P2025 : Smart Territories

Théo Bourgoin

Dossier technique du projet - partie individuelle

Table des matières

Situation dans le projet

1.1 - Synoptique de la réalisation

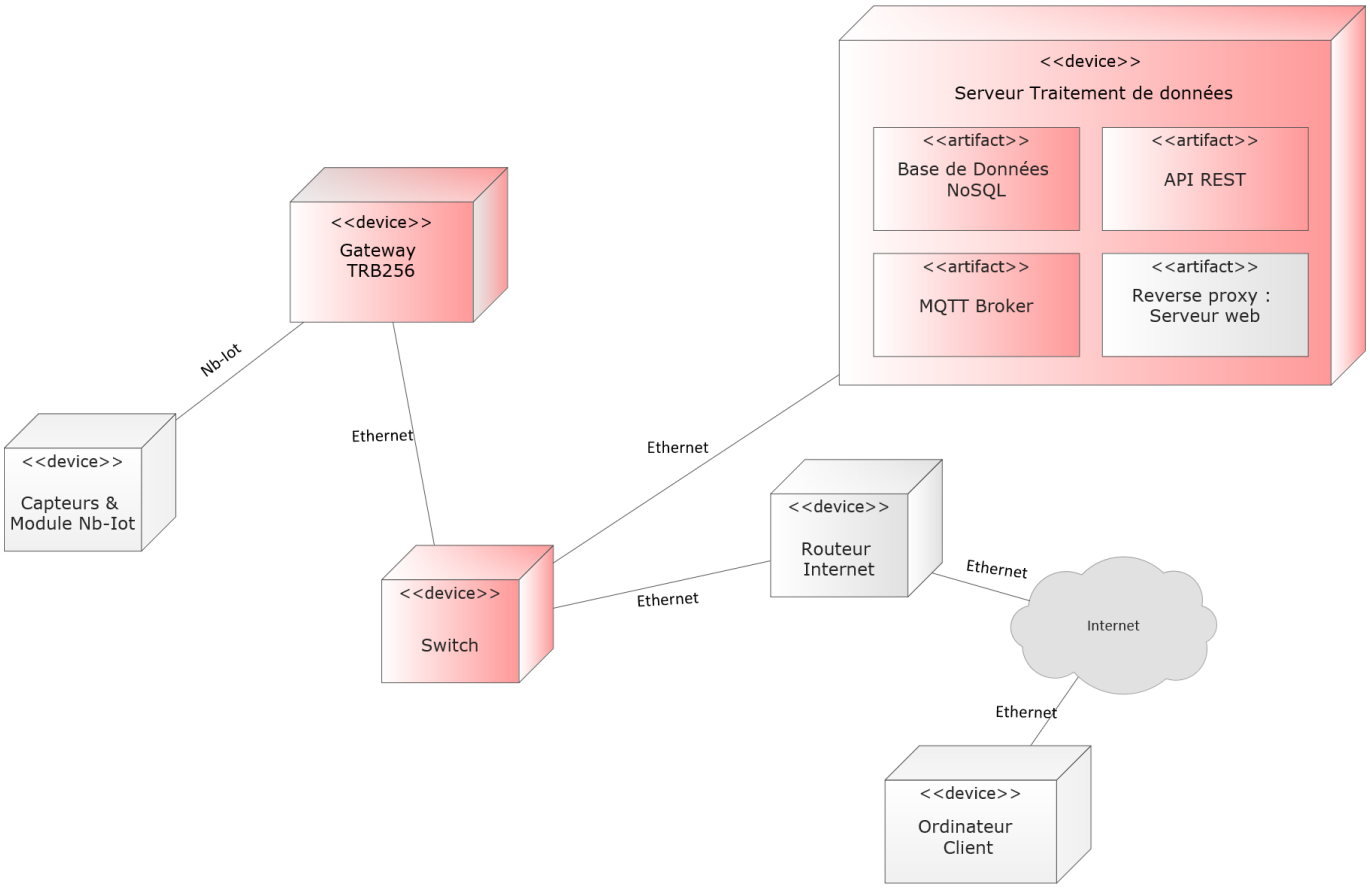


Diagramme de déploiement du projet, en rouge la partie traitée

Le projet Smart Territories demande une analyse de données venant de capteurs, envoyés à une passerelle jusqu'à un réseau d'analyse des données.

Il s'agit donc d'un transfert de données en longue distance et demandant une faible consommation d'énergie. Ainsi les données seront récoltées sur une plus longue période.

Le projet nécessitant une surveillance des données à long terme, il nous faut donc une base de données capable d'accueillir des données pendant plusieurs années.

De plus, une passerelle sera certainement nécessaire pour la réception des données, cette solution dépendra des besoins des ER, ainsi que leur moyen de communication IoT.

Pour ce qui est de l'affichage des données, il nous faudra une application et donc un serveur dédié. Ces mêmes données seront précédemment traitées et récupérées par un autre serveur. Il nous faudra donc un lien entre ces deux appareils pour pouvoir transmettre les données.

Ainsi, pour ma partie du travail, il me faut établir les solutions de : Passerelle, Serveur d'application soit Base de Données et API pour serveur web.

Le travail de réalisation de l'API sera fait sur une Machine Virtuelle avant d'être implémenté sur le serveur. Ainsi, je pourrais créer un manuel d'utilisation ainsi qu'avancer au fil des problèmes et différentes erreurs reportées. Cette même machine virtuelle me servira de machine de test. Ainsi les tests unitaires seront effectués sur la machine.

1.2 - Description de la partie personnelle

Ma partie consiste de la liaison des données d'un coté à l'autre de l'infrastructure. Avant la réception du matériel, l'important est de créer l'API avec une base de données de test.

Comme dis précédemment, cette partie sera réalisée sur machine virtuelle pour établir un manuel d'implémentation complet et ainsi repérer les problèmes principaux à cette implémentation.

Après avoir reçu le matériel commandé, la partie de liaison et réception des données capteurs sera implémentée directement sur le matériel.

Ensuite l'envoi des données directement dans la base prévue à cet effet sera réalisée.

Ainsi, la priorité est le cas d'utilisation « consulter les données ».

