***Cahier des charges personnel***

Table des matières

[Introduction : 1](#_Toc168236627)

[Compte rendu d’activité : 2](#_Toc168236628)

[Diagramme synoptique personnel : 3](#_Toc168236629)

[Diagramme des cas d’utilisation détaillé : 3](#_Toc168236630)

[Diagramme d’exigences : 4](#_Toc168236631)

[Scénarii : 6](#_Toc168236632)

[Diagrammes de classes : 8](#_Toc168236633)

[1. C++ : 8](#_Toc168236634)

[2. Javascript : 10](#_Toc168236635)

[Diagrammes de séquence : 11](#_Toc168236636)

[Dossier de tests et de validation (extrait) : 11](#_Toc168236637)

# Introduction :

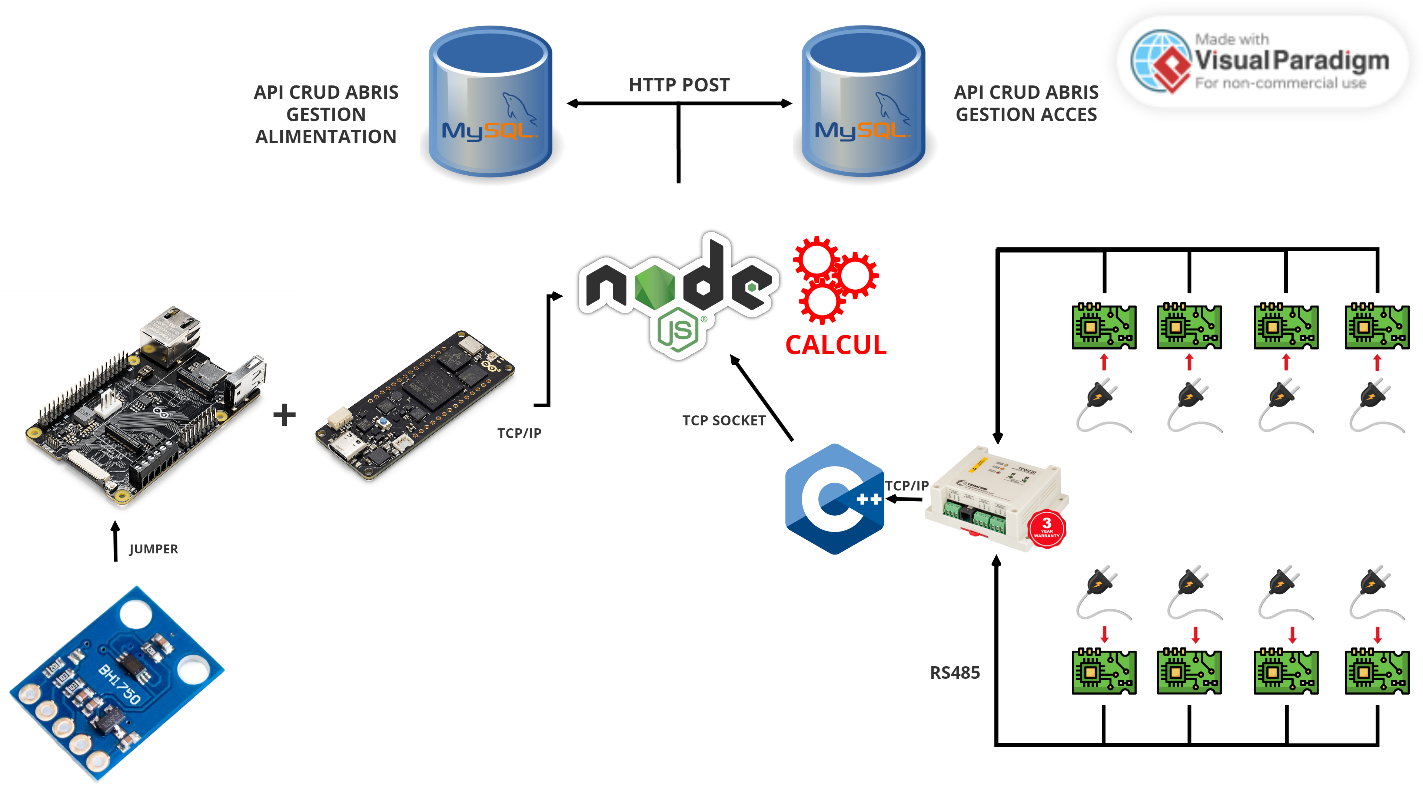
Mon rôle lors de ce projet a été de sélectionner l’équipement adéquoit afin de pouvoir capter les informations nécéssaires à l’application web de ce projet. D’une part pour un système électronique de mesure de luminosité. Puis de plusieurs autres système afin de capter les données de puissance des huit box compris dans l’abris à bicycles éco-conçus.  
  
 Une fois ces choix faits, j’ai dû travailler sur les programmes qui se chargent de récupérer ces données physiques, et de les envoyer au serveur Node.js. J’ai donc réaliser plusieur programmes C++ pour cette tâche.

