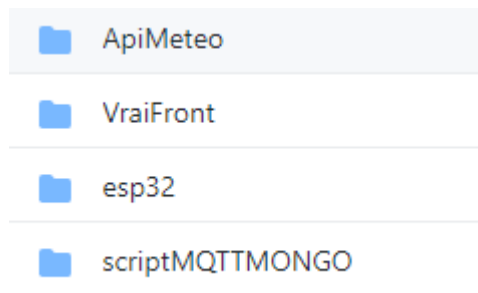


Schéma Architecture projet

Pour notre projet, nous avons décidé de diviser l'architecture en plusieurs parties comme vous pouvez le voir sur la photo ci-dessous :



Pour la suite, je vais donc rentrer dans les détails de notre architecture.

La partie Back-end :

Pour le back, nous avons besoin d'une api afin de gérer une base données. Nous avons donc utilisé Node js comme langage et Mongodb comme base de donnée. De plus, nous avons choisi d'utiliser Mongoose car cela fournit une solution simple basée sur un schéma pour modéliser les données d'application. Il comprend le casting de type intégré, la validation, la création de requêtes.

Exemple:

```
const mongoose = require('mongoose');

//structure de donnee d'une adresse Esp
const adresseEspSchema = mongoose.Schema({
  lng: { type: Number, required: true },
  lat: { type: Number, required: true },
});

//structure de donnee d'un esp
const EspSchema = mongoose.Schema({
  adresseMac: { type: String, required: true },
  adresse: {
    type: adresseEspSchema, required: true
  },
  userId: { type: String, required: true }
});
```

Nous avons un fichier serv2.js qui utilise Express. Cela nous permet de lancer le serveur en écoutant sur un certain port , de faire la redirection vers les différentes routes. Ce fichier contient aussi la connexion à la base de données.

```
// DB connection
mongoose.connect(DB_URI,{
  useNewUrlParser: true,
  useUnifiedTopology: true
}, (err) => {
  if(err) console.log(err)
  else console.log('Connected to the database')
})
```

Les dossiers config nous permettent de stocker les informations clé comme celle de notre base de données ou celle des API qu'on a utilisées.

```
//environnement de dev
const uri = 'mongodb+srv://iot:root@cluster0.efowh.mongodb.net/METEO?retryWrites=true&w=majority';
//environnement de prod
//const uri = 'mongodb://localhost:27017/METEO';
module.exports = uri;
```

```
//fichier de config pour l'api openWeatherMap
const openWeather_url = "https://api.openweathermap.org/data/2.5/";
const openWeatherKey = 'd0f74ba55214c45401d7ae1941791222';

module.exports = {
  openWeather_url,
  openWeatherKey
};
```

Exemple des différentes routes qui concernent la météo :

```

const meteoController = require('../controllers/meteo');
const express = require('express');
const router = express.Router();
const token = require('../auth');

//GET
router.get('/', meteoController.getMeteo);
router.get('/adresseMac/:id',meteoController.getMeteoById);
router.get('/freshData/:id', meteoController.getFreshMeteoById);
router.get('/openWeatherMeteo/:adress',meteoController.getMeteoOpenWeatherByAdress);
router.get('/prevision/:adress', meteoController.prevision);
router.get('/previsionbyid/:id' ,token,meteoController.previsionbyId);

//POST
router.post('/verif' ,meteoController.verificationDonnes);

module.exports = router;

```

Puis, pour les contrôleurs, ils contiennent différentes méthodes qui permettent de modifier des données dans la base de données ou pour du traitement de données.

Finalement, nous avons d'autres fichiers tels que le .env qui contient le code JWT et le fichier auth.js qui est utilisé lors de la connexion d'un utilisateur.

Il permet de vérifier le header des requêtes et de vérifier la présence du token ou non afin d'en autoriser l'accès.

```

const jwt = require('jsonwebtoken')

function auth(req,res,next){
  const token = req.header('x-auth-token');
  try {
    ///
    if(!token) {
      res.status(401).json({ message : "Connexion impossible, token manquant", code: res.statusCode });
    }else{
      req.user = jwt.verify(token, process.env.JWT_SECRET);
      next();
    }
    ///
    // next();
    ///
  } catch (e) {
    res.status(400).json({error : 'Token invalide', code: res.statusCode});
  }
}

```


Partie script d'insertion (écoute du broker):

Pour cette partie, on possède un script qui contient des topics qui récupèrent les données envoyées depuis l'esp et les traite afin de savoir si on l'ajoute à notre base de données MongoDB, pour la suite, ce serait expliqué en profondeur dans la partie Intégration de données".

Partie esp32 :

Concernant cette partie, nous avons le script de l'esp avec ses différentes méthodes pour se inscrire à un topic ou des accesseurs pour renvoyer la température et autres données.

Interface utilisateur :

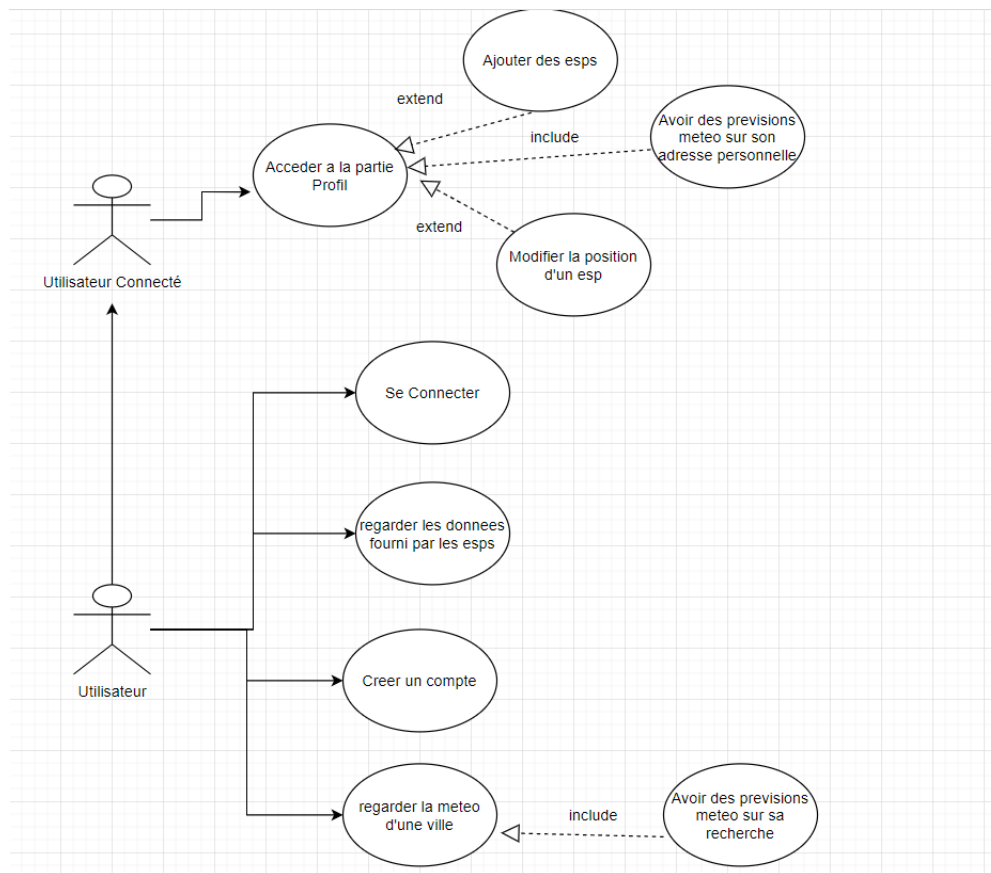
Concernant l'interface utilisateur, nous avons voulu utiliser un framework javascript dont notre groupe était familier et que l'on maîtrisait le vuejs.

Le Vue.js étant un framework très répandu pour les utilisateurs du langage Javascript qui permet de découper notre front sous forme de Component. Nous avons également pris appui sur une librairie VUETIFY qui nous permettait de construire notre interface grâce au Material Design. Pour plus de détails, veuillez vous diriger vers la partie "Usability de l'application".

Usability de l'application :

Pour notre application, nous avons effectué un diagramme de cas d'utilisation afin de présenter toutes les actions possibles de l'utilisateur. De plus, dans notre application, il y a une page Guide qui sert d'instructions aux nouveaux utilisateurs.

Use Case:



Lorsque un utilisateur arrive sur la page d'accueil de l'application, il aura tout d'abord 1 choix à faire parmi 3. Il aura le choix de se connecter, de se créer un compte ou de se connecter en tant qu'invité :

SE CONNECTER	S'ENREGISTRER	INVITÉ
<input type="text" value="E-mail"/>		
<input type="password" value="Mot de passe"/>		
<div>0</div>		
<div>CONNEXION</div>		

Utilisateur Invité:

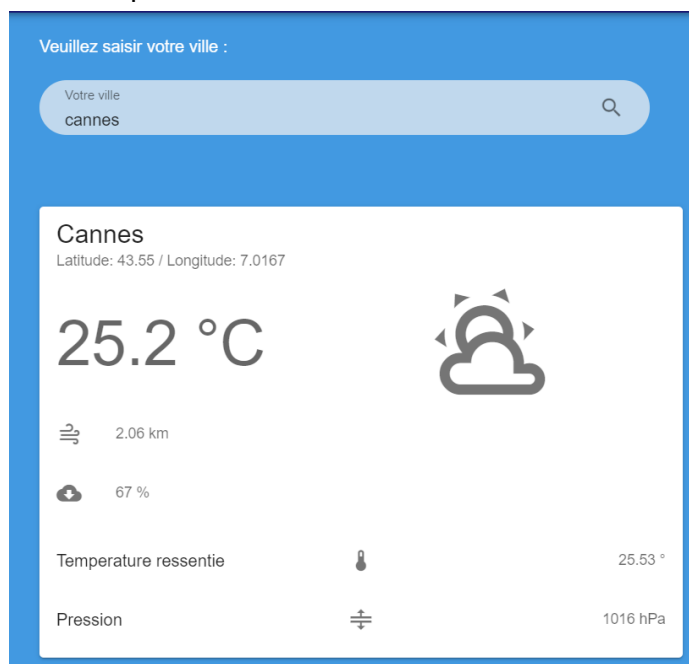
On peut choisir de se connecter ou de se créer un compte:

On envoie alors notre adresse mail et votre mot de passe. Le mot de passe est crypté grâce à md5.

Puis du côté de l'api on vérifie si les données existent dans la base de donnée et si cela correspond on accède à l'application avec le statut connecté. Dans le retour de la connexion de l'api on récupère un jwt token. Celui-ci nous permettra de pouvoir faire des requêtes qui étaient auparavant pas autorisées à les faire désormais.(voir partie sécurité des flux pour plus de détails).

Si on se connecte en tant qu'invité, nous avons accès à la page Map, qui nous permet d'effectuer une recherche sur une ville pour connaître ses données météorologiques et ses prévisions mais aussi de voir les ESP présents sur la carte.