2 - Réalisation du cas d'utilisation « Consulter les données »

Communiquer avec le serveur web est la principale fonction qui m'est attribuée dans ce cas d'utilisation, pour cela une API (Application Programming Interface) semble idéale. Pour cela, il existe plusieurs types d'API (Rest, Soap, …). Dans le cadre du projet, l'API devra proposer des données en fonction des éléments demandés par l'application (ou indirectement de l'utilisateur qui y est connecté).
Cependant il existe plusieurs architectures d'API, il faut donc choisir la bonne :

REST	Une API REST peut permettre à son utilisateur de lire, mettre à jour voir rajouter des données dans une base. Cela se fait par le biais d'actions ou requête HTTP. On a donc accès aux méthodes de requête HTTP : GET, PUT voir même POST.
RPC	Une API RPC permet aux développeurs d'appeler des fonctions distantes sur des serveurs externes comme si elles étaient propres à leur logiciel.
SOAP	SOAP (Simple Object Access Protocol), c'est un protocole standard pour l'échange de messages XML. Ce langage a été Initialement conçu pour que des applications développées avec différents langages sur différentes plateformes puissent communiquer. Ce protocole est assez ancien, mais lisible par tous les langages.

Il nous faudra donc mettre à disposition les données en fonction des demandes, sous la forme d'échanges. Ainsi une API REST semble la bonne solution.

Cette API récupèrera les données de la base en fonction des demandes de l'utilisateur (exemple : données de pollution pour « tel jour » à « tel endroit »). La base de données sera donc interrogée à l'aide de la requête. Les informations nécessaires seront donc renvoyées et affichées sur l'application.

Cependant, pour communiquer les données, il faut qu'elles soient stockées dans une base.

En respectant les contraintes citées dans l'étude préliminaire, il nous faut une Base de Données NoSQL, étant donné qu'il n'existera pas de lien concret entre les données. En effet, nous réceptionnerons les données venant des capteurs à un certain temps, ainsi qu'un champ pour les capteurs eux-mêmes ainsi que leur localisation. On pourra ainsi identifier les données à l'aide d'un temps et d'un capteur, et un capteur à l'aide d'une localisation.
Pour cela, deux Bases de Données NoSQL semblent être un bon choix :

Mongo DB	MongoDB est une base de données gratuite en auto-hébergèment, qui a cependant une limite de stockage de données (512 Mo en 100% gratuit) Cette base stocke les données en JSON et est facile d'utilisation. De plus elle est intégrable avec plusieurs langages de programmation.
Cassandra	Cassandra est une base de données créée par Apache, 100% gratuite et qui accepte un plus grand volume de données. Cependant elle nécessite une bonne modélisation des données pour une meilleure compréhension et surtout pour éviter les problèmes de données.

Le choix final de la base de données est MongoDB, et donc celle qui a été incluse dans l'implémentation.

2.1 - Conception détaillée

Pour ce qui est de l'installation de la base de données MongoDB, elle sera faite sur l'OS qui sera lui aussi présent sur le serveur : Ubuntu 20.04 (Focal) en raison de problèmes d'installation (voir 2.2.2).
La version de MongoDB choisie est la 4.4.

Ainsi, pour installer MongoDB, il faut suivre les commandes suivantes

Récupérer la clé GPG de MongoDB correspondant à la version	curl -fsSL https://www.mongodb.org/static/pgp/server-4.4.asc | \sudo gpg -o /usr/share/keyrings/mongodb-server-4.4.gpg \ --dearmor (Installer curl et gnupg si besoin)
Créer la liste de fichiers	Echo "deb [arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-server-4.4.gpg] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/4.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list
Recharger les paquets de la base de données	sudo apt-get update
Télécharger MongoDB à la version nécessaire	sudo apt-get install -y mongodb-org=4.4.29 mongodb-org-server=4.4.29 mongodb-org-shell=4.4.29 mongodb-org-mongos=4.4.29 mongodb-org-tools=4.4.29
Après cela il ne reste plus qu'à vérifier le bon démarrage de la BDD	sudo systemctl start mongod
Puis vérifier son status	sudo systemctl status mongod

Après recherches, j'ai retenu les commandes suivantes pour créer une base de données en CLI avec le shell MongoDB :

Entrée dans le shell MongoDB	mongo
Utiliser une base de données ou la créer si ce n'est pas déjà le cas	use « nomBDD »
Vérification de la BDD en utilisation	db
Inserer une entité dans la BDD	db."nom_bdd".insert({ "Nom_Champ1" : "Donnée1", "Nom_Champ2" : "Donnée2", [...], "Nom_ChampN" : "DonnéeN" })
Verifier les données dans la BDD	db."nom_BDD".find().forEach(printjson)

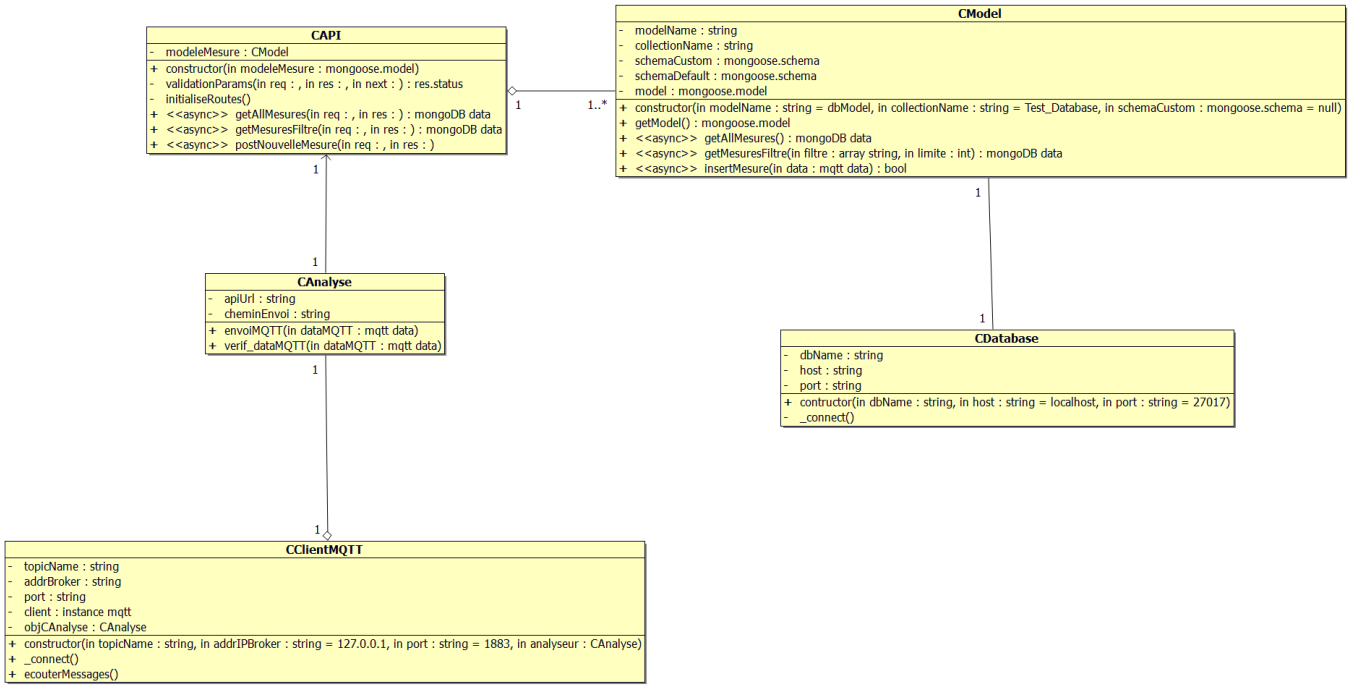
Après installation de la Base ainsi que la création de base de données de test, il faut installer les logiciels utiles pour l'API. Le langage de programmation utilisé pour le code sera le JavaScript (JS) à l'aide d'Express. En effet JavaScript avec Express est léger et rapide. De plus, beaucoup d'aides sont disponibles sur le web, ce qui sera utile pour découvrir un langage de programmation que je ne connais pas.
Enfin, il m'est impératif de coder avec des classes, soit en Programmation Orientée Objet.