Afin de pouvoir mieux regrouper et traiter toutes ces données. J’ai également dû créer un serveur backend en javascript avec le framework Node.js. Ce serveur réceptionne les données provenants de mes différents programmes C++, calcule les données manquantes, puis les envoie en base de données par le biais de plusieurs API CRUD créées d’une part par l’étudiant 2 de notre groupe, Simon Briaux, et d’autre part par un autre étudiant travaillant sur ce même projet, mais avec un groupe et un objectif différent du notre, Hassan Paziaud.

# Compte rendu d’activité :

|  |  |
| --- | --- |
| **Semaine 1** : du 15 au 21 janvier | → Compréhension du sujet → Début de la revue 1 → Réalisation du livrable 1 (présentation, synoptique, déploiement, use case, use case perso) |
| **Semaine 2** : du 22 au 28 janvier | → Réalisation du livrable 2 (exigences, scénarii) |
| **Semaine 3** : du 29 janvier au 4 février | → Correction du livrable 1 et 2 → Réalisation du livrable 3 (MCD, classes, séquences) |
| **Semaine 4** : du 5 au 11 février | → Correction du livrable 2 → Réalisation du livrable 4 (GANTT, CRA, outils organisationnels, DTV) |
| **Semaine 5** : du 12 au 18 février | → Réalisation du livrable 5 (Etude comparative, Bon de commande, présentation de revue 1) → Fin de la revue 1 |
| **Semaine 6** : du 19 au 25 février | → Début de la revue 2 → Réalisation du livrable 1 (MCD, jeu de données)  → Correction de la revue 1 |
| **Semaine 7** : du 11 au 17 mars | → Réalisation du livrable 2 (1er module de test) → Correction de la revue 1 |
| **Semaine 8** : du 18 au 24 mars | → Réalisation du livrable 3 (Séquences, GitHub, tests unitaires) → Etude comparative |
| **Semaine 9** : du 25 au 31 mars | → Réalisation du livrable 4 (2ème module de test) → Etude comparative |
| **Semaine 10** : du 1 au 7 avril | → Réalisation du livrable 5 (Séquences, analyse détaillée) → Etude comparative |
| **Semaine 11** : du 8 au 14 avril | → Cahier des charges → Programmation |
| **Semaine 12** : du 13 au 19 mai | → Programmation → Fin de la revue 2 |
| **Semaine 13** : du 20 au 26 mai | → Début de la revue 3 → Programmation → Présentation de revue 2 |
| **Semaine 14** : du 27 mai au 2 juin | → Programmation → Maquettage → Présentation de revue 3 |

# Diagramme synoptique personnel :



Ce diagramme représente toute ma partie dans le projet sur le plan physique.

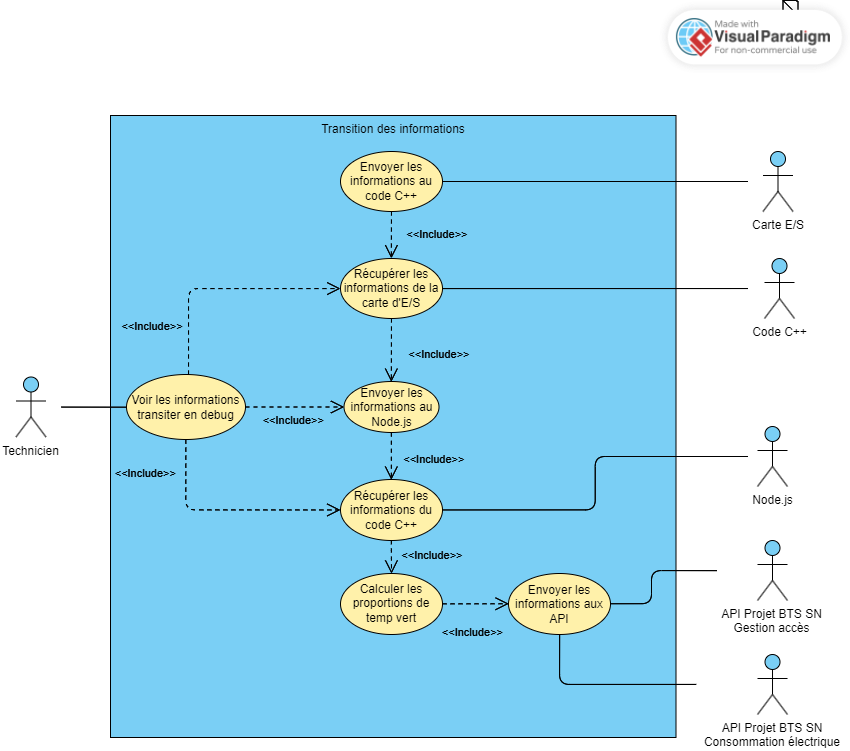
On peut y voir à gauche le système pour capter la luminosité des panneaux photovoltaïques.

A droite les différents systèmes pour capter les informations de puissance des 8 box de l’abris à bicycles éco-conçus.