Par exemple si on effectue une recherche sur la ville de Cannes :



Cela nous donne aussi la prévision à cette adresse :

[VOIR LES PRÉVISIONS](#)

La météo aujourd'hui sera pluvieux

La météo dans 1 jour sera nuageux

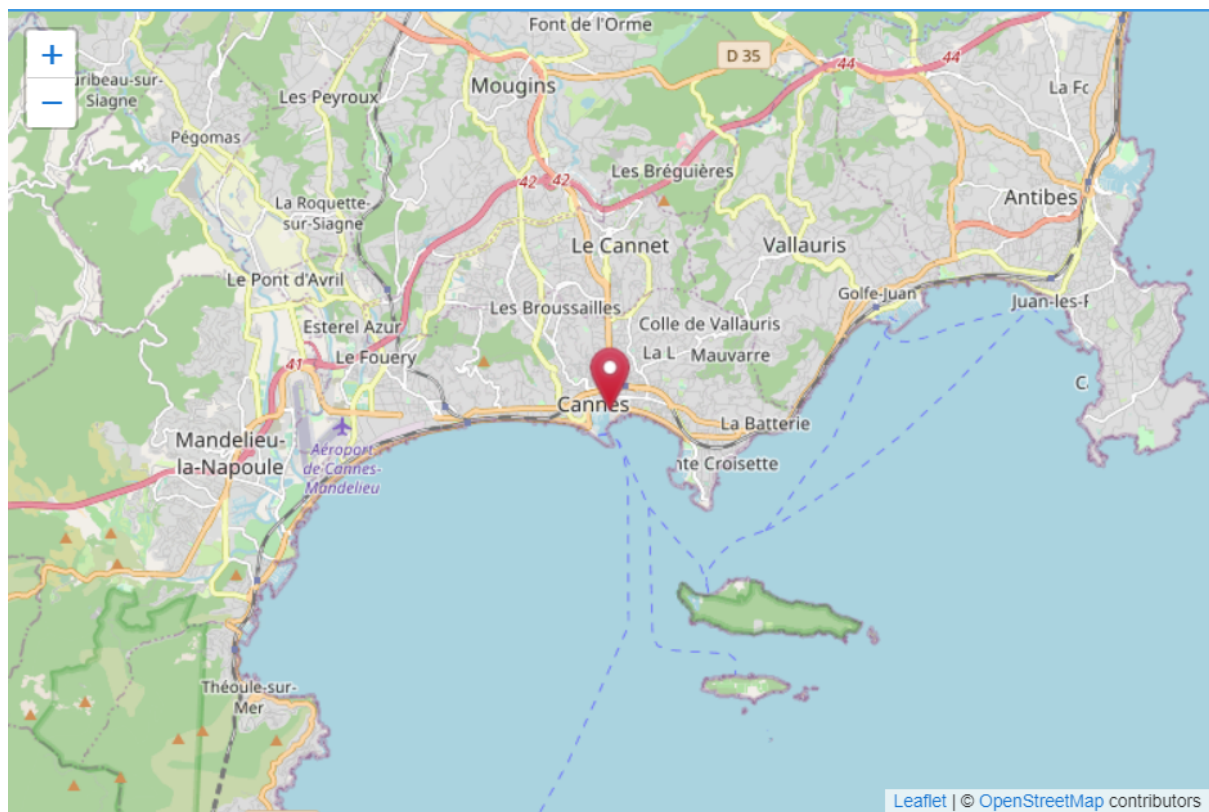
La météo dans 2 jours sera pluvieux

La météo dans 3 jours sera nuageux

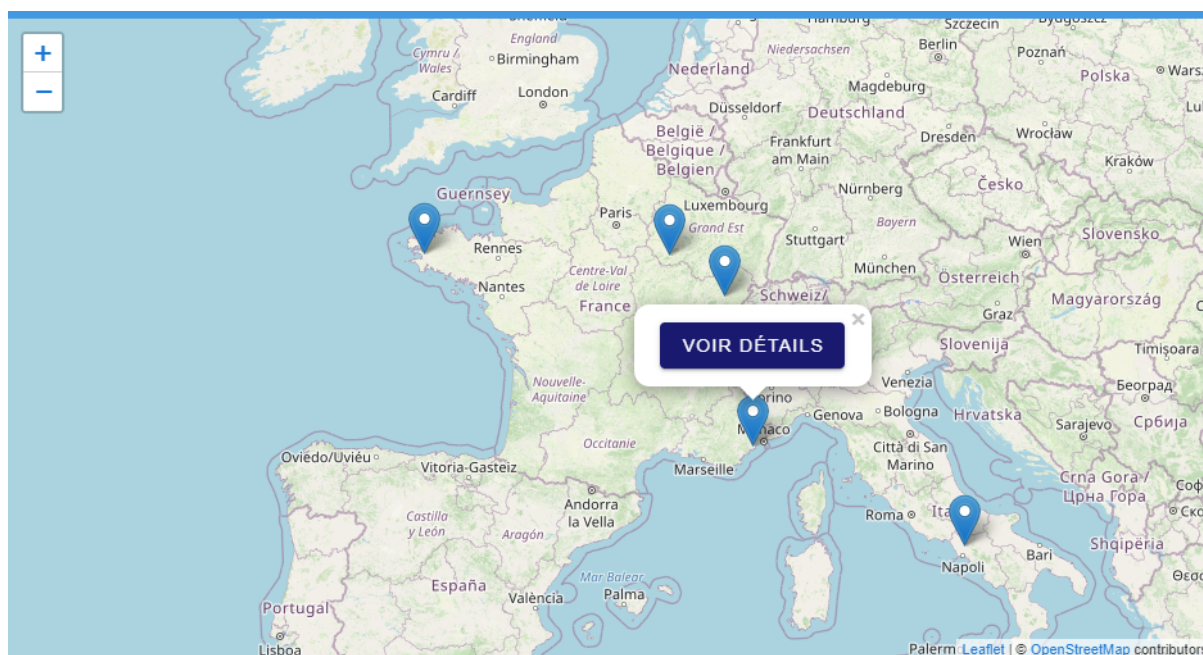
La météo dans 4 jours sera nuageux

CLOSE

Puis on peut apercevoir que sur la carte, un marqueur rouge se crée à l'endroit de la recherche :



Mais on peut également observer plusieurs marqueurs bleus qui correspondent à des Esp et nous permettent d'accéder aux données météorologiques de ceux-ci.



Nous observons plusieurs marqueurs bleus sur la carte, qui ne sont pas le résultat d'une de nos recherches, mais les emplacements des esp placer ici et là par les différents utilisateurs. Nous pouvons obtenir les détails météorologiques de chacun en cliquant dessus.

Cela ouvre un dialogue contenant les informations partagées de l'Esp.

En entête on retrouvera le nom de l'utilisateur qui a déployé cet ESP. Puis une ligne qui nous indique si les données récoltées par l'utilisateur sont à jour . On détermine par la date de la dernière valeur récoltée par l'ESP si cette dernière est la même que le jour actuel les données sont à jour ou non. On vérifie la fiabilité des données en affichant avec un marqueur rouge ou vert selon la qualité des données (voir la partie sur la vérification des données). On peut le graphique des données en fonction du temps. On peut choisir le type de donnée à afficher.

Nous avons aussi un récapitulatif les jours précédents concernant la température, l'humidité et la pression. On y affiche la moyenne de la journée sinon un icône rouge si l'on a pas de donnée sur ce jour.

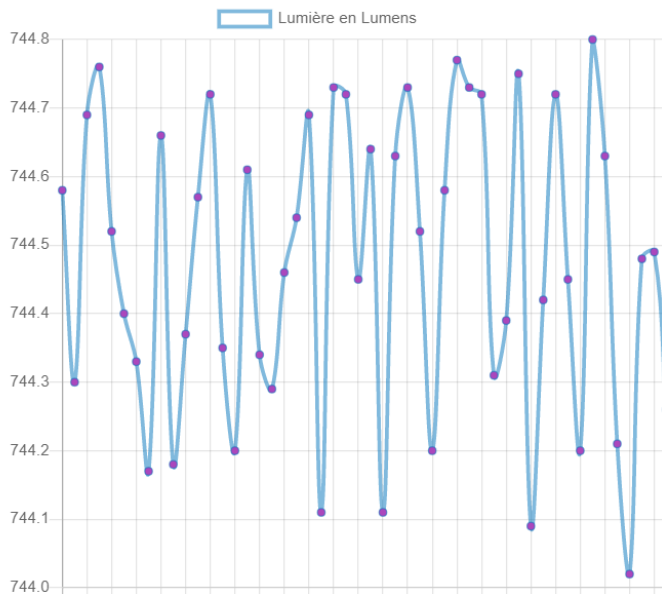
On retrouve aussi l'altitude de l'ESP.

Graphiques de votre recherche

Esp de l'utilisateur: Buffa Michel

Données à jour

Altitude : 198.98 m



Type	Fiabilité
temperature	!
pression	!
humidite	✓

Le graphique ci-dessus retransmet les valeurs récoltées par l'ESP choisit sur le graphique. On va avoir en ordonné les valeurs de la variable choisie et en abscisse la date et l'heure à laquelle elle a été récoltée. On va pouvoir changer la variable entre la lumière, la pression, l'humidité et la température grâce aux boutons situés en dessous. On va aussi pouvoir changer le nombre de valeurs affichées sur la courbe.

 Valeur

Lumière

Pression

Humidité

Température

50

Nb de Val

On peut également observer la moyenne des données par rapport à l'historique des derniers jours, par exemple, ici avec l'humidité :

Historique des 4 derniers jours :

Type/Date	2021-06-08	2021-06-07	2021-06-06	2021-06-05
humidite	!	!	!	54.07

Pour cela on va stocker la somme des valeurs ayant la même date dans une variable et diviser cette dernière par le nombre de valeurs. Ce qui va nous rendre notre moyenne à la date voulue.

A noter que si l'esp n'a pas enregistré de donnée à un jour J on affichera l'icône ! rouge pour le signaler comme ci dessus.

On peut également observer si les données de l'esp sélectionné sont fiables ou non. Pour cela un encart est affiché à droite de la courbe affichant la température, la pression et l'humidité avec un icon exprimant la fiabilité de cette donnée. Si la donnée est fiable on affichera un icon vert si elle ne l'est pas un icon rouge.

A noter que l'on ne gère pas la fiabilité de la lumière qui est beaucoup trop aléatoire en fonction de la position de l'esp.

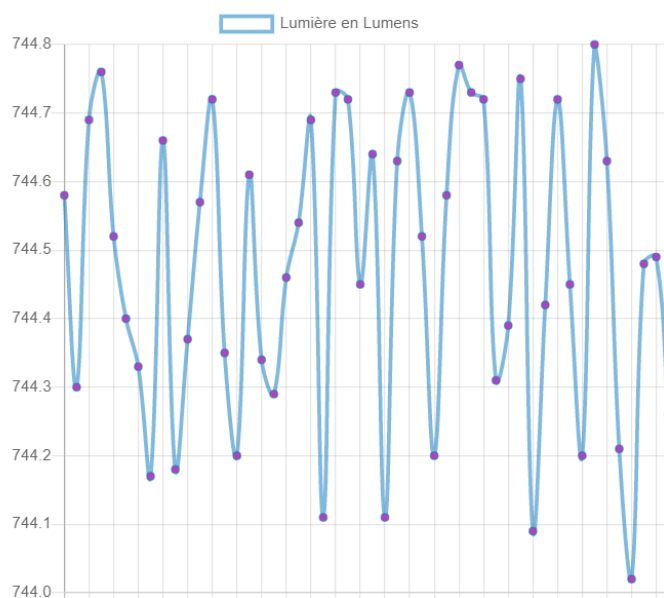
Par exemple :

Graphiques de votre recherche

Esp de l'utilisateur: Buffa Michel

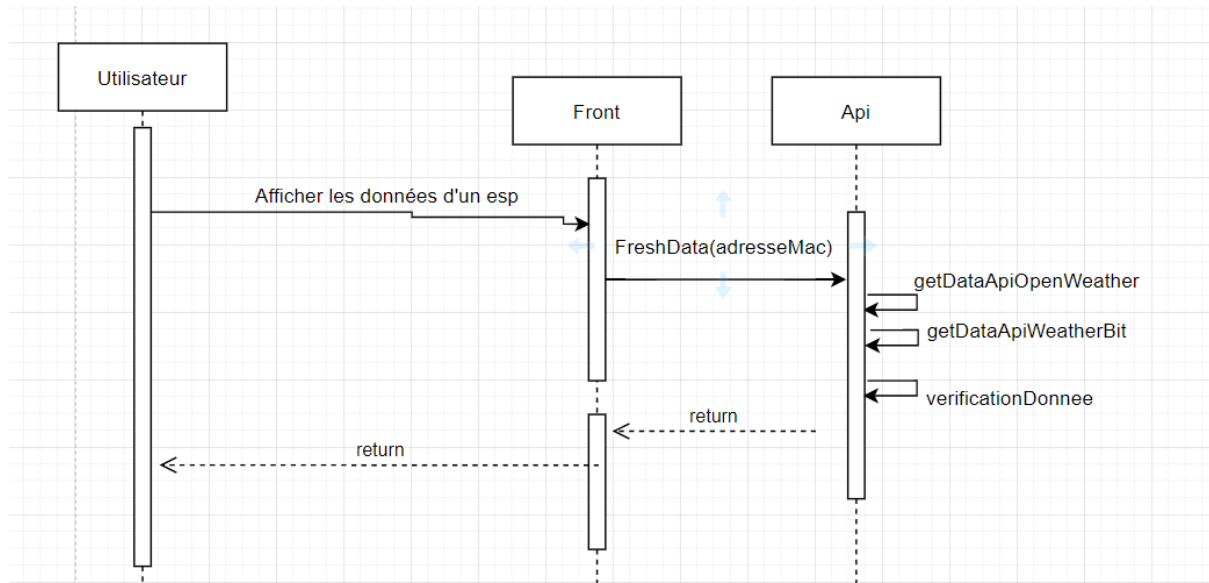
Données a jour

Altitude : 198.98 m



La fiabilité de l'information est gérée en comparant la dernière valeur de l'esp sélectionné avec les valeurs de 2 API, OpenWeather et WeatherBit. Pour cela on va vérifier si au moins une API à ses valeurs qui concorde avec celles de l'ESP tout en gardant une marge d'erreur (de 10% de plus ou de moins pour l'humidité, de 2.5°C de plus ou de moins pour la température et de 5 hPa de plus ou de moins pour la pression).