Avant de coder l'API, il faut installer tous les modules nécessaires au codage en JavaScript à l'aide d'express :
Il est aussi nécessaire de mettre à jour Node JS pour utiliser Mongoose, la bibliothèque qui nous servira de lien pour notre base de données MongoDB.

Installer NPM	sudo apt install nodejs npm -y
Installer l'outil de dev NPM	sudo apt install build-essential -y
Ajouter un répertoire pour l'API	mkdir "chemin et nom de repertoire voulu"
Créer un fichier .js pour le code de l'API Nommé entry.js pour le point d'entrée vers notre code. Le nom usuel est « index.js »	nano entry.js
Démarrer le projet Node JS (Répondre aux différentes questions comme le nom du projet et l'auteur, le point d'entrée, etc)	npm init
Un paquet JSON sera ajouté dans le répertoire créé précédemment (package.json) Si besoin de vérifier le paquet et les informations renseignées	nano package.json
Ajouter express à l'API	npm install express

Afin d'installer le gestionnaire de MongoDB pour NodeJS, il faut le mettre à jour à sa dernière version (ou antérieure tant que mongoose est supporté)	curl -fsSL https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.4/install.sh | bash nvm install 18
Lier mongoose à l'API	npm install mongoose

Après avoir installé tous les éléments nécessaires à la création de l'API REST liée avec MongoDB, il ne nous reste plus qu'à la coder. Pour cela il nous faudra écrire dans le fichier entry.js précédemment lié à l'API (voir « npm init »)
L'API réalisée sur la Machine Virtuelle sera accessible en localhost sur le port 8080.
Étant en Programmation Orientée Objet, le code sera réparti en plusieurs classes pour garantir un code plus optimal à différentes situations.



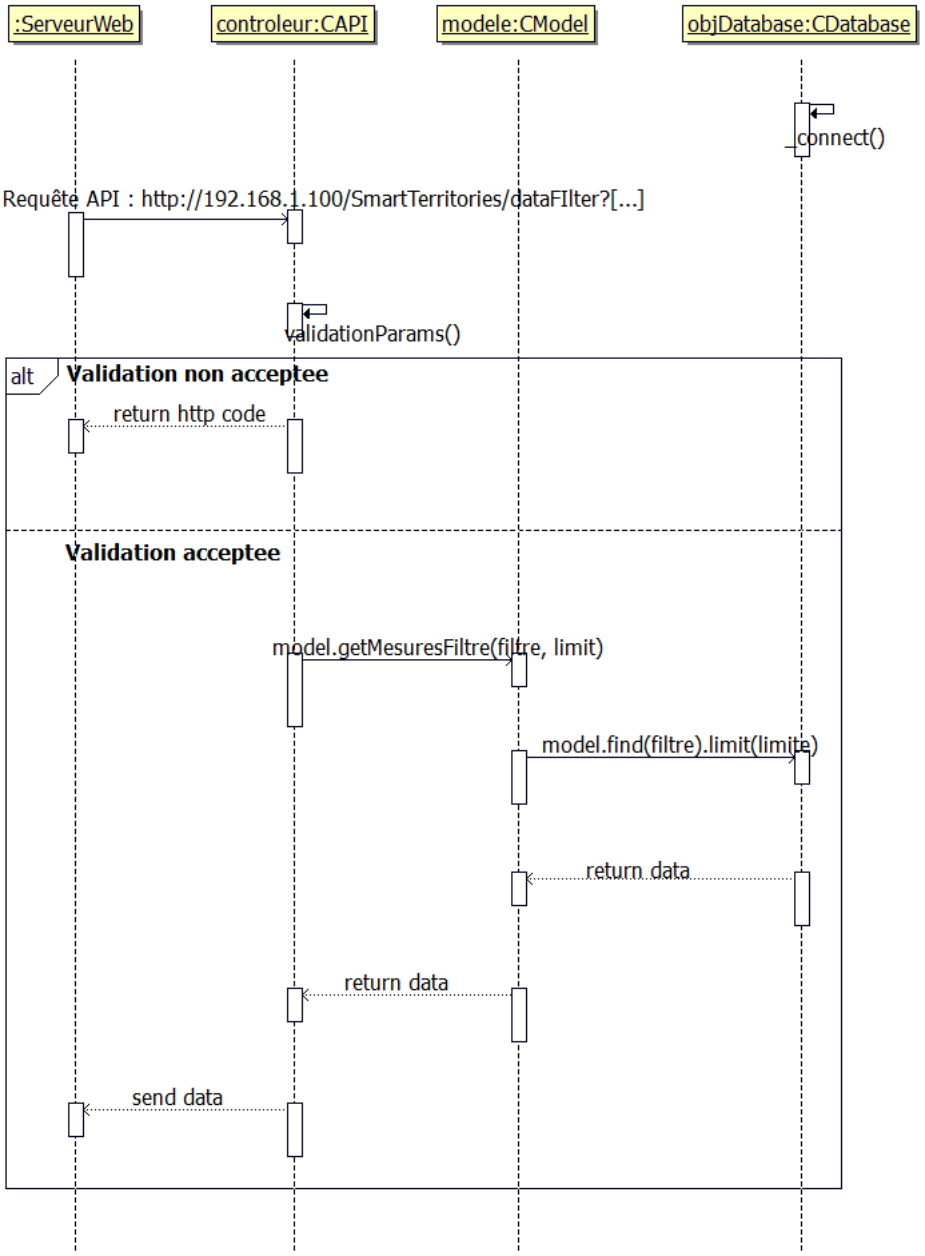
Ce diagramme correspond au code entier : la partie du code de l'incrément suivant a été incluse (CAnalyse et CClientMQTT)
On aura donc notre classe CClientMQTT, étant notre classe frontière : elle réceptionnera les données MQTT pour ensuite les analyser et les stocker dans la base de données (voir incrément UC Archiver les données pour plus de détails)
La classe CAPI regroupe toutes les requêtes possibles de l'API (POST pour stocker les données, GET avec un filtre et une limite pour avoir les données voulues, GET simple pour avoir toutes les données).
Les classes CAnalyse et CAPI sont donc nos classes de contrôle.
La classe CModel sert d'intermédiaire entre l'API et la BDD. En effet on utilise on modèle mongoose pour récupérer et envoyer les données vers la base.
Enfin, la classe CDatabase sert de connexion à une base de données du choix de l'utilisateur. Elle établit la connexion à cette base.
Les classes CModel et CDatabase sont donc nos classes entités.