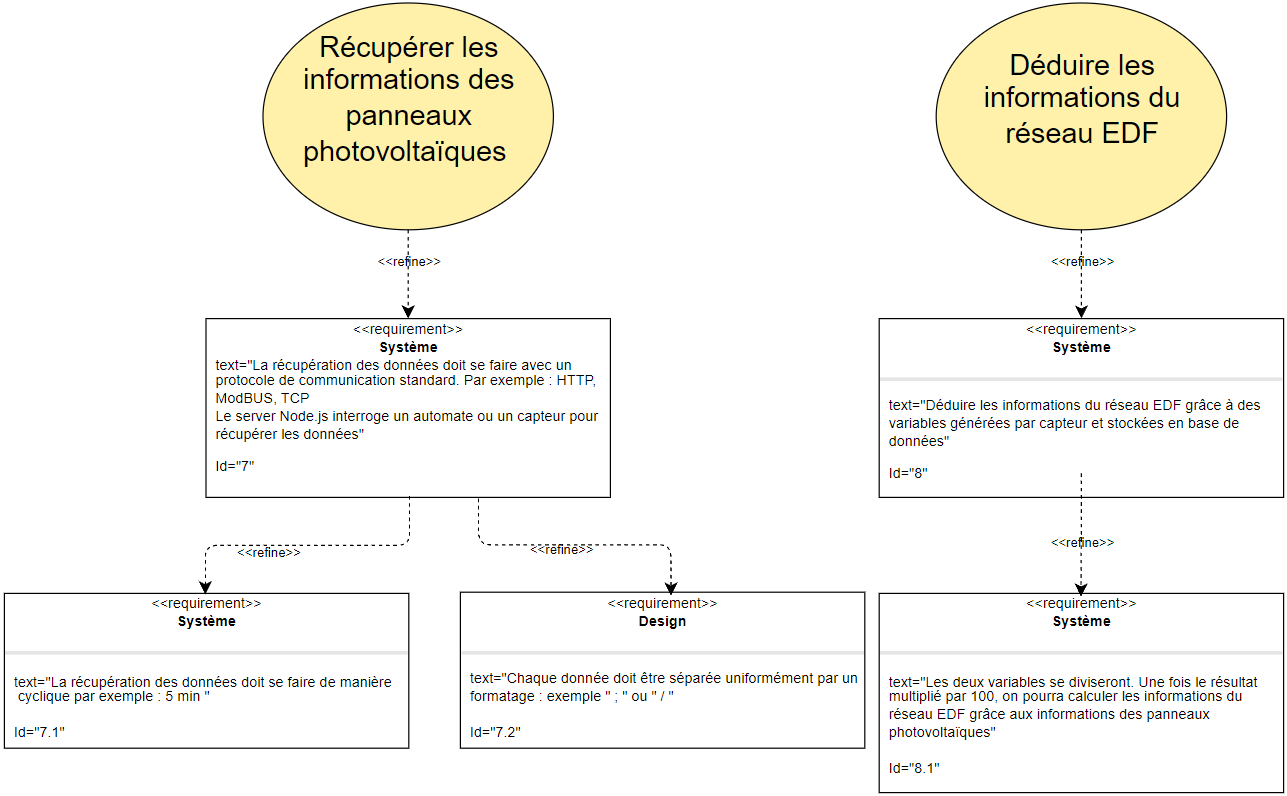
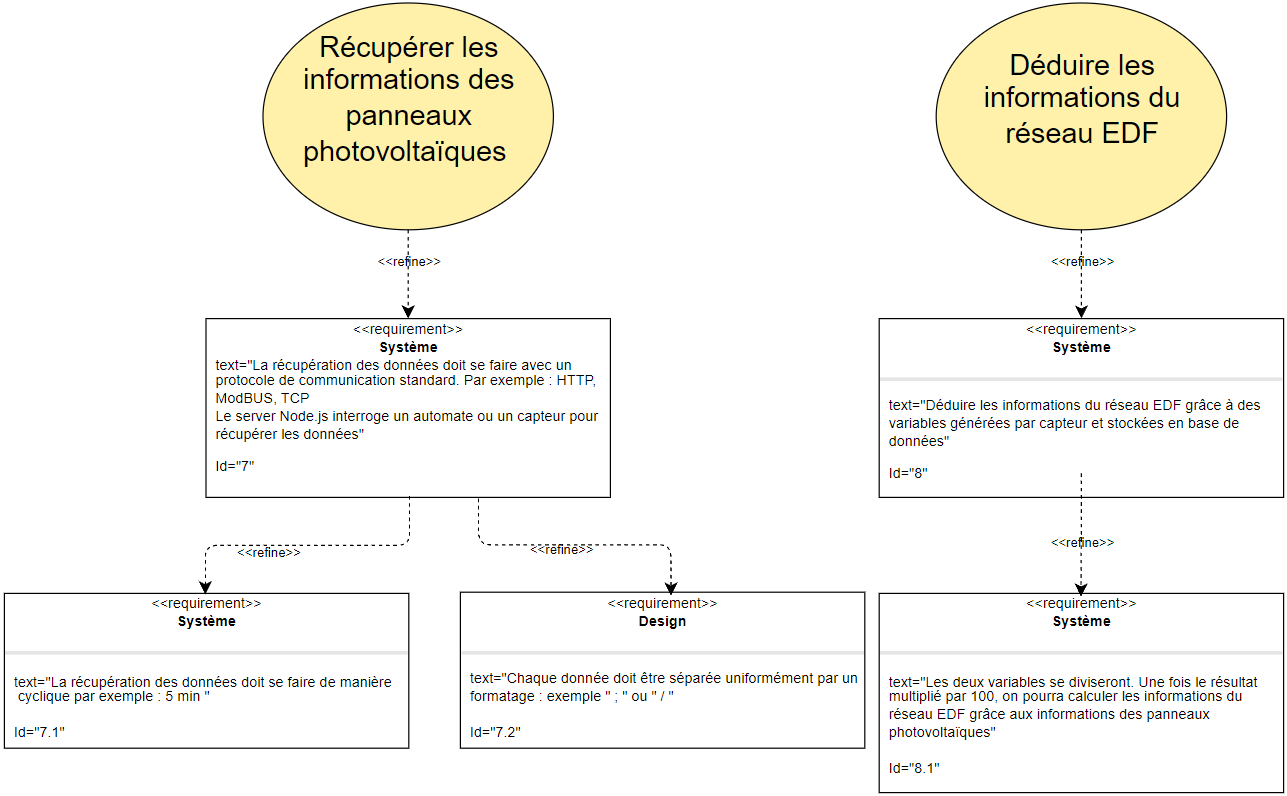
Au centre le serveur Node.js qui regroupe et fait le calcul des données manquantes.  
En haut les deux bases de données auxquelles j’envoie les données traitées.