Diagramme de séquence de la vérification de la fiabilité des données:



Lorsqu'un utilisateur va demander d'afficher les détails d'un Esp cela va appeler directement la méthode FreshData qui va retourner les 7 000 dernières données ou toutes les données qui ont moins de 2 semaines (voir le code ci-dessous). Nous faisons cela pour éviter de surcharger le site lors du chargement des données. Une fois les données chargées nous allons les afficher sous forme du diagramme, calculer l'historique, l'afficher et faire la vérification des données. Nous chargeons donc les données des 2 API et appliquons les méthodes de test (voir code ci-dessus) à ces derniers. Ce qui va nous retourner un booléen si les conditions sont valides.

```

//permet de récupérer les dernières datas de l'esp
const getFreshMeteoById = async (req, res) => {
  //maximum de data par requete
  const maxData = 7000;
  const addMac = req.params.id;
  const today = new Date();
  today.setDate(today.getDate() - 14);

  //find permet de chercher dans le MeteoModel
  //limit permet de limiter le nombre de données à recup
  //sort permet de trier par date -1 signifie qu'on recup les données depuis la dernière insérée
  await MeteoModel.find({id: addMac, date: {$gte: today.toISOString("sv-SE", {timeZone: "Europe/Paris"})}}).limit(maxData).sort({ date: -1 })
    .then(rslt => {
      //reverse permet d'inverser le tableau
      rslt.reverse();
      rslt.length ? res.status(200).json(rslt) : res.status(200).json({erreur: "ESP inconnu ...."})
    })
    .catch(err => {
      res.status(400).send({message: err.message});
    })
}

```



```

function testHumidity(espH1, h2) { //Permet de comparer les données d'humidité des api et celle de l'esp
  //console.log('humid : ', espH1, h2)
  return (espH1 <= h2 + 10) && (espH1 >= h2 - 10);
}

function testTemp(espT1, t2) { //Permet de comparer les données de la température de l'api et des esp
  //console.log('temp : ', espT1, t2)
  return (espT1 <= t2 + 2.5) && (espT1 >= t2 - 2.5);
}

function testPression(espP1, p2) { //Permet de comparer les données de pression de l'api et des esp
  //console.log('pression: ', espP1, p2)
  return (espP1 <= p2 + 5) && (espP1 >= p2 - 5);
}

```

Nous les affichons ensuite dans le tableau grâce à l'objet ci-dessous qui utilise les méthodes précédemment expliquées ce qui remplira les champs de l'objet à False ou à True.

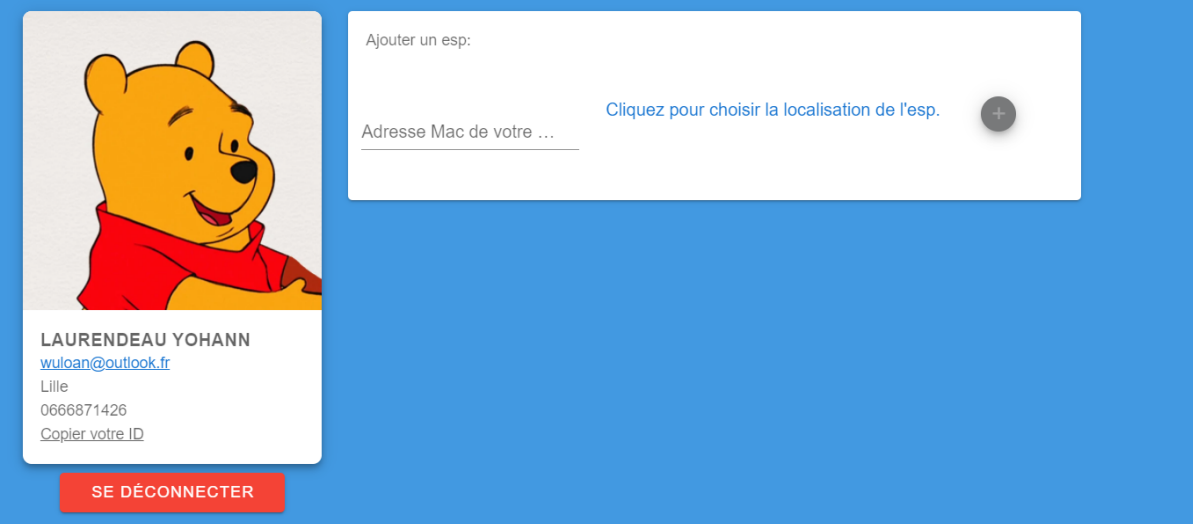
```

let objetValid;
try {
  objetValid = {
    temperature: testTemp(esp.temperature, dataApi1.data[0].temp) || testTemp(esp.temperature, dataApi2.main.temp),
    pression: testPression(esp.pression, dataApi1.data[0].pres) || testPression(esp.pression, dataApi2.main.pressure),
    humidite: testHumidity(esp.humidite, dataApi1.data[0].rh) || testHumidity(esp.humidite, dataApi2.main.humidity)
  }
} catch (e) {
  try {
    objetValid = {
      temperature: testTemp(esp.temperature, dataApi2.main.temp),
      pression: testPression(esp.pression, dataApi2.main.pressure),
      humidite: testHumidity(esp.humidite, dataApi2.main.humidity),
    }
  } catch (e) {
    let message = "Verification indisponible pour le moment. La limite de vérification a peut-être été atteinte.";
    res.status(400).send(message);
  }
}

```

Utilisateur connecté

Si on se connecte en tant qu'utilisateur, nous accédons tout d'abord à la page profil qui contient nos informations et nous permet d'ajouter un esp avec sa localisation et son adresse mac :



Ajouter un esp:

Cliquez pour choisir la localisation de l'esp.

Adresse Mac de votre ...

LAURENDEAU YOHANN
wuloan@outlook.fr
Lille
0666871426
[Copier votre ID](#)

SE DÉCONNECTER

On récupère la longitude et la latitude du marqueur sur la carte .

Cliquez sur la carte pour ajouter la localisation de votre esp :



Grâce à une règle de syntaxe regex ,on autorise que les adresses qui respectent ce format.

```
adressMacRules: [  
  (v) => !!v || "l'adresse mac est requises",  
  (v) =>  
    /^[0-9A-Fa-f]{2}[:]){5}([0-9A-Fa-f]{2})$/ .test(v) ||  
    "L'adresse mac n'est pas valide.", //permet de tester avec un regex si l'adresse mac a un bon format  
],
```

On récupère l'id utilisateur grâce aux variables de session et on envoie l'objet contenant l'adresse mac , l'adresse de l'esp et l'id de son propriétaire.

Intégration des données:

L'intégration des données est un point important dans notre projet, en effet sans données nous ne pouvons fournir les services souhaités.

Comme dit précédemment, nos données sont envoyées par un esp sur un topic MQTT qui est sensors/IOTMIAGE/datas ce topic est lié au broker MQTT Mosquitto que nous avons installé sur le VPS. Ce topic est écouté par un script node géré par pm2 (un gestionnaire de processus node). Si le script valide la donnée on l'ajoute à la bd MongoDB (on reviendra plus en détail sur le script).

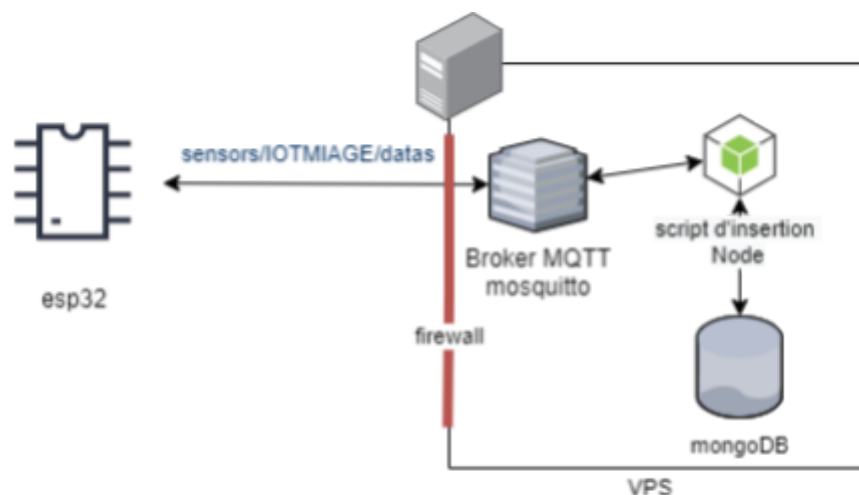


Schéma explication insertion esp

MongoDB

Avant de vous présenter la structure que nous avons choisie pour MongoDB et ses collections, on peut se demander pourquoi le choix de mongoDB et pas un autre SGBD SQL ou noSQL ? Et bien c'est très simple mongoDB est un SGBD noSQL qui stock des objets JSON, ce que nous souhaitons pour faciliter les différentes couches (c'est une orientation documents). Mongo offre une très bonne performance avec beaucoup de données, et offre une flexibilité pour les futures mise à jour d'applications (ajout du vent par exemple).

Notre base mongo nommé "**METEO**" possède 3 collections :

- **datas** (données de l'esp)
- **ESP_USER** (données qui fait la liaison entre l'utilisateur et les datas)
- **USER_DATA** (données de l'utilisateur)

Ces collections ont chacune un validateur propre a elle même.

Chaque validateur à le niveau de validation dit "moderate", car des données ont pu être insert avant que ces validateurs existent, le niveau "strict" est un plus pour une collection "neuve".