Par respect du nombre de page l'entièreté du code ne sera pas copiée, les parties correspondant aux tests seront étudiées. Le code entier se trouve sur le github du groupe, ainsi que les différentes versions du code.

Les parties du code détaillées seront les suivantes :

- Requête de filtre de la classe CAPI
- Méthode de Validation des paramètres de CAPI

Ainsi, lors d'une requête venant du serveur web, l'API est interrogée. Celle-ci vérifiera les critères présents dans la requête pour ensuite aller (ou non) chercher les données dans la Base de Données. Notre classe CModel servira de relai entre l'API et la Base de Données.



Requête avec filtre pour les données (classe CAPI)

On renvoie une alerte avant de retourner toutes les données & renvoie à une autre requête

On instancie la limite de données modifier à celle entrée dans la requête, sinon à 10 par défaut.
Puis on vérifie s'il est positif.

Si un paramètre autorisé se trouve dans la requête alors on prépare le filtre pour renvoyer les données nécessaires

On peut filtrer sur une plage de date ou une date simple, cependant si les deux sont présents alors la date simple est ignorée

```
    async getMesuresFiltre(req, res) {
        try {
            logger.info(`Requête reçue: GET /dataFilter avec paramètres: ${JSON.stringify(req.query)}`);
//          On vérifie que l'élément "req.query" n'est pas vide pour ne pas surcharger le serveur par l'envoi de données
            if (Object.keys(req.query).length === 0) {
                logger.warn("Tentative de requête sans filtre. Utilisation de /allData recommandée.\n");
                return res.status(303).json({ message: "ATTENTION : vous êtes sur le point d'afficher toutes les données et la base, utiliser la requête : '/SmartTerritories/allData' si c'est votre demande." });
            }

            const { Capteur, TypeDeDonnee, Date, DateDebut, DateFin, Limite } = req.query;
            const filtre = {};

//      On créé une variable comportant tous les paramètres possibles dans la remarque pour éviter une erreur de syntaxe ou de paramètre inutile
            const params_Requete = ["Capteur", "TypeDeDonnee", "Date", "DateDebut", "DateFin", "Limite"];


//          On gère le paramètre de limite de données (limite de renvoi maximum)
//          Cette partie du code est gérée ici et non dans "validationParams" pour avoir la variable "limit" en local et ne pas a devoir la placer en global
            const limit = Limite || 10;
            if (isNaN(limit) || limit <= 0) {
                logger.warn(`Limite invalide : ${Limite}\n`);
                return res.status(400).json({ message: "Le paramètre 'Limite' doit être un nombre positif." });
            }

            if (Capteur) {
                //console.log("Filtre par capteur: ", Capteur);
                filtre.Capteur = Capteur;
            }

            if (TypeDeDonnee) {
                //console.log("Filtre par TypeDeDonnee: ", TypeDeDonnee);
                filtre.TypeDeDonnee = TypeDeDonnee;
            }

            if (DateDebut || DateFin) {
                filtre.Date = {};

                if (DateDebut) {
            filtre.Date.$gte = DateDebut;
                }
                if (DateFin) {
            filtre.Date.$lte = DateFin;
                }
```

	//console.log("Filtrage par plage de dates : ", filtre.Date); } else if (Date){ //console.log("Filtre par Date: ", Date); filtre.Date = Date; }
On envoie la requête dans la classe CModel, qui gère le retour des données.	const retour = await this.model.getMesuresFiltre(filtre, limit); //console.log("Retour des données: ", retour); if (retour.length === 0) { logger.warn("Aucune donnée trouvée avec les critères fournis.\n"); return res.status(444).json({ message: "Aucune donnée trouvée avec ces critères." }); }
Si une erreur à été rencontrée alors on la retourne.	logger.info(`Données renvoyées : ${retour.length}\n`); res.status(200).json(retour); } catch (error) { logger.warn(`Erreur dans getMesuresFiltre : ${error.message}\n`); //console.log("Retour d'erreur: ", error); res.status(500).json({ message: "Erreur interne du serveur.", error }); } }
Méthode de Validation des paramètres (classe CAPI)	validationParams(req, res, next) { logger.info(`Requête reçue: GET /dataFilter avec paramètres: ${JSON.stringify(req.query)}`); const paramsAutorises = ["Capteur", "TypeDeDonnee", "Date", "DateDebut", "DateFin", "Limite"]; const paramsRecus = Object.keys(req.query);
On compare les paramètres possibles dans la requête avec ceux présents.	if (paramsRecus.find(param => !paramsAutorises.includes(param))) { logger.warn(`Paramètre(s) invalide(s) reçu(s) : ${paramsRecus}\n`); return res.status(400).json({ message: "Paramètres invalides détectés." }); }
Si une date simple avec une date de fin ou de début alors on renvoi un message de prévention, mais pas les données	// On vérifie qu'il n'existe pas ET une Date simple ainsi qu'une date de début ou fin if (req.query.Date && (req.query.DateDebut || req.query.DateFin)) { logger.warn("Date ignorée car DateDebut ou DateFin également fourni.\n"); return res.status(206).json({ message: "La date (simple) sera ignorée car vous avez entrée une date de début et/ou de fin" }); }
On vérifie le format de la date si elle est présente, si il en a plusieurs on les vérifies une par une.	// On vérifie le format de la date (si fournie) if (req.query.Date) { const regexDate = new RegExp("^\\d{4}-\\d{2}-\\d{2} ((0\\d)|(1\\d)|(2[0-3]))$"); if (Array.isArray(req.query.Date)) { logger.warn(`Date invalide reçue : ${req.query.Date}\n`); for (const d of req.query.Date) { if (!regexDate.test(d)) {