# Diagramme des cas d’utilisation détaillé :

Ce diagramme représente l’acheminement des données depuis les différents capteurs jusqu’aux API CRUD.



# Diagramme d’exigences :



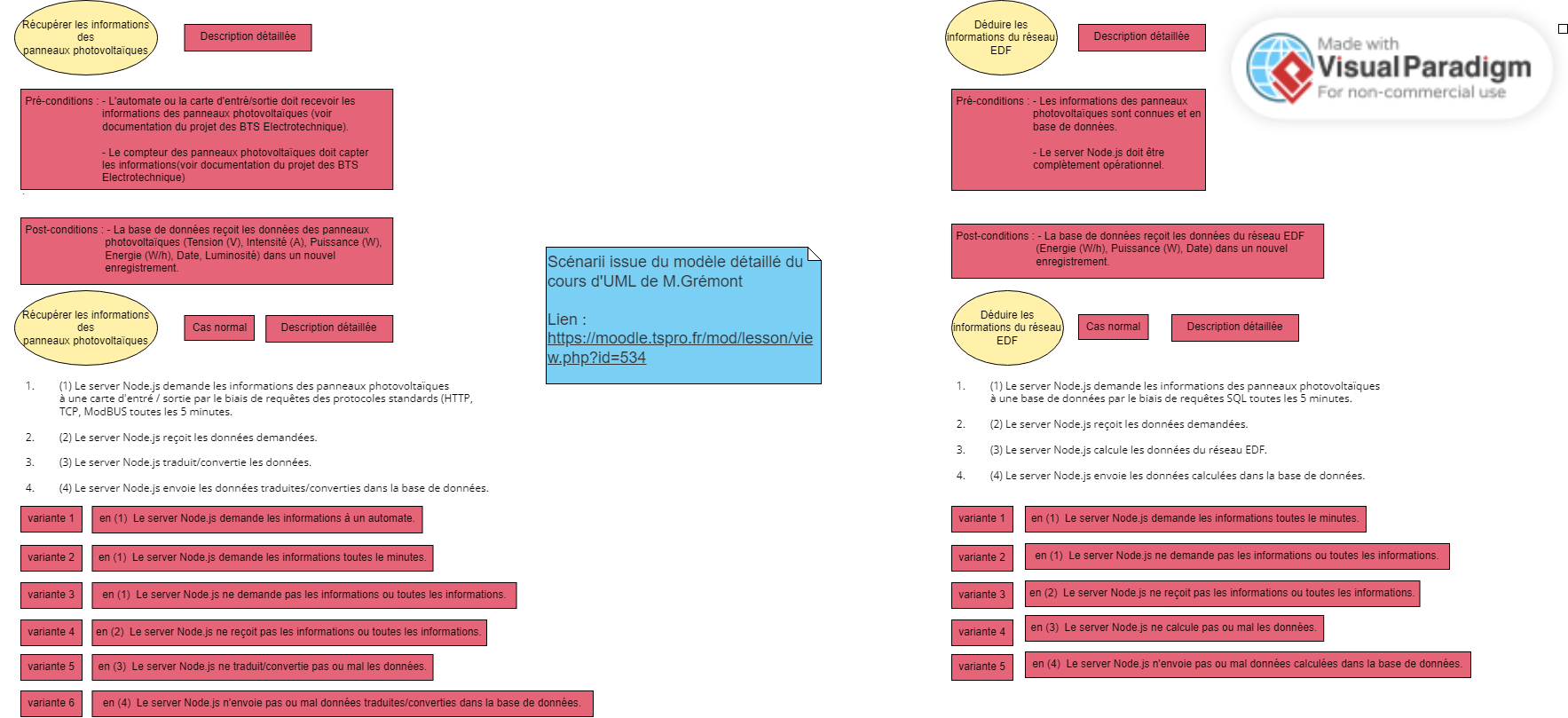
Ma partie du diagramme d’exigences se divise en deux.