Pour se renseigner plus précisément sur les validateurs MongoDB n'hésitez pas à consulter cette page de la doc officiel : <https://docs.mongodb.com/manual/core/schema-validation/>

Collection data

La collection **datas** est la collection qui répertorie **toutes les données** de tous les **esp** qui envoient des données. Ces données sont traitées et modifiées à travers un script Node (le script scriptMQTTMONGO/node_iot_prod.js).

elle est représentée par le validateur suivant :

```
validator: {
  $jsonSchema: {
    bsonType: "object",
    required: ["id", "userId", "lumiere", "pression", "altitude", "humidite",
"temperature", "date", "adresse"],
    properties: {
      id: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      userId: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      lumiere: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      pression: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      altitude: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      humidite: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      temperature: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      date: {
        bsonType: "string",
        description: "must be a string and is required"
      }
    }
  }
}
```



```

validator: {
  $jsonSchema: {
    bsonType: "object",
    required: ["adresseMac", "adresse", "userId"],
    properties: {
      adresseMac: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      adresse: {
        bsonType: "object",
        required: ["lat", "lng"],
        properties: {
          lat: {
            bsonType: "number",
            description: "must be a number and is required"
          },
          lng: {
            bsonType: "number",
            description: "must be a string and is required"
          }
        }
      },
      userId: {
        bsonType: "string",
        description: "must be a string and is required"
      }
    }
  }
}

```

Un exemple de données qui va avec :

```

{
  "_id" : ObjectId("60b9e28b8f8f119a3da94178"),
  "adresseMac" : "25:6F:68:69:C9:14",
  "adresse" : {
    "_id" : ObjectId("10b9e28c8f8f119a3da94179"),
    "lat" : 43.65707029201576,
    "lng" : 7.135788121548403
  },
  "userId" : "60b2aa582ea85f6b509d5435",
  "__v" : 0
}

```

Dans notre exemple vous avez sûrement remarqué la présence d'un champ "`__v`", ce champ est ajouté par mongoose (utilisé sur l'API node/express) lors de la première création d'un document par mongoose.

Collection USER_DATA

La collection **USER_DATA** est la collection qui répertorie **toutes les données** de chaque **utilisateur**.

elle est représentée par le validateur suivant :

```
validator: {
  $jsonSchema: {
    bsonType: "object",
    required: ["userFirstName", "userLastName", "userEmail", "userPassword",
"userAddress"],
    properties: {
      userFirstName: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      userLastName: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      userEmail: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      userPassword: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      userAddress: {
        bsonType: "string",
        description: "must be a string and is required"
      }
    }
  }
}
```

Un exemple de données qui va avec :

```
{
  "_id" : ObjectId("60b4b5a12e90b515474251eb"),
  "picUrl" :
"https://media.discordapp.net/attachments/807286057460039690/847914836046118983/
E2Gi0QHIXIAQtK1n.png",
  "userFirstName" : "MICHEL",
  "userLastName" : "BUFFA",
  "userEmail" : "micbuffa@gmail.com",
  "userPassword" : "b6edd10559b20cb0a3ddaeb15e5267cc",
  "userPhoneNumber" : "0640597461",
}
```



```

    "userAddress" : "Cannes",
    "__v" : 0
  }

```

Dans notre exemple vous avez sûrement remarqué la présence d'un champ "__v", même explication qu'au-dessus, ce champ est ajouté par mongoose (utilisé sur l'API node/express) lors de la première création d'un document par mongoose.

Bonus:

Pour **créer** une **nouvelle collection** avec un **validateur** faire :

```

db.createCollection("nomCollection", {
  validator: {
    ....
  }
})

```

Pour **ajouter** un **validateur** à une **collection existante** :

```

db.runCommand( {
  collMod: "nomCollection",
  validator: {
    ....
  },
  validationLevel: "your_validation_level" //moderate ou strict
} )

```

Script NodeJS, l'écoute du topic MQTT

Après avoir vu la base de données MongoDB nous allons nous intéresser aux données et comment elles sont insérées.

Comme dit précédemment le script écoute le topic sensors/IOTMIAGE/datas du broker.

Le script tourne en boucle grâce à pm2 un gestionnaire de processus node permettant de lancer un script et de ne jamais l'arrêter (ce qui est notre cas). Mais que se passe-t-il lors de la réception d'une donnée ?

Avant d'expliquer voici un exemple de données envoyé sur le topic :

```

{
  "id":"7F:F2:21:BD:01:78",
  "userId":"60abb8d105b5ec078018b3b4",
  "lumiere":"644.17",
  "pression":"2055.04",
  "altitude":"499.86",
  "humidite":"14.41",

```

```
"temperature": "14.48"  
}
```

Lorsque le script reçoit une donnée il va dans un premier temps voir si la donnée est bien du JSON et si c'est le cas il va la parser.

ensuite il va faire une requête au serveur mongo pour vérifier si dans la collection ESP_DATA un document possède la même adresse mac que la donnée reçu et le même userId, si c'est le cas alors on peut insérer la donnée en ajoutant l'heure actuelle et l'adresse qu'on récupère dans la collection ESP_DATA. Mais pourquoi faire cette vérification ? Au bout de quelques jours de développement on s'est retrouvé avec des données ESP associée à aucun utilisateurs (inutilisable par l'application web), ce qui augmentait la taille de la base de données considérablement avec des données mortes.

Une problématique que nous n'avons pas résolu est la problématique de la taille des données hormis cette vérification qui réduit le nombre de documents ajoutés en base, il faut demander une limite par esp pour éviter d'avoir des données d'un ou deux ans qui ne serviront pas (hormis pour un historique). Une solution serait de limiter le nombre de données par ESP (imaginons 13 000 données météo). Lorsqu'on attendrait cette donnée, on supprimerait la donnée la plus vieille pour ajouter la plus récente et ainsi faire -1 +1 et garder toujours 13 000 données météo par ESP.

API externe

Même si nous n'intégrons pas les données de l'api dans notre base de données MongoDB cette partie est importante et n'est pas à négliger. il est important de souligner que l'application web ne se base pas que sur les données des esps.

Pour notre projet nous utilisons 2 apis externes:

- OpenWeatherMap
- WeatherBit

Ces 2 apis ont besoin d'une clef pour pouvoir fonctionner.

OpenWeatherMap est une API gratuite (dans la limite de 60 appels/minutes ou 1M appels/mois ou illimité et gratuit pour les étudiants pendant 6 mois) qui récolte ses données via un réseau de stations météorologiques mondial. Nous l'utilisons pour les prédictions météorologiques, le temps actuel et la vérification de nos données (données de l'ESP vérifié via l'api).

Dire qu'openWeatherMap est une API est un peu faux, openWeatherMap fournit plusieurs API comme : Current Weather Data, Hourly Forecast 4 days, Solar Radiation API, etc... Pour notre part nous utilisons l'api **Current Weather Data** pour obtenir le temps actuel sur la page d'accueil, nous l'utilisons aussi pour la vérification des données (la fiabilité des données).

Nous utilisons aussi **5 day weather forecast** d'openWeatherMap pour avoir les prédictions météo sur 5 jours.

WeatherBit est aussi une plateforme d'api gratuite (dans la limite de 500 appels/jour). On utilise **Current weather data** pour la vérification des données si les données de OpenWeatherMap ne sont pas fiables (voir l'explication au dessus).

L'ESP32

L'ESP32 du projet est découpé en deux parties, la partie électronique et la partie code

Vous pouvez retrouver ci dessous le schéma électrique

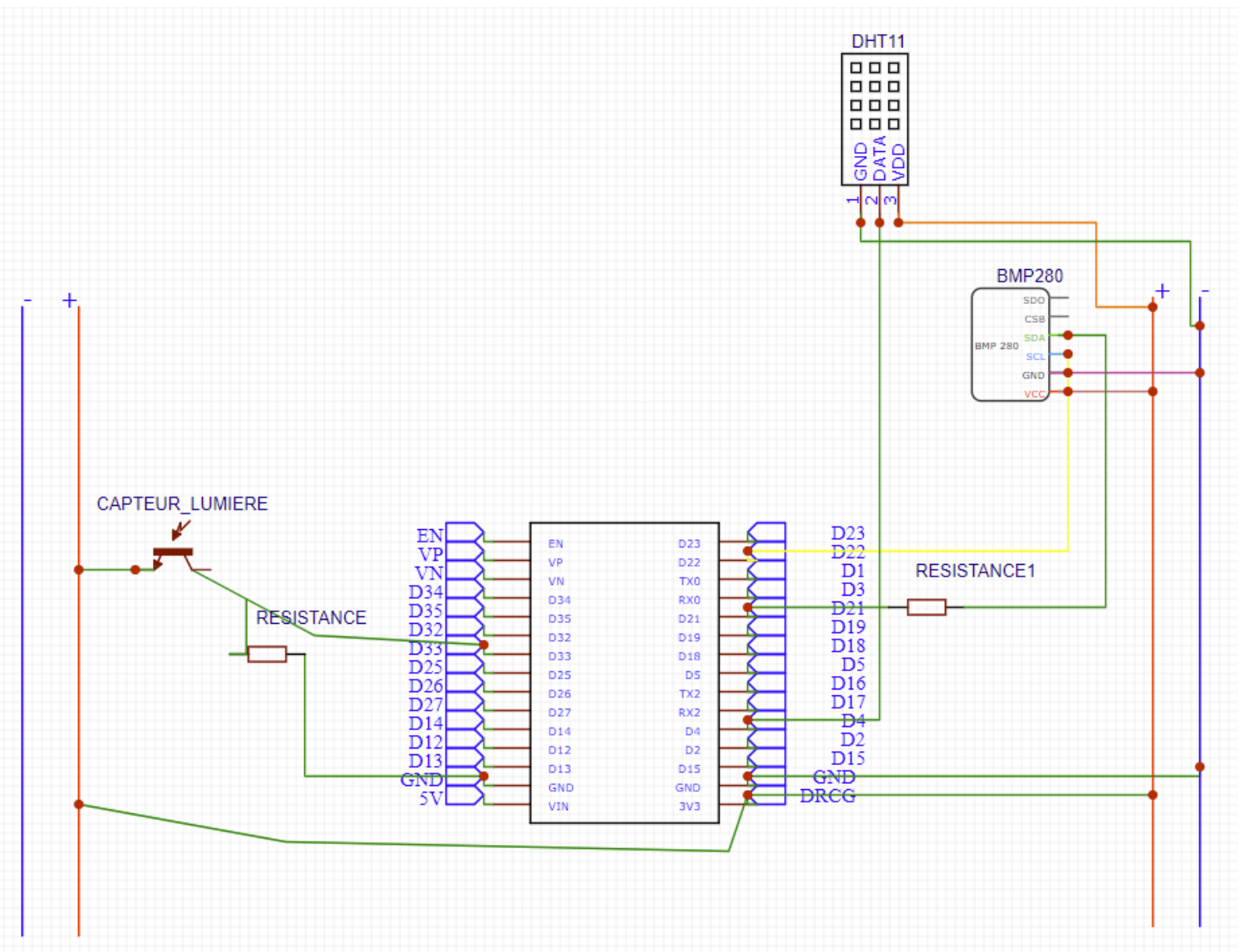


Schéma de la station météo

Code

Doc capteurs et initialisation

L'ESP récupère les informations de la lumière avec le capteur fourni par la fac, le capteur DHT11 sert à récupérer l'humidité et le capteur BMP280 donne la température, la pression et l'altitude.

La documentation du capteur DHT11 est disponible sur ce lien <https://www.gotronic.fr/pj2-35212-2265.pdf>

La documentation du capteur BMP280 est disponible sur ce lien: <https://www.gotronic.fr/pj2-sen-ky052-manual-21-05-19-2006.pdf>

Une des problématiques rencontrée sur cette partie à été la soudure du capteur BMP280 (obligé de souder, l'utilisation du capteur sans soudure entraîne des faux contacts).

Le matériel de soudure n'était pas adéquat pour de la soudure si précise ce qui a entraîné des difficultés.

Concernant le code, nous avons gardé la base du projet météo vu en TD et l'avons étoffé de fonctionnalités (en plus d'avoir intégré la gestion des nouveaux capteurs)

Voici le bout de code pour initialiser le capteur BMP280

```
if (!bmp.begin(0x76)) {  
    Serial.println(F("Could not find a valid BMP280 sensor, check wiring  
or "  
    "try a different address!"));  
    while (1) delay(10);  
}  
//conf capteur bmp  
bmp.setSampling(Adafruit_BMP280::MODE_NORMAL,      /* Operating Mode.  
*/  
    Adafruit_BMP280::SAMPLING_X2,          /* Temp. oversampling */  
    Adafruit_BMP280::SAMPLING_X16,         /* Pressure oversampling */  
    Adafruit_BMP280::FILTER_X16,          /* Filtering. */  
    Adafruit_BMP280::STANDBY_MS_500); /* Standby time. */
```

A savoir que pour trouver quelle information rentrée dans les parenthèses :

```
if (!bmp.begin(0x76))
```

Il faut suivre ce tutoriel :

https://www.circuitschools.com/interfacing-16x2-lcd-module-with-esp32-with-and-without-i2c/#Code_to_get_the_I2C_address

Cela permet d'obtenir l'adresse I2C qui peut être différente même si le pin utilisé est le même que sur le schéma électrique plus haut.