Même chose pour date de début en vérifiant qu'il n'y en existe pas plusieurs dans la requête

Même chose pour la date de fin.

```
                return res.status(400).json({ message: "Format de date invalide. Utiliser YYYY-MM-DD hh." });
            }
        }
        } else {
//          Vérification d'une seule date
        if (!regexDate.test(req.query.Date)) {
            logger.warn(`Date invalide reçue : ${req.query.Date}\n`);
            return res.status(400).json({ message: "Format de date invalide. Utiliser YYYY-MM-DD hh." });
        }
        }
    }

//  On vérifie le format et le nombre de DateDebut dans la requête
    if (req.query.DateDebut) {
        const regexDate = new RegExp("^\\d{4}-\\d{2}-\\d{2} ((0\\d)|(1\\d)|(2[0-3]))$");


        if (Array.isArray(req.query.DateDebut)) {
            logger.warn(`DateDebut invalide : ${req.query.DateDebut}\n`);
            return res.status(400).json({ message: "Il ne peut pas y avoir plusieurs dates de début." });
        } else {
//          Vérification d'une seule date
            if (!regexDate.test(req.query.DateDebut)) {
                logger.warn(`DateDebut invalide : ${req.query.DateDebut}\n`);
                return res.status(400).json({ message: "Format de date invalide. Utiliser YYYY-MM-DD hh." });
            }
        }
    }


//  On vérifie le format et le nombre de DateFin dans la requête
    if (req.query.DateFin) {
        const regexDate = new RegExp("^\\d{4}-\\d{2}-\\d{2} ((0\\d)|(1\\d)|(2[0-3]))$");


        if (Array.isArray(req.query.DateFin)) {
            logger.warn(`DateFin invalide : ${req.query.DateFin}\n`);
            return res.status(400).json({ message: "Il ne peut pas y avoir plusieurs dates de fin." });
        } else {
//          Vérification d'une seule date
            if (!regexDate.test(req.query.DateFin)) {
                logger.warn(`DateFin invalide : ${req.query.DateFin}\n`);
                return res.status(400).json({ message: "Format de date invalide. Utiliser YYYY-MM-DD hh." });
            }
        }
    }


    next();
}
```

2.2 - Tests unitaires

Les tests unitaires de toutes les classes présentes sur le diagramme seront réalisés, cependant j'ai décidé de ne présenter que ceux qui sont en rapport avec le code que j'ai présenté ci-dessus. Il s'agit des tests de la requête de l'API prenant différents critères en paramètre et retournant les données en fonction de ceux-ci.

Elle est vérifiée par la méthode « validParams » de la même classe, ainsi les différentes possibilités ont été testées.

2.2.1 -Test unitaire de l'API et sa requête de filtrage de données

Élément testé :	Ces tests seront effectués sur les méthodes de la classe CAPI, sans la connexion jusqu'à la classe CDatabase, permettant ainsi une vérification seule de la classe CAPI.		
Objectif du test :	Ce test vérifiera les différentes possibilités de requête de l'API avec différentes données de test pour vérifier du bon renvoi des données ou d'un message d'erreur.		
Nom du testeur :	Théo Bourgoin	Date :	27/05/2025
Moyens mis en œuvre :	Logiciel : API-Rest	Matériel : Machine virtuelle hôte	Outil de développement : Node JS & Express.

Procédure du test :

Id	Description du vecteur de test	Résultat attendu	Résultat obtenu	Validation (O/N)
1.1	Requête renvoi de toutes les données Ce test correspond à l'utilisation de la requête « allData » qui permets donc de retourner toutes les données de la base. Requête : curl "http://localhost:8080/SmartTerritories/allData"	On attend le code retour correspondant aux données en échantillon.	{ Capteur: "Roseraie", TypeDeDonnee: "Temperature", Date: "2025-05-24 15", Valeur: 18 }, { Capteur: "Saint-Serge", TypeDeDonnee: "Temperature", Date: "2025-05-28 08", Valeur: 12 }, { Capteur: "Saint-Serge", TypeDeDonnee: "Humidite", Date: "2025-05-26 09",	O

			Valeur: 70 }, { Capteur: "Monplaisir", TypeDeDonnee: "Temperature", Date: "2025-05-26 09", Valeur: 13 }	
1.2	Requête de filtre vide Ce test correspond à l'utilisation de la requête « dataFilter » ne comprenant aucun filtre. Requête : curl "http://localhost:8080/SmartTerritories/dataFilter"	On attend le code retour statut http 200 (success) avec les données correspondant à la requête	« Requête reçue : GET /SmartTerritories/dataFilter avec paramètres : {} Tentative de requête sans filtre. Utilisation de /allData recommandée. »	O
2.1	Filtre par Capteur La suite de test suivante correspond à un filtre par critère simple : ainsi on va chercher les données en entrant un seul critère à la fois. Ici on nous retourne les données suivantes pour le capteur Saint-Serge Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?Capteur=Saint-Serge"	On attend le retour des données qui comportent le capteur donné	{ Capteur: "Saint-Serge", TypeDeDonnee: "Temperature", Date: "2025-05-28 08", Valeur: 12 }, { Capteur: "Saint-Serge", TypeDeDonnee: "Humidite", Date: "2025-05-26 09", Valeur: 70 }	O
2.2	Filtre par TypeDeDonnee Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?TypeDeDonnee=Temperature"	On attend le retour des données qui comportent le TypeDeDonnee donné.	{ Capteur: "Roseraie", TypeDeDonnee: "Temperature", Date: "2025-05-24 15", Valeur: 18 }, { Capteur: "Saint-Serge", TypeDeDonnee: "Temperature", Date: "2025-05-28 08", Valeur: 12 }, {	O

			Capteur: "Monplaisir", TypeDeDonnee: "Temperature", Date: "2025-05-26 09", Valeur: 13 }	
2.3	Filtre par Date Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?Date=2025-05-26%2009"	On attend le retour des données qui comportent la date donnée.	{ Capteur: "Saint-Serge", TypeDeDonnee: "Humidite", Date: "2025-05-26 09", Valeur: 70 }, { Capteur: "Monplaisir", TypeDeDonnee: "Temperature", Date: "2025-05-26 09", Valeur: 13 }	O
2.4	Filtre avec plage de dates Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?DateDebut=2025-05-25%2009&DateFin=2025-05-26%2009"	On attend le retour des données qui sont présentes dans la plage donnée.	{ Capteur: "Saint-Serge", TypeDeDonnee: "Humidite", Date: "2025-05-26 09", Valeur: 70 }, { Capteur: "Monplaisir", TypeDeDonnee: "Temperature", Date: "2025-05-26 09", Valeur: 13 }	O
2.5	Filtre avec limite Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?Limite=3"	On attend le retour des données maximum correspondant à la limite. Si limite = 3 alors 3 données.	{ Capteur: "Roseraie", TypeDeDonnee: "Temperature", Date: "2025-05-24 15", Valeur: 18 }, { Capteur: "Saint-Serge", TypeDeDonnee: "Temperature",	O