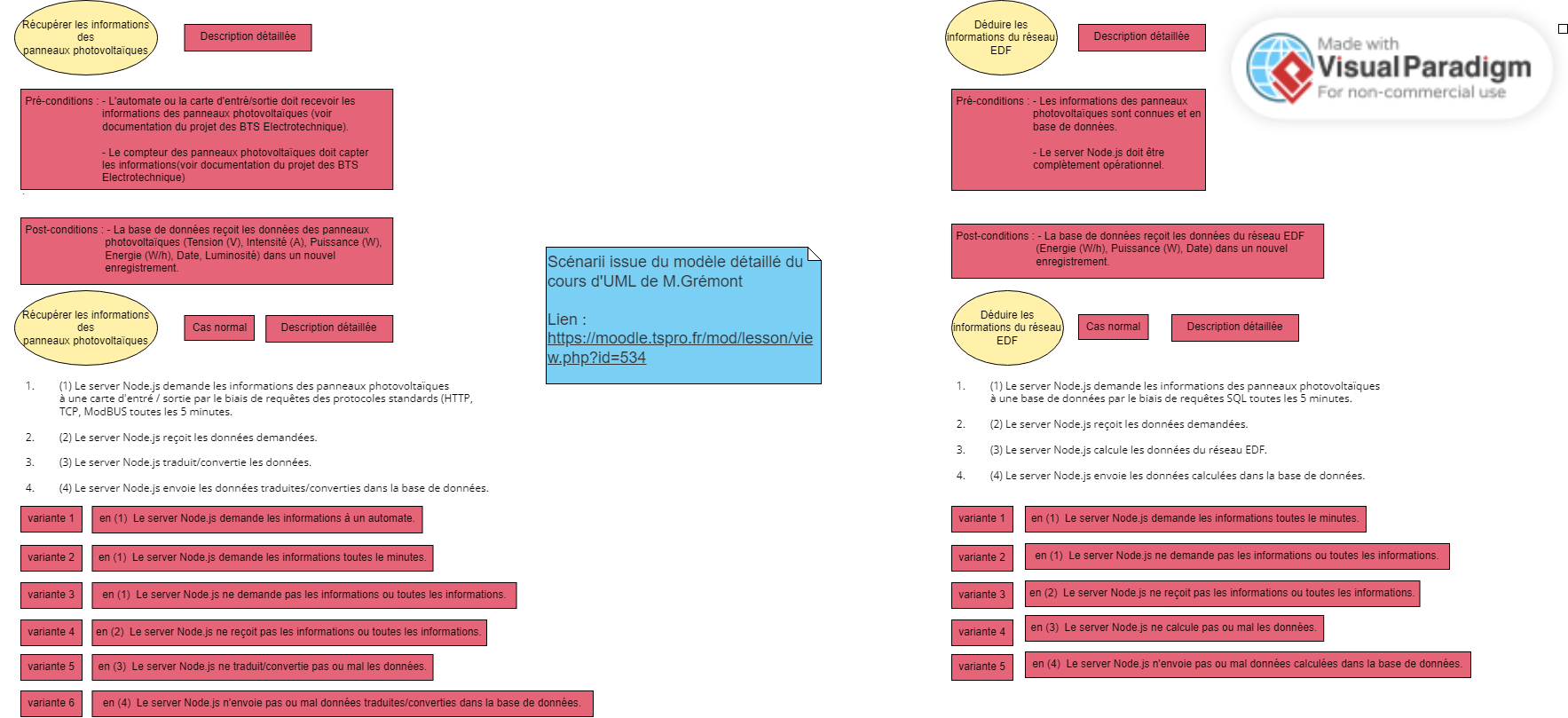
Récupérer les informations des panneaux photovoltaïques se concentre sur la manière dont nous allons récupérer les informations.

Déduire les informations du réseau EDF se concentre sur les calculs nécessaires pour avoir les informations du réseau EDF.

Nous avons décidé d’ajouter cette exigence au projet car nous aurions très bien pu mettre deux capteurs afin d’avoir ces informations. Cependant, avec le budget donné, nous avons préféré économiser de l’argent.

# Scénarii :





Ces scénarios répondent à deux de nos fonctionnalités de notre diagramme des cas d’utilisation simplifié :  
  
- Récupérer les informations des panneaux photovoltaïque.  
  
- Déduire les informations du réseau EDF

# Diagrammes de classes :

## C++ :



Les classes CarteES et Luminosite sont relativement similaires à l’exeption des données qu’elles envoyent.

CarteES envoit les données binaires correspondantes à si la puissance consommée provient des panneaux photovoltaïques (1), ou du réseau EDF (0) ; un tableau de 8 booléens (0 ou 1) correspondant à l’état d’alimentation de chacune des box de l’abris à bicycles éco-conçu ; ainsi que la date à laquelle les informations ont été envoyées. Nous générons cette date en amont pour des soucis de communications. Car si la date est générée lors de l’envoi des données en base de données par exemple. Si les données venaient à être envoyées 3 à la fois à cause d’un bloquage, on aurait 3 fois la même date. Toutes les données binaires de CarteES sont générées aléatoirement.