L'initialisation du capteur DHT11 est plus simple:

```
#define DHTTYPE DHT11
uint8_t DHTPin = 4;
DHT dht(DHTPin, DHTTYPE);
```

Paramètres utilisateur

Afin de permettre à l'utilisateur de changer les paramètres de son ESP32 sans avoir à claquer en dur les valeurs, nous avons mis en place une interface permettant de changer en local les différents paramètres tel que le SSID, la fréquence d'envoi ou encore le mot de passe WIFI.

Pour ce faire, il faut appuyer sur le bouton boot qui est écouté par un listener:

```
Button button1 = {0, 0, false};

//Quand le bouton est pressé on appel cette méthode qui va changer
l'état du bouton, on peut alors savoir si celui-ci a été appelé et
changer le comportement de l'ESP
void IRAM_ATTR isr() {
    button1.numberKeyPresses += 1;
    button1.pressed = true;
}

int buttonState = 0;

void setup () {
    attachInterrupt(button1.PIN, isr, FALLING);
}
```

Ensuite dans les différents endroits où le code boucle on va regarder si le bouton boot a été pressé ou non, si c'est le cas alors c'est que l'utilisateur veut configurer son ESP32, on change alors le mode WIFI

```
WiFi.mode(WIFI_STA);
connect_wifi();
```

Et on envoie la page web au client qui se connecte au WIFI émis par l'ESP32 et qui se rend sur la page 192.168.4.1

On envoie la page contenant le formulaire (de l'html dans un String)

Pour faciliter la mise en page nous avons utilisé un

```
char index_html[] = R"rawliteral()
```

Nous voulions insérer l'adresse MAC de l'ESP32 dans cette page, étant donné que le type `R"rawliteral()`

ne permet pas de faire cela avec de la concaténation il nous a fallu passer par des templates :

```
String processor(const String& var)
{
    if(var == "TEMPLATE")
        return F(whoamiChar);
    return String();
}

void loop () {
    if (!configured){
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){
    request->send_P(200, "text/html", index_html, processor);
});
    }
}
```

Quand le `R"rawliteral()` rencontre "TEMPLATE" il remplace la chaîne "TEMPLATE" par l'adresse MAC.

Une fois configuré, l'utilisateur appuie sur valider ce qui renvoie à l'ESP32 une requête get, on regarde alors ce qu'elle contient et on sauvegarde dans la mémoire morte les différents paramètres en passant par une librairie prévue à cet effet.

Exemple pour le paramètre `SSID`

```
server.on("/get", HTTP_GET, [] (AsyncWebServerRequest *request) {
    String inputMessage;
    int inputMessageFreq;
    String inputParam;

    //pour chaque if on enregistre le paramètre dans la mémoire morte
    si le paramètre a bien été changé
    if (request->hasParam(SSID)) {
        inputMessage = request->getParam(SSID)->value();
        if (inputMessage[0] != '\0')
        {

            inputParam = SSID;
        }
    }
}
```

```

        preferences.begin("credentials", false);
        preferences.putString("ssid", inputMessage);
        Serial.println(inputParam);
        preferences.end();
    }

```

La configuration du client MQTT:

```

WiFiClient espClient; // Wifi
PubSubClient client(espClient) ; // MQTT client
const char* mqtt_server = "ip_vps";
#define TOPIC "sensors/IOTMIAGE/datas"
#define TOPIC_LED "sensors/aaa/led"
void setup () {
    client.setServer(mqtt_server, 8883);
}

void loop()
{
    if(client.connect("esp32", mqttUser, mqttPassword )) { // Attempt to
connect
        Serial.println("connected");
        client.subscribe(topic);
    }
}

```

L'envoi des données

```

    payload = "{\"id\": \"" + whoami + "\", \"userId\": \"" + userID +
    "\", \"lumiere\": \"" + get_light() + "\", \"pression\": \"" +
    get_pressure() + "\", \"altitude\": \"" + get_alt() + "\", \"humidite\":
    \"" + get_humidity() + "\", \"temperature\": \"" + get_temperature()
    + "\"\" + \"}\"";
    payload.toCharArray(data, (payload.length() + 1));

    Serial.println(data);
    client.publish(TOPIC, data);

```

Système LED

Nous avons mis en place un système de code visuel à l'aide de la LED, celle ci clignote plus ou moins vite suivant ce qu'il se passe :

Elle reste allumée pour indiquer que le mode configuration est activé, elle clignote de manière régulière pour indiquer qu'elle cherche à se connecter à un réseau WIFI et elle clignote deux fois rapidement toute les 5 secondes pour indiquer qu'elle est connectée au réseau mais n'accède plus au broker MQTT.

Voici un exemple de code pour allumer la LED :

```
#define ONBOARD_LED 2
digitalWrite(ONBOARD_LED,1);
```

Simulation de données

Pour simuler des données nous avons réalisé un script écrit en javascript afin d'en créer une instance sur notre VPS.

Cette instance permet de stocker des ESP avec des valeurs étalons :

```
var esp = {id:"28:A5:21:5A:BE:17", userid:"60b2aa212ea85f3b509d5435",
lumiere:600, pression:1017, alt:200, hum:50, temp:22};
var esp2 = {id:"28:30:D1:73:E2:AB", userid:"60b2fa582ea85f3b509d5435",
lumiere:750, pression:1117, alt:199, hum:60, temp:20};
var esp3 = {id:"78:15:14:51:B2:51", userid:"60b2ba552ea85f3b509d5434",
lumiere:200, pression:1057, alt:220, hum:70, temp:21};
var esp4 = {id:"7F:F2:21:BD:01:51", userid:"60a2b8d105b5ec078018b3a3",
lumiere:650, pression:2057, alt:500, hum:20, temp:15};
var esps = [esp,esp2,esp3,esp4];
```

On créer un client que l'on va connecter à notre broker MQTT

```
var mqtt = require('mqtt');
console.log("connection");
var client = mqtt.connect('mqtt://ip_vps:8883',{clientId:"scriptounet",
username:"<username_mqtt>",
password:"<password_mqtt>"});

client.on("connect",function(){
  console.log("connection");
  onConnect();
})
```

Une fois connecté on envoie pour chaque ESP un string au format JSON contenant les données :


```

while (true)
{
    //temps entre chaque envoie en l'occurrence 10min
    await sleep(600000);
let i = 0;
    for (var esp in esps) {
        console.log("envoie");
        console.log(esps[i]);
client.publish("sensors/IOTMIAGE/datas",getDataRandomData(esps[i]),1);
i++
    }
}
function getDataRandomData(esp){
    return"{\"id\": \"\" + getID(esp) + "\", \"userId\": \"\" + userID(esp)
+ "\", \"lumiere\": \"\" + get_light(esp) + "\", \"pression\": \"\" +
get_pressure(esp) + "\", \"altitude\": \"\" + get_alt(esp) + "\",
\"humidite\": \"\" + get_humidity(esp) + "\", \"temperature\": \"\" +
get_temperature(esp) + \"\" + \"}\"";
}

```

Exemple de variation aléatoire de la valeur étalon :

```

function get_pressure(esp){
    let res = Math.random() * (esp.pression+2+0.8- esp.pression-2) +
esp.pression-2;
    if(res <0)
        res=0;
    return res.toFixed(2)
}

```

On peut ainsi facilement simuler l'envoi massif de données provenant de divers ESP32!

Sécurité:

Sécurité des flux de données:

JWT JSON Web Tokens:

Pour la sécurité de nos données, nous avons choisi d'utiliser des jwt token afin de protéger les routes de notre api.

Un JSON Web Token est une proposition de norme Internet pour la création de données avec signature facultative et/ou cryptage facultatif dont la charge utile contient du JSON qui

affirme un certain nombre de revendications. Les jetons sont signés à l'aide d'une clé publique/privée.

En effet, grâce à cela nous restreignons certaines routes de notre api et donc nous protégeons notre base de données d'extraction de données non désirée.

Par exemple, il n'est pas possible d'ajouter un esp a la base de donnée sans avoir ce token.

```
const jwt = require('jsonwebtoken')

function auth(req,res,next){
  const token = req.header('x-auth-token');
  try {
    ///
    if(!token) {
      res.status(401).json({ message : "Connexion impossible, token manquant", code: res.statusCode });
    }else{
      req.user = jwt.verify(token, process.env.JWT_SECRET);
      next();
    }
    ///
    // next();
    ///
  } catch (e) {
    res.status(400).json({error : 'Token invalide', code: res.statusCode});
  }
}

module.exports = auth;
```

```
const espController = require('../controllers/esp');
const express = require('express');
const router = express.Router();
const token = require('../auth');

//GET
router.get('/', espController.getAll);
router.get('/getEsp/:id', token, espController.getEspById);

//POST
router.post('/addEsp', token, espController.newEsp); // créer un nouvel utilisateur (voir le model pour le body)
router.post('/addEsp', token, espController.newEsp);

//DELETE
router.delete('/delete/:id', token, espController.deleteEsp);

//PUT
router.put('/put', token, espController.updateEsp);
```

Le Token est récupéré dans le retour de la connexion d'un utilisateur. Nous la stockons dans une variable de session afin de pouvoir l'utiliser dans le reste du projet.

Grâce à ce token récupéré , nous avons donc la possibilité d'utiliser toutes les routes de l'api sans avoir des erreurs d'authentification.

En effet, en plaçant dans le header de la requête le jeton jwt, nous pouvons accéder aux routes qui requièrent un token.

```

getMyEps: async function () { //Permet de get les esp de l'utilisateur
  this.showSpinner = true;
  await fetch(`${urlApi}/esp/getEsp/${this.userId}`, {
    headers: {
      "Content-Type": "application/json",
      "x-auth-token": this.$session.get("token"),
    },
  })
  .then((res) => {
    res.json().then((resJson) => {
      //on cache le spinner
      this.showSpinner = false;
      this.listEsp = resJson;
      //on recupere un code 200 mais on a pas d'esp, juste un objet "erreur" on remet donc la list vide
      if (this.listEsp.erreur) {
        this.listEsp = [];
      }
    });
    this.showSpinner = false;
  })
  .catch((err) => {
    console.error(err);
    //on cache le spinner si on arrive pas à récupérer les données pour ne pas gêner l'utilisateur
    this.showSpinner = false;
  });
},

```

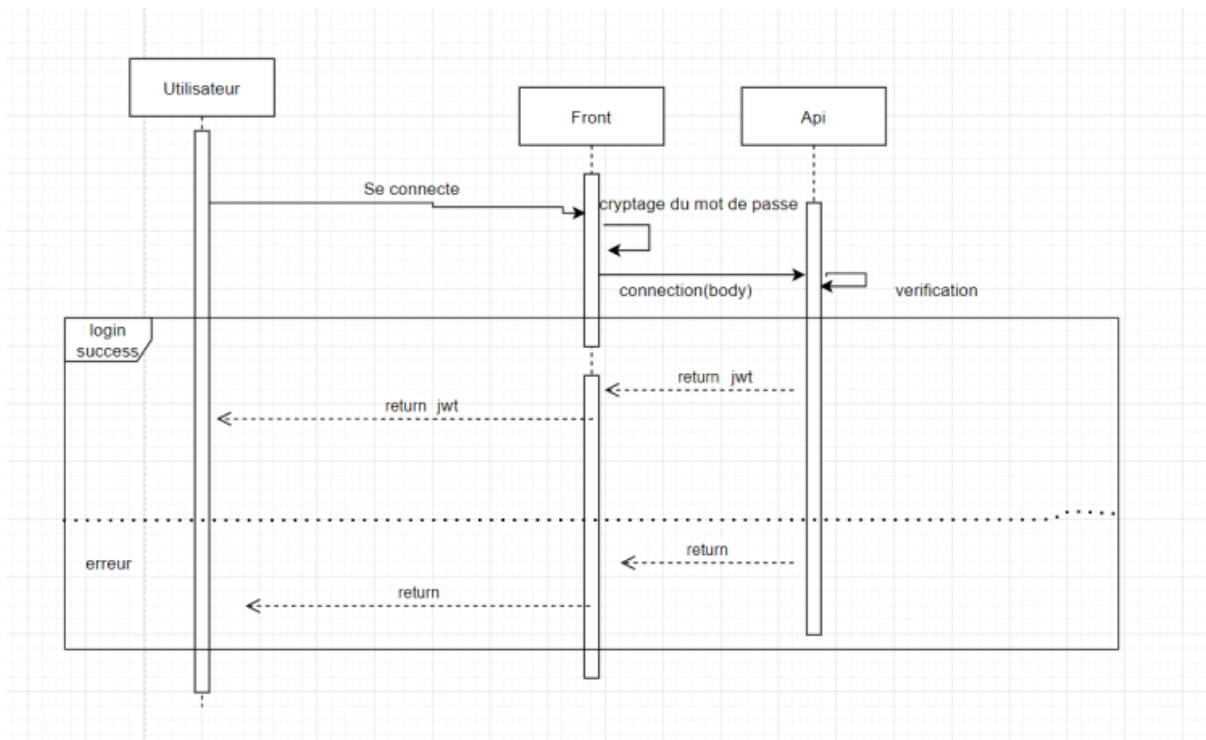
Sécurité de la Connexion:

Dans la partie connexion d'un utilisateur nous utilisons md5.