			Date: "2025-05-28 08", Valeur: 12 }, { Capteur: "Saint-Serge", TypeDeDonnee: "Humidite", Date: "2025-05-26 09", Valeur: 70 }	
3	Valeur inconnue Ce test correspond à un retour de données dit « valide » mais qui renvoie à un filtre non connu par la base de données : on filtre par une valeur qui n'existe pas. Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?Capteur=blablabla"	Message : « Aucune donnée trouvée avec ces critères »	« Aucune donnée trouvée avec les critères fournis »	O
4	Mauvais format de date Le format de la date étant très stricte pour l'insertion dans la base, il faut être sur que l'utilisateur ne se trompe pas. Dans le cas d'une erreur on renvoit donc le format exact à suivre Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?Date=26-05-2025%2009"	Message « Format de date invalide. Utiliser YYYY-MM-DD hh. »	{"message":"Format de date invalide. Utiliser YYYY-MM-DD hh."}	O
5	Mauvais critère Dans le cas où l'utilisateur se trompe de critère, que ce soit une faute de syntaxe ou une simple erreur, il faut renvoyer un avertissement. Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?NouveauCritère=Valeur"	Message « Paramètres invalides détectés. »	{"message":"Paramètres invalides détectés."} Ainsi que du coté de l'API le message suivant : « Paramètre(s) invalide(s) reçu(s) : NouveauCritère »	O
6.1	Plusieurs dates de début On laisse le choix à l'utilisateur d'afficher les données en fonction d'une plage de dates. Cependant on ne peut pas afficher de plage comprenant deux dates de début Requête :	Message : « Il ne peut pas y avoir plusieurs dates de début ».	« Il ne peut pas y avoir plusieurs dates de début. »	O

	curl "http://localhost:8080/SmartTerritories/dataFilter?DateDebut=2025-05-24%2009&DateDebut=2025-05-26%2009"			
6.2	Plusieurs dates de fin Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?DateFin=2025-05-26%2009&DateFin=2025-05-27%2009"	Message : « Il ne peut pas y avoir plusieurs dates de fin ».	« Il ne peut pas y avoir plusieurs dates de fin. »	O
6.3	Date de début/fin et date simple Malgré le fait qu'on puisse entrer une plage de date ou une date seule, les deux combinés entraine une erreur. En effet la date seule serait ignorée et on retournerait seulement la plage de dates. Il nous faut donc un avertissement pour l'utilisateur Requête : curl "http://localhost:8080/SmartTerritories/dataFilter?DateDebut=2025-05-24%2009&Date=2025-05-24%2011"	Message : « La date (simple) sera ignorée car vous avez entré une date de début et/ou de fin »	"La date (simple) sera ignorée car vous avez entré une date de début et/ou de fin"	O
7	Insertion des données Ce test correspond à une insertion des données via le broker & client MQTT. En supposant que ces données seraient validées par la classe CAnalyse qui contient son propre plan de test. Ainsi on nous retourne bien la validation de l'insertion (les données ne sont pas inscrites dans la base car aucune connexion n'est mise en place). Requête : curl -X POST "http://localhost:8080/SmartTerritories/addData" -H "Content-Type: application/json" -d '{"Capteur": "Roseraie", "TypeDeDonnee": "Temperature", "Date": "2025-05-26 09", "Valeur": 15}'	Message : « Insertion simulée réussie » en renvoyant les données insérées.	"Insertion simulée réussie"	O
Conclusion du test :	Les tests des méthodes n'ont donné aucune erreur, ils ont tous été validés sans problème. On en conclut que la classe fonctionne sans problème, elle est donc validée.			

2.2.2 -Problèmes rencontrés

Les problèmes rencontrés n'ont pas été à propos des tests eux-mêmes mais au niveau de l'installation des différentes composantes du projet. En effet, je rencontrais des problèmes de compatibilité entre ma version d'Ubuntu et MongoDB, le bon compromis a été Ubuntu 20.04 (focal) et MongoDB v.4.4.29. L'installation de mongoose (liaison à ma base MongoDB) m'a demandé de mettre à jour npm, étant ma base Node JS.

3 - Réalisation du cas d'utilisation « Archiver les données »

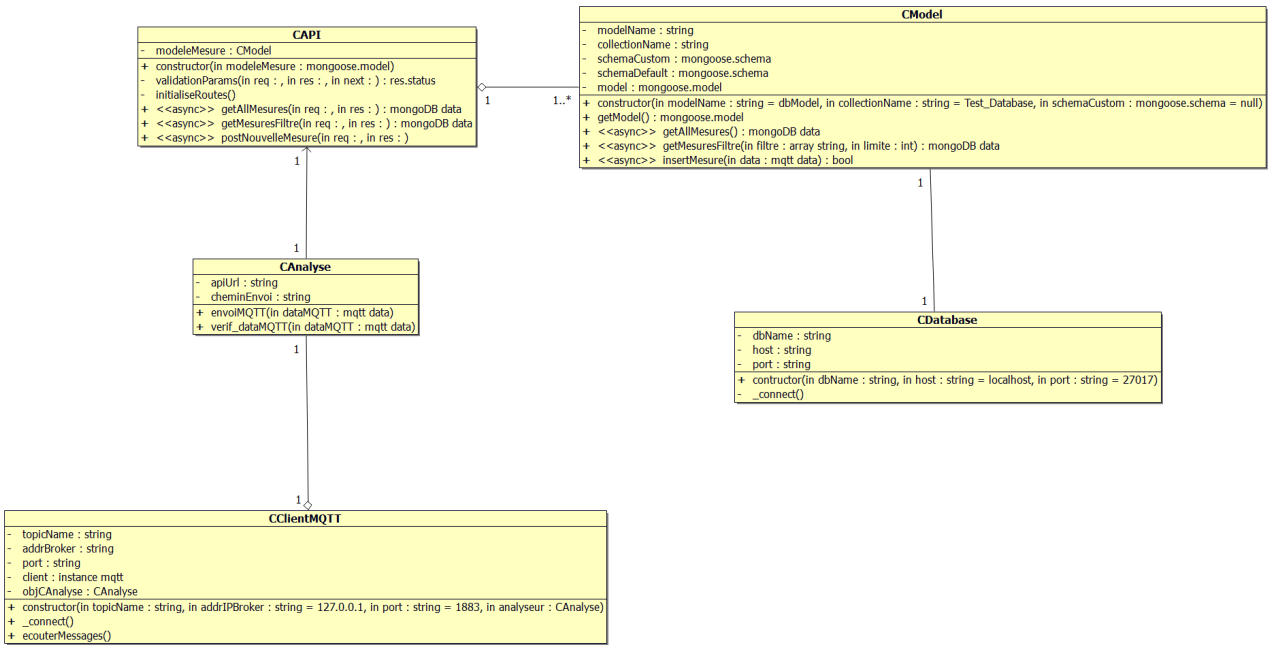
La deuxième partie du projet me concernant, consistait à récupérer les données des capteurs météorologiques et de pollution, puis les enregistrer dans ma base de données. Le choix de la base de données étant déjà fait, il ne me fallait que réfléchir à un moyen de communication avec les capteurs Nb-IoT.