Luminosité se charge d’envoyer les données de luminosité générées aléatoirement, ainsi qu’une date.

Ces deux classes utilisent la librairie boost 1.84.0 afin d’avoir une communication sans surcouches socket TCP avec le serveur Node.js .

  
  
 La classe Capteur\_luminosite est la continuïté de la classe Luminosite.  
Capteur\_luminosite n’utilise plus de données générées aléatoirement comme les deux précédentes classe, mais récupère des données réelles provenants d’un capteur de luminosité. Capteur\_luminosite contient donc des attributs et méthodes suplémentaires afin de pouvoir capter ces données.

Cette classe envoit des données de puissance, luminosité, intensité, date au serveur Node.js .

Comparée aux deux classes précédentes, cette classe fonctionne en 32bits car elle doit utiliser la librairie Dask afin de pouvoir réceptionner les informations d’une carte AdLink 9111. La librairie boost 1.84.0 des précédentes classe a été remplacé par les librairies standard de C++ car boost ne compilait qu’en 64bits et ne possaidait pas de version 32bits.

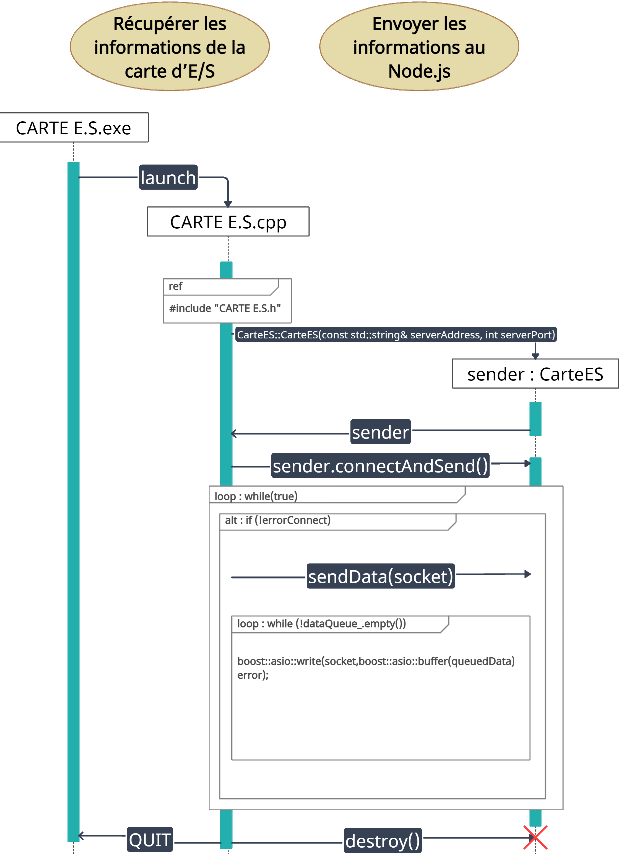
## Javascript :



La classe Compteurs fait marcher le serveur Node.js de manière asynchrone. On reçoit en premier lieu les données du C++, puis on calcule les données manquantes dans une boucle. Enfin, on les mets en file d’attente si on arrive pas à se connecter aux API qui envoyent les données en base de donnée. Sinon, on les envoye directement en base de données sans les mettre en file d’attente.   
  
Ces diagrammes de classe répondent à deux de nos fonctionnalités de notre diagramme des cas d’utilisation simplifié :  
  
- Récupérer les informations des panneaux photovoltaïque.  
  
- Déduire les informations du réseau EDF

# Module de test 1, 2, 3 :

## Diagramme de séquence :



Le premier diagramme de séquence représente le module de test 1, concernant la classe CarteES. Il test si le programme arrive bien à se connecter au serveur Node.js et s’il arrive bien à envoyer ses données.

Je n’en ai pas fait pour les modules de test 2 et 3 car les programmes sont similaire en fonctionnement.

## Code du module 1 :

La classe CarteES fonctionne avec la librairie boost 1.84.0, elle possède un constructeur, une méthode pour se connecter au serveur Node.js, une méthode pour créer une valeur au format DATETIME, une autre pour envoyer les données au serveur Node.js.  
  
#pragma once

#ifndef CARTE\_ES\_H

#define CARTE\_ES\_H

#include <boost/asio.hpp> // Inclusion de la bibliothèque Boost.Asio pour les opérations d'entrée/sortie asynchrones

#include <boost/property\_tree/ptree.hpp> // Inclusion de la bibliothèque Boost.PropertyTree pour manipuler des arbres de propriétés

#include <boost/property\_tree/json\_parser.hpp> // Inclusion de la bibliothèque Boost.PropertyTree pour parser du JSON

#include <deque>

#include <iostream>

#include <chrono>

#include <thread>

#include <ctime>

using boost::asio::ip::tcp; // Utilisation de l'espace de noms TCP de Boost.Asio

class CarteES {

public:

// Constructeur de la classe CarteES

// @param serverAddress : adresse IP du serveur Node.js

// @param serverPort : port d'écoute du serveur Node.js

CarteES(const std::string& serverAddress, int serverPort);

// Méthode pour établir la connexion avec le serveur Node.js et envoyer les données

void connectAndSend();