Un algorithme de chiffrement des messages MD5 est une fonction de hachage largement utilisée qui produit une valeur de hachage de 128 bits.

Cela nous permet de crypter le mot de passe de l'utilisateur lors de la création du compte et de ne garder que la version hasher de son mot de passe dans la base de donnée. Lors d'une connexion nous cryptons le mot de passe entrée et nous le comparons au hash présent en base de donnée.

L'algorithme cryptographique MD5 n'est pas réversible, c'est-à-dire que l'on ne peut pas décrypter une valeur de hachage créée par le MD5 pour ramener l'entrée à sa valeur initiale. Il n'y a donc aucun moyen de décrypter un mot de passe MD5.



En cryptographie, le sel est une chaîne aléatoire qu'on peut ajouter à un mot d'entrée, pour générer un hachage différent de celui du mot seul.

MD5 n'offre pas vraiment cette fonctionnalité dans l'algorithme cryptographique, mais on peut concaténer deux chaînes pour obtenir le même résultat

On peut soit utiliser une chaîne statique qu'on va utiliser à chaque fois ou en créer une de manière dynamique.

Cela fait partie des améliorations possibles sur ce sujet.

Sécurité du VPS:

Certaines sécurités dont je parle ici sont à retrouver dans le tutoriel de déploiement situé dans ce rapport nous verrons alors comment les mettre en place.

Pare-feu avec UFW et Fail2Ban

Un serveur VPS a besoin d'être sécurisé, hormis la sécurité de base SSH (RSA keys : <https://www.digitalocean.com/community/tutorials/how-to-create-ssh-keys-with-putty-to-connect-to-a-vps>)

dont je ne vous parlerai pas ici.

Je vais vous parler d'une sécurité classique qu'on oublie parfois, le pare-feu.

Le pare-feu ou firewall est essentiel, il permet la protection de la totalité du trafic réseau.

UFW (Uncomplicated Firewall) est un firewall simplifié mais aussi efficace que n'importe quel autre pare-feu. Par défaut, il refuse les connexions entrantes. Voici l'état de notre pare-feu actuel, ce qui est important ce sont les connexions (Allow) autorisée:

```
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip

To Action From
--
22/tcp DENY IN Anywhere
443/tcp ALLOW IN Anywhere
80/tcp ALLOW IN Anywhere
50 ALLOW IN Anywhere
50 /tcp ALLOW IN Anywhere
22 DENY IN Anywhere
1983 ALLOW IN Anywhere
3000 DENY IN Anywhere
8883 ALLOW IN Anywhere
1883 ALLOW IN Anywhere
8083 ALLOW IN Anywhere
18083 ALLOW IN Anywhere
3001 DENY IN Anywhere
22/tcp (v6) DENY IN Anywhere (v6)
443/tcp (v6) ALLOW IN Anywhere (v6)
80/tcp (v6) ALLOW IN Anywhere (v6)
50 (v6) ALLOW IN Anywhere (v6)
50 /tcp (v6) ALLOW IN Anywhere (v6)
22 (v6) DENY IN Anywhere (v6)
1983 (v6) ALLOW IN Anywhere (v6)
3000 (v6) DENY IN Anywhere (v6)
8883 (v6) ALLOW IN Anywhere (v6)
1883 (v6) ALLOW IN Anywhere (v6)
8083 (v6) ALLOW IN Anywhere (v6)
18083 (v6) ALLOW IN Anywhere (v6)
3001 (v6) DENY IN Anywhere (v6)
```

status UFW firewall

Ajouter en complément d'UFW le framework fail2ban pour bannir les IPS qui n'arrivent pas à se connecter et votre pare-feu est complètement opérationnel.

Cette configuration est importante car cela permet de sécuriser le reverse proxy utilisé avec le serveur Apache.

SSL/TLS

Avant de vous montrer en quoi le SSL était important voici un petit rappel sur le SSL et TLS: Le TLS (Transport Layer Security) est un protocole de sécurisation de la couche transport du modèle TCP/IP. Il a pour prédécesseur l'appellation SSL (Secure Sockets Layer). Par raccourci pour désigner TLS on emploie souvent le terme TLS. SSL/TLS permet un chiffrement des données lors de l'échange entre le client et le serveur.

Dans notre cas, nous avons mis à la disposition de notre serveur Apache un certificat TLS 1.3 généré par certbot un outil de génération de SSL/TLS qui utilise Let's Encrypt une société qui délivre des certificats SSL gratuits.

Notre certificat supporte les protocoles TLS 1.2 et 1.3 supporté par quasiment tous les navigateurs excepté :

- IE 11 sur Win Phone 8.1
- Safari 6 sur IOS 6.0.1
- Safari 7 sur IOS 7.1
- Safari 7 sur OS X 10.9
- Safari 8 sur IOS 8.4
- Safari 8 sur OS X 10.10

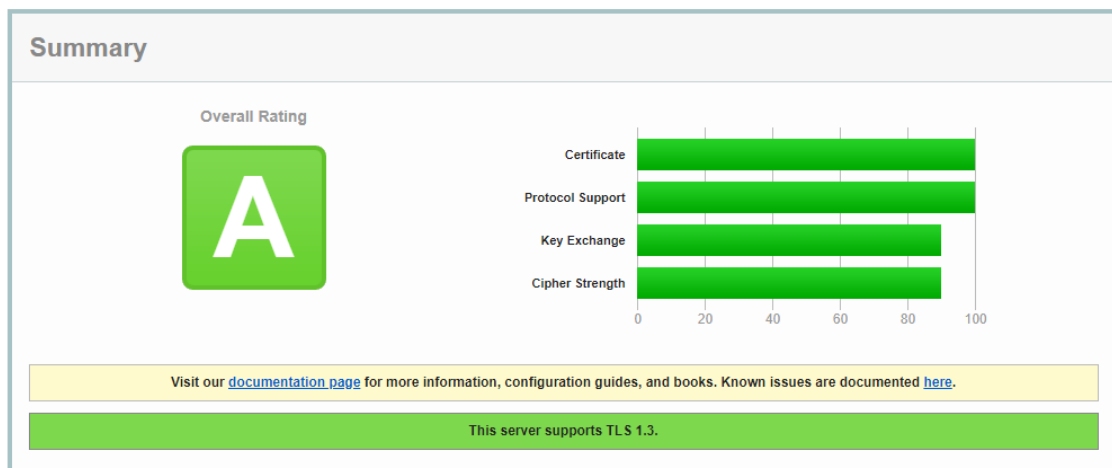
L'analyse complète est disponible ici:

<https://www.ssllabs.com/ssltest/analyze.html?d=valentinbordy.fr>

SSL Report: valentinbordy.fr (152.228.216.110)

Assessed on: Tue, 08 Jun 2021 10:37:53 UTC | [Hide](#) | [Clear cache](#)

[Scan Another »](#)



Reverse proxy

La sécurité du reverse proxy est faite par défaut grâce au pare-feu mais je pense qu'il est important d'en parler. Un reverse proxy est une passerelle entre internet et le réseau local du vps, cela veut dire qu'il redirige les requêtes demandées par l'utilisateur sur son réseau local. Par exemple pour accéder à une application node/express disponible sur le vps au port 3000 (notre cas) le client ne pourra pas passer directement par l'ip/nom du serveur comme suit: valentinbordy.fr:3000. Il devra passer par une route configurée sur le reverse proxy comme suit: valentinbordy.fr/meteo nous en parlerons plus amplement dans la partie déploiement du projet.

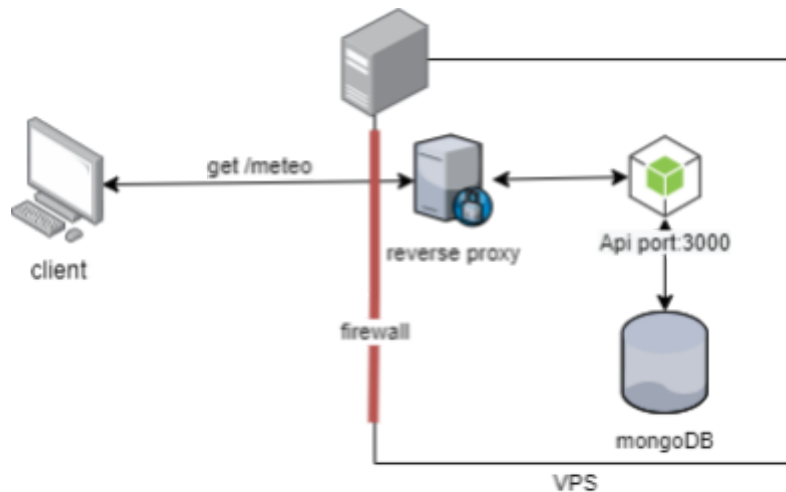


Schéma explication ReverseProxy

Au début du projet nous souhaitons passer par le port 3000 mais nous n'avons pas réussi à avoir des certificats SSL/TLS générés par let's Encrypt ce qui rendait l'api non sécurisé car nous n'avons pas de certificats. En cherchant, c'est là que nous avons trouvé la solution du reverse proxy qui est plus sécurisé que celle du port 3000.

Broker MQTT

Pour notre broker MQTT nous n'avons pas de certificats SSL/TLS car les données envoyées par l'esp ne sont pas des données privées. Néanmoins pour pouvoir envoyer des données sur le broker MQTT nous demandons un username et un mot de passe générique aux différents clients. Nous avons aussi désactivé les connexions anonymes au broker.

Mise en place et déploiement sur le VPS

Pour le déploiement nous ne sommes pas partis sur une solution "gratuite" que ce soit pour des raisons de sécurité avec des services en ligne où on ne peut avoir un contrôle total (heroku éteint le projet au bout de 30min, pas de TLS 1.3).

L'un de nous s'est proposé d'utiliser son VPS pour mettre en place ce projet et avoir un contrôle sur toute la chaîne du projet, que ce soit avec les ssl, le serveur apache, broker MQTT, mongo, etc..

Dans cette partie nous allons donc voir dans un premier temps comment installer son environnement sur le VPS c'est à dire installer le broker MQTT, la base mongoDB, le serveur apache avec en bonus le reverse proxy, le gestionnaire de processus pm2 avec node et comment avoir un certificat SSL/TLS pour son serveur apache avec certbot et let's encrypt.