La réflexion avec les ER m'a amené à différents choix de communication, et ainsi différents protocoles :

CoAP	Leger, basse consommation, fait pour réseaux à faible bande passante Fonctionne sur UDP
MQTT	Leger, basé sur publication/abonnement Fait pour communication bidirectionnelle et temps réel
HTTP/HTTPS	Si ressources suffisantes (lourd) alors très utile Besoin de connexion fiable & lourd à cause des entêtes
LwM2M	Plus leger que HTTP & basé sur CoAP, utile à grande échelle Gère des dispositifs IoT, collecte les données & effectue des commandes à distance

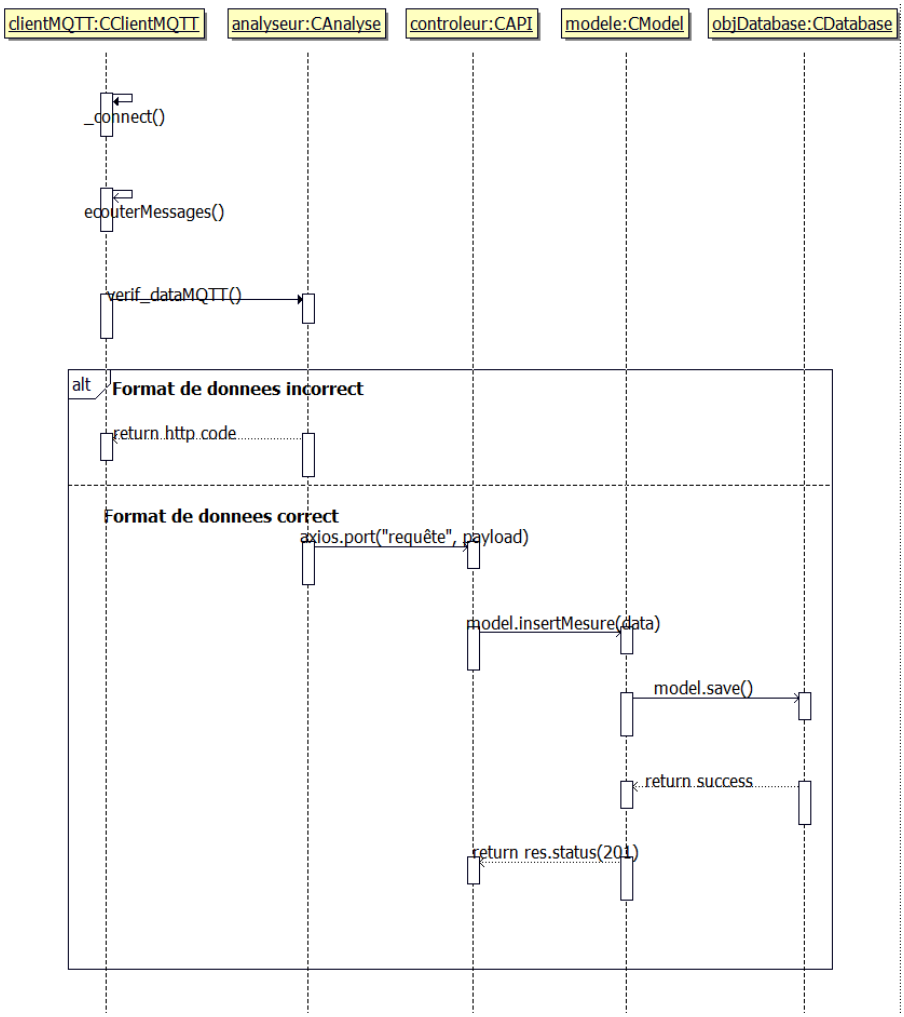
Les différents protocoles sont intéressants pour une solution IoT. Cependant, pour un projet IQA qui contient des capteurs qui envoient des données asynchroniquement, MQTT est la meilleure solution : on s'abonne à un « topic » sur un « broker » MQTT, les capteurs envoient les données à ce broker et tous les appareils qui sont abonnés à ce topic les reçoivent.

Il me faut donc créer un client MQTT qui s'abonnera au topic puis enverra les données à l'API pour les stocker dans l'API.



Viennent donc les classes CClientMQTT et CAnalyse. La classe CClientMQTT, grâce à son constructeur, pourra se connecter à différents topics suivant les besoins de l'utilisateur, tout en pouvant se connecter à un broker de son choix.

Ce même client utilisera un attribut de la classe CAnalyse pour accéder à sa méthode de vérification des données. Ainsi les données seront vérifiées avant d'être envoyées.



3.1 - Conception détaillée

Les éléments principaux du code ont déjà été installés, cependant pour cette partie, il nous faudra installer d'autres bibliothèques :

Installation de la bibliothèque axios, il s'agit de la bibliothèque gérant les requêtes HTTP pour node JS.	npm install axios
Installation de la bibliothèque MQTT, permettant de s'abonner au topic et se connecter au broker.	npm install mqtt

Pour cette partie, l'analyse se concentrera sur l'insertion des données dans la base de données, mais surtout sur leur vérification. En effet, avant de les insérer il faut vérifier leur format (date, syntaxe…) ainsi que leur intégrité, tout en vérifiant qu'elles sont complètes.