//Méthode pour créer un string de la date actuelle au format SQL DATETIME

std::string createDateTime();

private:

// Méthode pour envoyer des données au serveur Node.js

// @param socket : socket TCP pour la communication avec le serveur Node.js

bool sendData(tcp::socket& socket);

tcp::endpoint endpoint\_; // Point de terminaison TCP représentant l'adresse IP et le port du serveur Node.js

static const size\_t maxQueueSize = 3; // Définir la taille maximale de la file d'attente (ajustable au choix)

std::deque<std::string> dataQueue\_; // File d'attente pour stocker les données à envoyer en cas d'échec d'envoi

};

#endif /\* CARTE\_ES\_H \*/

## Code du module 2 :

Le module 2 est relativement similaire au module 1, il teste la classe Luminosite, qui fonctionne également avec la librairie boost 1.84.0, elle est similaire à la classe CarteES à l’exeption des données qu’elle envoie. CarteES envoie des données binaires générées aléatoirement représentant les informations de puissances des box. Tandis que Luminosite génére des données de luminosité aléatoirement.

#pragma once

#ifndef LUMINOSITE\_H

#define LUMINOSITE\_H

#include <boost/asio.hpp> // Inclusion de la bibliothèque Boost.Asio pour les opérations d'entrée/sortie asynchrones

#include <boost/property\_tree/ptree.hpp> // Inclusion de la bibliothèque Boost.PropertyTree pour manipuler des arbres de propriétés

#include <boost/property\_tree/json\_parser.hpp> // Inclusion de la bibliothèque Boost.PropertyTree pour parser du JSON

#include <deque>

#include <iostream>

#include <chrono>

#include <thread>

#include <ctime>

using boost::asio::ip::tcp; // Utilisation de l'espace de noms TCP de Boost.Asio

class LUMINOSITE {

public:

// Constructeur de la classe LUMINOSITE

// @param serverAddress : adresse IP du serveur Node.js

// @param serverPort : port d'écoute du serveur Node.js

LUMINOSITE(const std::string& serverAddress, int serverPort);

// Méthode pour établir la connexion avec le serveur Node.js et envoyer les données

void connectAndSend();

//Méthode pour créer un string de la date actuelle au format SQL DATETIME

std::string createDateTime();

private:

// Méthode pour envoyer des données au serveur Node.js

// @param socket : socket TCP pour la communication avec le serveur Node.js

bool sendData(tcp::socket& socket);

tcp::endpoint endpoint\_; // Point de terminaison TCP représentant l'adresse IP et le port du serveur Node.js

static const size\_t maxQueueSize = 3; // Définir la taille maximale de la file d'attente (ajustable au choix)

std::deque<std::string> dataQueue\_; // File d'attente pour stocker les données à envoyer en cas d'échec d'envoi

};

#endif /\* LUMINOSITE\_H \*/

## Code du module 3 :

Le module 3 est légèrement différent aux module 1 et 2. Il teste la classe Capteur\_luminosite, qui fonctionne avec la librairie dask afin de pouvoir communiquer avec la carte AdLink 9111 qui communique avec un véritable capteur de luminosité avec une communication Dsub-37. Sa différence avec les deux précédentes classes et l’utilisation de données réelles. La librairie dask fonctionne en 32bits, ce qui m’a empéché de réutiliser la librairie boost 1.84.0 qui ne fonctionne qu’en 64bits. J’ai donc dû refaire le système de communication avec le serveur Node.js avec les librairies standard.  
  
#ifndef CAPTEUR\_LUMINOSITE\_H

#define CAPTEUR\_LUMINOSITE\_H

#include "Dask.h"

#include <deque>

#include <iostream>

#include <chrono>

#include <thread>

#include <ctime>

#include <vector>

#include <numeric>

#include <string>

#include <sstream>

#include <iterator>

#include <algorithm>

#include <ws2tcpip.h>

#include <winsock2.h>

#pragma comment(lib, "Ws2\_32.lib")

class capteur\_luminosite

{

public:

capteur\_luminosite(const std::string& serverAddress, int serverPort, int carte = PCI\_9111DG);

~capteur\_luminosite();

void lire\_tension(double& tension, int canal = 0, int gamme = AD\_B\_10\_V);

double lire\_tension(int canal = 0, int gamme = AD\_B\_10\_V);

unsigned short lire\_tension\_AI\_ReadChannel(int canal = 0, int gamme = AD\_B\_10\_V);

int getnumcarte() { return this->numcarte; };

void setnumcarte(int value) { this->numcarte = value; };

//------------------------------------------------//

// Méthode pour établir la connexion avec le serveur Node.js et envoyer les données

void connectAndSend();

//Méthode pour créer un string de la date actuelle au format SQL DATETIME

std::string createDateTime();

//------------------------------------------------//

double getresistance() { return this->resistance; };

private:

int m\_carte;

int numcarte;