Dans un second temps nous verrons comment déployer notre projet sur l'environnement que nous venons d'installer.

Pour rappel, voici l'architecture de notre projet. On peut voir que le vps possède toutes les parties de notre projet.

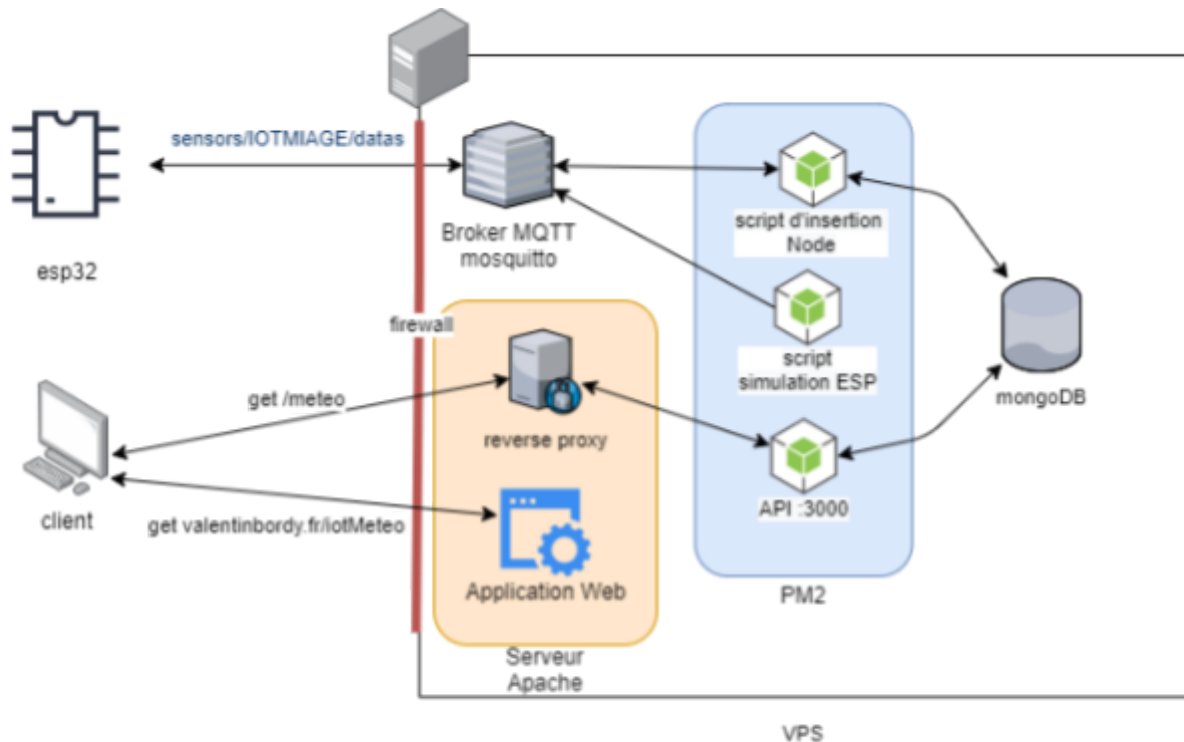


Schéma Architectural du projet

Mise en place du VPS

Comme vous vous en doutez la documentation et les tutoriels sur les vps Ubuntu sont abondants, je ne vais donc pas réinventer la roue. Ici je vous proposerai les tutoriels utilisés pour mettre en place un environnement d'exploitation. Mais avant de commencer les explications vous allez avoir besoin :

- d'un VPS Ubuntu 20.10, un autre OS ferait l'affaire mais les consignes qui vont suivre sont pour un serveur ubuntu.
- d'un nom de domaine relié au VPS pour le TLS/SSL.
- d'un minimum de connaissance Unix afin de pouvoir faciliter l'utilisation de son VPS.

Pour la suite nous partons du principe que votre VPS est actuellement à jour et sécurisé avec ufw (ou iptable) et fail2ban.

Pour sécuriser un VPS voici quelques liens utiles :

documentation ovh ssh et fail2ban:

<https://docs.ovh.com/fr/vps/conseils-securisation-vps/#configuration-du-pare-feu-interne-iptables>

documentation clef privée vps digital ocean:

<https://www.digitalocean.com/community/tutorials/how-to-create-ssh-keys-with-putty-to-connect-to-a-vps>

documentation ufw:

:
<https://www.digitalocean.com/community/tutorials/how-to-setup-a-firewall-with-ufw-on-an-ubuntu-and-debian-cloud-server>

Bien, commençons ! (N'oubliez pas de vous mettre avec un utilisateur ayant les privilèges sudo).

Apache (LAMP)

Dans un premier temps nous allons installer **Apache**, pour notre part nous avons installé directement la pile LAMP via le tutoriel de digital ocean écrit par Erika Heidi: <https://www.digitalocean.com/community/tutorials/how-to-install-linux-apache-mysql-php-lamp-stack-on-ubuntu-20-04-fr>

Nous reviendrons sur le reverse proxy et le certificat SSL/TLS plus tard.

Node et PM2

Nous avons besoin de node et PM2 pour gérer toute notre partie node avec l'API, le script d'écoute du broker et le script de simulation d'ESP.

Pour Node pareil il suffit de suivre le tutoriel suivant aussi disponible sur digital ocean écrit par Brian Boucheron:

<https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-20-04-fr>

Nous avons utilisé l'option 1 du tutoriel.

Node doit être en version 14 au minimum.

Une fois installé il faudra installer pm2 via la commande npm suivante :

```
npm install pm2@latest -g
```

nous reviendrons dans la seconde partie lors du déploiement sur comment utiliser pm2, en attendant voici la documentation officiel pour les curieux :

<https://pm2.keymetrics.io/docs/usage/quick-start/>

MongoDB

Pour mongoDB même principe que pour les précédents services il faut suivre le tutoriel de digital ocean mais attention cette fois-ci il ne faut surtout pas faire l'étape 4, elle consiste à autoriser le port de mongoDB (27017) sur le pare feu, ce que nous ne souhaitons surtout pas, car le seul moyen de modifier la base de donnée mongodb est en interne, via le shell, notre API ou via notre script d'insertion. L'article est écrit par Mateusz Papiernik:

<https://www.digitalocean.com/community/tutorials/how-to-install-mongodb-on-ubuntu-18-04>

Certificat SSL/TLS avec certbot

Cette partie n'est pas complexe mais on peut rencontrer plusieurs erreurs.

Certbot est un outil automatique de création de certificat SSL ou TLS.

Il va aussi modifier notre configuration apache pour le faire fonctionner en https:

<https://certbot.eff.org/lets-encrypt/ubuntufocal-apache>

Nous avons eu un problème lié au renouvellement de certificat, cette réponse a résolu notre soucis :

<https://community.letsencrypt.org/t/invalid-host-in-redirect-target-cant-find-the-error/132487/4>

Une fois installer vérifier que dans /etc/apache2/sites-available/default-ssl.conf le SSLEngine soit on comme ceci :

```
SSLEngine on
```

vérifiez aussi que vos fichiers de certificat soit bien initialisés comme ci-dessous avec à la place de <your_website> votre site web:

```
SSLCertificateFile /etc/letsencrypt/live/<your_website>/fullchain.pem
SSLCertificateKeyFile /etc/letsencrypt/live/<your_website>/privkey.pem
Include /etc/letsencrypt/options-ssl-apache.conf
```

Mise en place du reverse proxy

Un reverse proxy est important pour nous cela va nous permettre d'accéder à notre Api sans passer par le port de cette dernière (en l'occurrence 3000) et passer par notre serveur Apache.

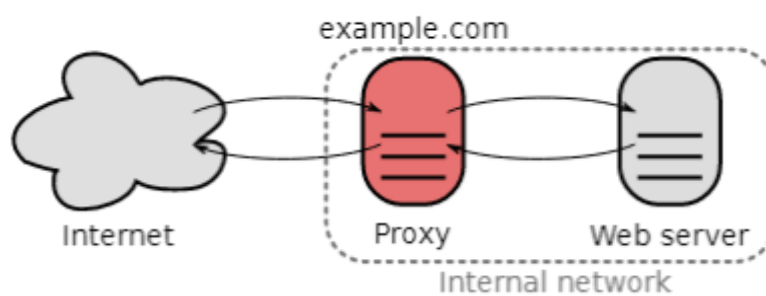


Schéma explication d'un reverse proxy

Avec Apache dans un premier temps il faut activer le proxy avec les 2 commandes suivantes à exécuter dans le terminal de votre vps :

```
sudo a2enmod proxy
sudo a2enmod proxy_http
```

n'oubliez de redémarrer le service apache avec :

```
sudo systemctl restart apache2
```

Dans le même fichier que pour la partie ssl soit /etc/apache2/sites-available/default-ssl.conf, ajouter à la fin avant </VirtualHost> la configuration suivante:

```
SSLProxyEngine on
SSLProxyVerify none
SSLProxyCheckPeerCN off
SSLProxyCheckPeerName off
SSLProxyCheckPeerExpire off

ProxyPass /meteo https://127.0.0.1:3000 retry=0 timeout=1600
Keepalive=On
ProxyPassReverse /meteo https://127.0.0.1:3000
```

Cela active le proxy sur notre configuration apache et désactive certaines vérifications du serveur distant (au niveau du SSL).

Pour rediriger vers notre api node on utilise ProxyPass et ProxyPassReverse.

Broker MQTT Mosquitto

Pour le broker MQTT nous avons aussi utilisé la documentation digital ocean écrit par Brian Boucheron et Hanif Jetha:

<https://www.digitalocean.com/community/tutorials/how-to-install-and-secure-the-mosquitto-mqtt-messaging-broker-on-ubuntu-18-04>

Néanmoins nous avons sauté la partie 3 liée au SSL, car nos données envoyées sur le broker ne sont pas privées et sont disponibles à tous.

L'étape 2 est importante et a ne pas sauter, elle permet de mettre en place une sécurité en supprimant les utilisateurs anonymes et en mettant en place un utilisateur / mot de passe.

Déploiement

Maintenant place au déploiement:

- l'application web via apache
- l'API node/express via pm2
- le script node d'écoute du broker MQTT via pm2
- le script de simulation via pm2

Avant de commencer le déploiement des différentes parties il vous faut:

- un accès FTP à votre serveur VPS
- nodeJS installé sur votre ordinateur
- le projet git en local sur votre ordinateur

L'application web

Si vous avez suivi correctement les tutoriels précédents notamment celui d'apache, vous devriez avoir un dossier `/var/www/<your_website>` connectez vous à votre vps via filezilla ou un autre logiciel de FTP (File Transfer Protocol). et rendez-vous à l'emplacement de ce dossier.

En parallèle nous allons générer le build final de l'application web vueJS.

Avant de générer ce build nous allons vérifier la configuration de l'url utilisé pour accéder à l'API pour cela se rendre dans `/VraiFront/src/components/ConfApi.js` et vérifier que la variable uri correspond à ce que vous avez mis dans ProxyPassReverse, en l'occurrence `/meteo` ce qui donne comme uri ceci:

```
const uriApi = 'https://<your_website>.fr/meteo';
```

une fois cela fait ouvrez un cmd dans `/VraiFront` et exécuter la commande suivante :

```
npm i
```

puis lancer la commande suivante:

```
npm run build
```

cela devrait vous générer un dossier `/dist`. ce dossier contient ceci:

```
dist/  
-css/*  
-img/*  
-js/*  
-favicon.ico  
-index.html
```

Maintenant nous allons transférer les fichiers via votre FTP.