Méthode de réception des messages MQTT dans le Client. *Vérification que la connexion est toujours en cours*	ecouterMessages() { if (!this.client) { console.error("Client MQTT non connecté."); return; }

On stocke les données dans une variable *On les envoie à l'analyse* *Si une erreur est survenue alors on la reporte.*	this.client.on('message', async (topic, message) => { try { const payload = JSON.parse(message.toString()); console.log('Message reçu :', payload); await this.objCAnalyse.verif_dataMQTT(payload); } catch (err) { console.error('Erreur de traitement du message :', err.message); } }); }
Vérification des données après envoi par le Client. *On stocke les champs possibles dans le paquet* *On vérifie que les données soient complètes* *On vérifie le format de la date* *On vérifie que la valeur est bien un nombre*	verif_dataMQTT(dataMQTT) { const champsObligatoires = ["Capteur", "TypeDeDonnee", "Date", "Valeur"]; for (const champ of champsObligatoires) { if (!(champ in dataMQTT)) { console.error(`Champ manquant : ${champ}`); return false; } } const regexDate = new RegExp("^\\d{4}-\\d{2}-\\d{2} ((0\\d)\|(1\\d)\|(2[0-3]))$"); if (!regexDate.test(dataMQTT.Date)) { console.error("Format de date invalide. Format attendu : 'AAAA-MM-JJ HH'"); return false; } if (isNaN(parseFloat(dataMQTT.Valeur))) { console.error("Valeur non numérique"); return false; } return true; }

3.2 - Tests unitaires

Les tests unitaires de ces méthodes se sont faits après ceux de l'API, en effet, selon l'organisation du projet, et la planification, cette partie du projet était pour moi la deuxième à être traitée.

On testera ici toutes les possibilités encadrées par la méthode « verif_dataMQTT », ainsi que le bon enregistrement des données.

3.2.1 -Test unitaire du client MQTT

Élément testé :	Ces tests seront effectués sur les méthodes de la classe CClientMQTT, on travaillera sur la réception des données ainsi que les possibilités de création d'une nouvelle instance de la classe.		
Objectif du test :	Ce test vérifiera la bonne réception des données, ainsi que les différents codes d'erreur lors de leur validation. Les différents scénarios seront validés.		
Nom du testeur :	Théo Bourgoin	Date :	21/05/2025
Moyens mis en œuvre :	Logiciel : Broker MQTT	Matériel : Machine virtuelle hôte	Outil de développement : Node JS & Express.

Procédure du test :

Id	Description du vecteur de test	Résultat attendu	Résultat obtenu (voir le document de plan de test sur le github pour avoir l'échantillon exact des données)	Validation (O/N)
1.1	Instanciation de client MQTT par défaut Ce test correspond au constructeur de la classe, on instancie un nouvel objet du client MQTT avec les valeurs proposées par défaut ainsi qu'un analyseur lambda (classe CAnalyse non incluse).	topicName (donné par l'utilisateur) addrBroker : 127.0.0.1 (localhost par défaut) port : 1883 (port MQTT par défaut)	« Client MQTT instancié : CClientMQTT { topicName: 'test/topic', objCAnalyse: {}, addrBroker: '127.0.0.1', port: '1883', client: null } »	O
1.2	Client MQTT avec données personnalisées Ce test correspond au constructeur de la classe, on instancie un nouvel objet de ClientMQTT, cette fois-ci avec des données personnalisées. L'analyseur sera toujours lambda.	Les critères donnés doivent coïncider avec le retour topicName (donné) addrBroker : donné dans constructeur port : donné dans constructeur	« Client MQTT personnalisé instancié : CClientMQTT { topicName: 'custom/topic', objCAnalyse: {}, addrBroker: '192.168.1.100', port: '1884', client: null } »	O
2	Connexion au broker MQTT Ce test correspond à la méthode de connexion de la classe (_connect) permettant de se connecter au broker MQTT.	On s'attend à recevoir le message qui nous valide la connexion au broker ainsi que le topic : « Connecté au broker MQTT Abonné au topic : *topic name* »	« Connecté au broker MQTT Abonné au topic : test/topic »	O

3	Ce test correspond à la méthode de réception de messages MQTT. On prépare donc il client MQTT puis on lui envoie un message à l'aide du topic choisi et du broker mosquitto (ici en localhost) puis après avoir lancé le programme, copier-coller la ligne de broker mosquitto (précisée en dessous du code) dans un autre terminal. Ainsi on envoie des données dans un format accepté (JSON) et les données sont reçues par le client	On s'attend à recevoir les données reçues sans erreur, et toujours dans le bon format. Ex : « { Capteur : « … », TypeDeDonnee : « … », Date : « … », Valeur : « … » } »	« Message reçu : { Capteur: 'Roseraie', TypeDeDonnee: 'Temperature', Date: '2025/05/19 15', Valeur: 23.4 } »	O
4	On ne précise pas que les données doivent être reçues en JSON, une erreur de format est vite arrivée. Il faut donc vérifier ce qui nous est retourné lors de cette erreur. On répète donc la même procédure que le test précédent, mais cette fois-ci le message MQTT envoyé n'est pas bon.	On s'attend à ce que les données soient refusées, avec le message d'erreur nous permettant de savoir ce qui ne va pas.	« Erreur de traitement du message : Unexpected token m in JSON at position 0 »	O
Conclusion du test :	Les tests des méthodes n'ont donné aucune erreur, ils ont tous été validés sans problème. On en conclut que la classe fonctionne sans problème, elle est donc validée.			

3.2.2 -Problèmes rencontrés

Les tests en eux-mêmes n'ont pas posé de problème, cependant la gestion des erreurs en fonction des données a été problématique, principalement du fait d'être en accord avec les électroniciens sur le format d'envoi des données. Principalement le format de la date. Un autre problème a été la connexion au réseau public, résolu par une carte SIM possédant une adresse IP publique, mise en place sur la passerelle.

4 - Bilan de la réalisation personnelle

Le projet Smart Territories est un projet IQA qui nécessite un envoi de données venant de capteurs, qui seront ensuite traitées et envoyées dans une base de données. Ces mêmes données seront ensuite demandées par un utilisateur via une page web et demandée par le serveur web. Les cas d'utilisations principaux de ce projet ont été respectés et mis en place.

Cependant, une autre amélioration possible serait la mise en place de sécurité : chiffrer les données envoyées, mettre en place un système d'identifiants sur l'API ainsi que sur le broker MQTT (et notre topic personnel). Il s'agit d'une partie de plus en plus importante dans les réseaux privés, de nos jours. Il serait tout aussi important de mettre en place des VLANs pour sécuriser le réseau ainsi qu'utiliser le port 8883 (port MQTT permettant le chiffrement des données).

Malgré les retards pris, le projet est fonctionnel. Même si certaines parties sont encore à développer pour un étudiant ou un autre, il n'est potentiellement pas intégrable dans une entreprise professionnelle pour le moment.