Dans Filezilla ou votre FTP rendez-vous là où vous souhaitez mettre votre application web. En l'occurrence nous cela sera : `/var/www/<your_website>/iotMeteo`

Si vous n'avez pas changé la configuration de base de votre serveur apache vous aurez via l'url `https://<your_website>/iotMeteo` votre projet pour nous cela est :

```
🔒 valentinbordy.fr/iotMeteo/#/login
```

Si vous n'avez pas de résultat et que votre serveur apache est bien lancé cela est sûrement dû à la route configuré dans le projet vueJS pour vérifier cela rendez-vous dans le fichier `vue.config.js` et vérifiez que vous avez bien la configuration suivante:

```
module.exports = {  
  "transpileDependencies": [  
    vue.config.resolveModuleName.resolveModuleName('vue')  ]  
}
```

```
"vuetify"
],
publicPath: '/iotMeteo/'
}
```

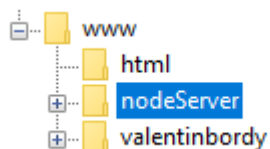
La propriété `publicPath` doit correspondre à la route apache que vous avez choisie pour votre projet toujours dans notre exemple `/iotMeteo`.

L'API node/express avec pm2

Ici nous n'avons pas de commande à lancer sur l'ordinateur, seulement des vérifications de configurations.

Mais avant cela nous allons nous rendre sur notre VPS pour préparer l'espace pour l'API.

Personnellement nous avons préparé l'espace dans le répertoire `www` du serveur apache mais cela peut être fait n'importe où.



L'espace est un dossier comprenant d'autres dossiers listant les différents processus node utilisé par le VPS :

- ApiMeteo
- ApiRestaurant
- mqttToMongolOT
- mqttToMongolOTDEV
- scriptSimulationData

Intéressons nous au dossier `ApiMeteo`, dans ce dossier nous allons d'abord générer un certificat `ssl` pour l'application `express` via `openssl`. Cette partie a peut être sauté car notre proxy ignore ce certificat mais ne pouvant plus toucher au code nous allons générer ce certificat pour n'avoir aucune erreur lors du lancement du processus. Si vous décidez de sauter ce processus n'oubliez pas d'enlever le bout de code que nous verrons plus tard qui est lié aux certificats `SSL`. Pour se faire se rendre dans le dossier `ApiMeteo` sur le `vps` et exécuter la commande suivante:

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
./key.key -out key.crt
```

cela devrait vous générer 2 fichiers dans le dossier `ApiMeteo` :

- `cert.pem`
- `key.pem`

Maintenant vérifions les configurations de notre Api.

Dans un premier temps, vérifions que la dbconfig est bonne. Rendez-vous dans ApiMeteo/config/dbconfig.js et vérifiez que la variable uri soit bien celle de votre base de données local du vps soit :

```
const uri = 'mongodb://localhost:27017/METEO';
```

Il faut vérifier en fonction de l'étape du SSL que la config dans le fichier serv2.js soit la bonne :

```
const options = {  
  key: fs.readFileSync('./key.pem'),  
  cert: fs.readFileSync('./cert.pem'),  
  passphrase: '<votre_mdp_cert>'  
};  
https.createServer(options, app).listen(port);
```

(Si vous n'avez pas fait l'étape précédente du certificat SSL vous pouvez enlever la const options sans oublier de l'enlever dans https.createServer).

Une fois ces vérifications faites, vous devrez récupérer les fichiers présent dans le dossier /ApiMeteo à l'exception du dossier ./nodes_modules, de package-lock.json et du .gitignore.

Via Filezilla, déplacez ces fichiers dans le dossier que vous avez préparé précédemment.

Rendez-vous dans ce dossier via votre VPS et exécuter la commande suivante :

```
npm i
```

Bien, maintenant que votre application est prête vous pouvez la lancer via pm2 !
Pour ce faire il suffit juste d'exécuter la commande pm2 suivante :

```
pm2 start serv2.js
```

Voilà votre API est lancée !
si vous souhaitez voir votre les logs il suffit de faire

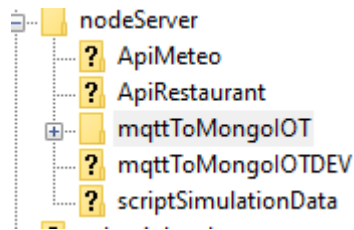
```
pm2 logs serv2
```

la liste des commandes pm2 disponibles est trouvable ici:

<https://pm2.keymetrics.io/docs/usage/quick-start/>

Le script node d'écoute du broker MQTT

Pour déployer le script il faut préparer un dossier pour l'accueillir pour se faire nous avons préparé un dossier nommé mqttToMongolOT dans notre serveur VPS.



Via votre client FTP transférez les fichiers node_iot_prod.js et package.json du dossier scriptMQTTMONGO dans le dossier préparez.

Rendez-vous dans ce dossier via votre VPS et exécutez la commande suivante :

```
npm i
```

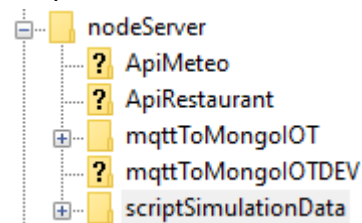
Maintenant nous pouvons lancer ce script via pm2:

```
pm2 start node_iot_prod.js
```

Le script de simulation d'esp

Même mécanisme que pour le script d'écoute sauf que cette fois nous n'avons pas de package.json.

Dans ce cas préparez un dossier pour logger cette application, pour nous il s'appellera scriptSimulationData :



Transférez le fichier scriptSimulationData.js dans le dossier.

Maintenant nous allons installer les nodes_modules à la main. Sur votre VPS rendez-vous dans le dossier scriptSimulationData et faites la commande suivante:

```
npm i mqtt
```

Maintenant nous pouvons lancer ce script via pm2:

```
pm2 start scriptSimulationData.js
```

Améliorations futures pour le déploiement

Pour les améliorations futures il serait un plus de préparer des fichiers ENV pour stocker les différentes configurations (prod et dev). Un script permettant de déployer depuis un

répertoire github serait un gain de temps pour les développeurs qui pourront le faire eux même ou bien pour le devops.

Une autre amélioration serait de, via l'API express, afficher sur la route "/" l'application vueJS.

Pour cela il suffit de déplacer les fichiers de l'application web (dossier dist) dans le répertoire de l'API, puis au lieu de rendre "Hellow world" via le get "/" la page html on devrait renvoyer le fichier html du dossier dist comme suit:

```
app.get('/', (req, res) => res.sendFile(path.join(__dirname +  
'/index.html')));
```

Dans ce cas, notre application sera rendue à l'adresse valentinbordy.fr/meteo au lieu de valentinbordy.fr/iotMeteo, mais dans ce cas il faudra faire attention au publicPath que vous avez renseigné lors du build de l'application VueJS.

Bonus: Installation et lancement du projet en local

Si vous souhaitez faire des modifications au projet actuel il est fortement conseillé de l'installer en local, mais comme vous vous en doutez nous n'allons pas reproduire l'environnement de production (VPS) sur votre machine, nous allons passer par des outils similaires mais externe.

Dans cette partie nous allons donc voir comment faire pour lancer l'API, l'application web et transférer le code à son esp.

Nous conseillons pour n'avoir aucun problème de posséder sur son poste de travail une version de nodeJs supérieure à la 14.x.

Api nodeJS

L'installation et le lancement de l'API se fait facilement avec node. Une fois dans le dossier du projet, rendez-vous dans le dossier de l'api et lancez la commande suivante:

```
npm i
```

puis :

```
npm run dev
```

Cette commande va permettre de lancer notre projet avec nodemon un must have dans le développement d'application node, permettant de redémarrer automatiquement le serveur à chaque changement de code.

L'application devrait démarrer sur votre port 3000 s'il n'est pas déjà occupé, si c'est le cas vous pouvez changer le port dans le fichier serv2.js.

Il faut maintenant vérifier l'adresse mongo dans le fichier dbconfig.js de l'api. Par défaut le fichier contient l'adresse localhost de la base de données située sur le VPS, ici vous pouvez

le changer par une base de données mongo Atlas qui servira de base de développement. Pour le développement en local il faudra aussi changer le fichier serv2.js en aillant bien :

```
//const options = {  
//    key: fs.readFileSync('./key.pem'),  
//    cert: fs.readFileSync('./cert.pem'),  
//    passphrase: '<passphrase_de_votre_certssl>'  
// };  
//https.createServer(options, app).listen(port);  
app.listen(port, () => console.log(`http://localhost:${port}/`));
```

Application Web VueJS

l'application VueJS fonctionne de la même façon que l'Api nodeJS, pour la démarrer il faut faire les commandes suivantes :

```
npm i
```

puis :

```
npm run serve
```

Cela devrait vous lancer une application web sur le port 8000, si ce n'est pas le cas pas de soucis VueJS mettra un port non utilisé par votre machine, pour ma part c'est le 8080.

Script d'insertion/écoute du broker MQTT et script de simulation

Vous pouvez si vous le souhaitez lancer le script d'insertion sur votre poste, pour cela il faudra utiliser pm2, vous pouvez pour l'installer utiliser la même commande que sur le vps soit :

```
npm install pm2@latest -g
```

Ensuite il faut installer les nodes_modules donc on ne perd pas la main:

```
npm i
```

Une fois fait, vous devrez changer la valeur de l'url MQTT, je conseille d'utiliser <http://broker.hivemq.com> un broker public, dans ce cas, les options utilisateurs et mot de passe ne sont plus utiles.

Il faudra aussi changer l'adresse mongoDB pour correspondre à celle située dans le fichier dbconfig.js de l'API (url que vous avez normalement changé plus haut).

Une fois cela fait vous pouvez démarrer votre script via la commande suivante:

```
pm2 start serv2.js
```

Pour le script de simulation il faut seulement modifier l'url du broker MQTT et enlever les options comme fait au-dessus pour le précédent script, il faudra donc mettre la même url.

une fois fait vu que la dépendance 'mqtt' a été installé avec le npm i précédent nous pouvons directement lancer le script :

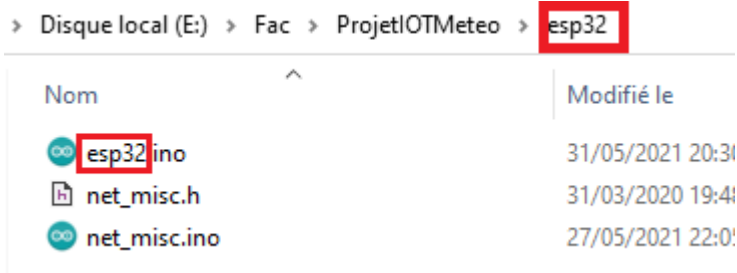
```
pm2 start scriptSimulationData.js
```

Par défaut, les valeurs seront envoyées toutes les 10 minutes, néanmoins vous pouvez changer cette valeur si vous le souhaitez.

Environnement de travail pour l'ESP32

Pour développer sur votre ESP32 il vous faut l'IDE d'Arduino disponible ici : <https://www.arduino.cc/en/software>

Ouvrez le projet dans l'environnement, attention il faut que le nom du dossier contenant le projet ESP32 corresponde au nom du fichier en .ino



Voici toutes les librairies à installer pour le projet :

- Adafruit BMB280
- Adafruit Circuit Playground
- Adafruit Unified Sensor
- AdafruitJson
- AsyncHTTPRequest_Generic
- DallasTemperature
- DHT sensor library
- ESPAsync_WiFiManager
- EspMQTTClient
- OneWire
- PathVariableHandlers
- PubSubClient

Suivez les explications de ce lien <https://github.com/espressif/arduino-esp32> pour ajouter la carte ESP32 a votre environnement

Pour l'utilisation générale de l'IDE : <https://www.arduino.cc/en/Guide/Environment>

Il faudra aussi que l'adresse du broker MQTT corresponde à celle que vous avez précisé plus haut dans le script d'écoute et de simulation.

Amélioration

Nous avons pensé notre projet pour qu'il soit facilement modulable ainsi les améliorations possibles seront facile à mettre en place.

Nous avons visualisé comme point d'amélioration les points suivants:

- Lors d'un changement d'adresse ne plus afficher les données ultérieures de l'ESP.
- Améliorer la précision des prévisions.
- Modification du profil (exemple image, nom, adresse).
- Amélioration de la liaison ESP-USER.
- Amélioration du déploiement comme dit plus haut.
- Amélioration du cryptage de données avec le sel dynamique