//------------------------------------------------//

// Méthode pour envoyer des données au serveur Node.js

// @param socket : socket TCP pour la communication avec le serveur Node.js

bool sendData(SOCKET& socket);

std::string serverAddress\_; // Adresse IP du serveur Node.js

int serverPort\_; // Port d'écoute du serveur Node.js

static const size\_t maxQueueSize = 3; // Définir la taille maximale de la file d'attente (ajustable au choix)

std::deque<std::string> dataQueue\_; // File d'attente pour stocker les données à envoyer en cas d'échec d'envoi

sockaddr\_in endpoint\_; // Point de terminaison TCP représentant l'adresse IP et le port du serveur Node.js

//------------------------------------------------//

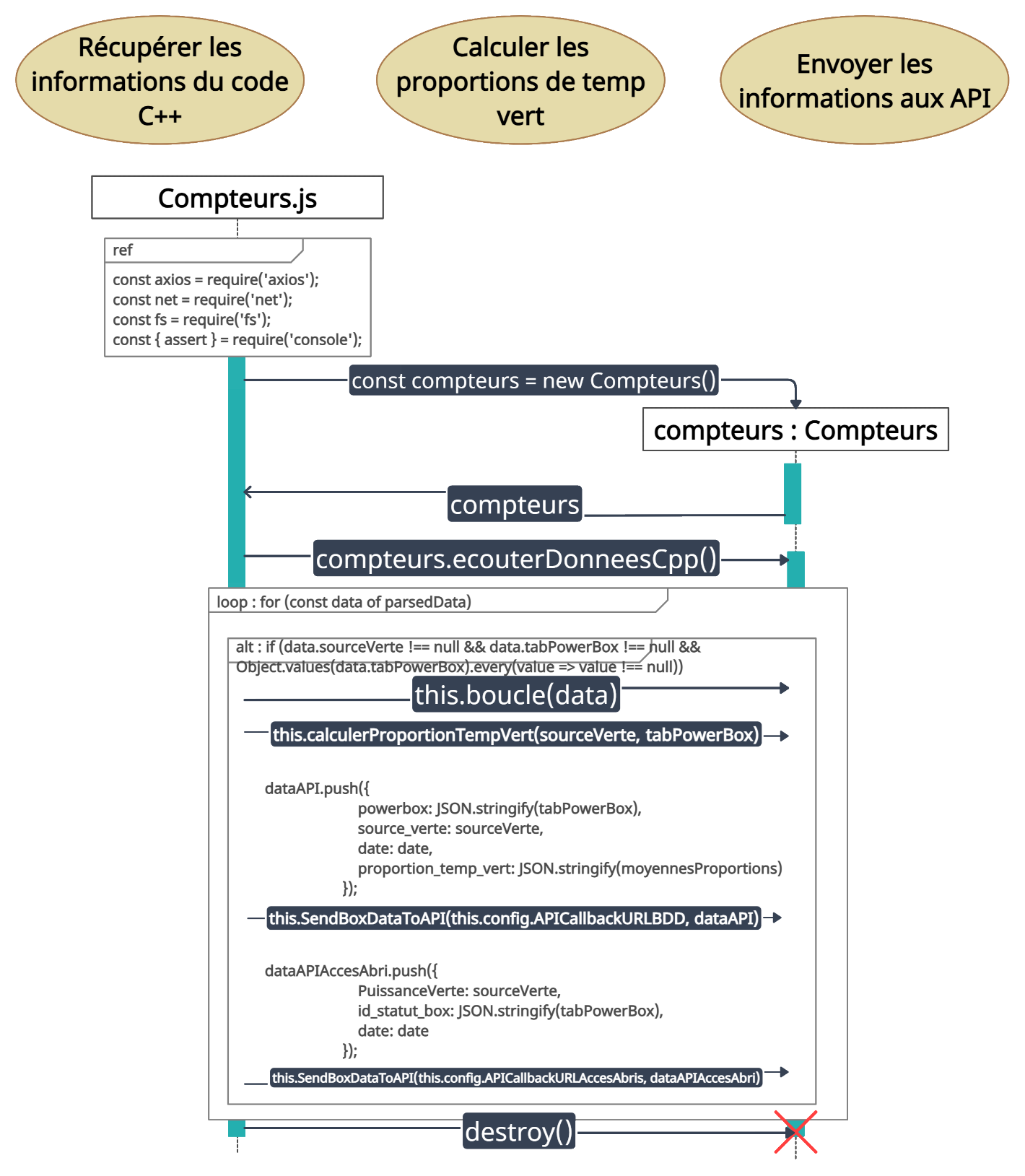
double resistance = 44.6; // Par exemple, une résistance de 44.6 ohms

};

#endif // CAPTEUR\_LUMINOSITE\_H

# Module de test 4 :

## Diagramme de séquence :



// Importation du module axios pour les requêtes HTTP

const axios = require('axios');

// Importation du module net pour les sockets avec le CPP

const net = require('net');

const fs = require('fs');

const { assert } = require('console');

const { resolve } = require('dns');

// Classe Compteurs qui simule les données d'une carte d'entrée/sortie et calcule les proportions de temps vert.

class Compteurs {

    // Constructeur de la classe Simulateur.

    constructor() {

        this.proportionsTempVertStockees = []; // Proportions de temps vert stockées

        this.sourceVerteStockees = []; // Valeurs de source verte stockées

        this.MAX\_SIZE = 10; // Taille maximale du tableau de toutes les proportion de temps vert stockées. Ajustable au choix.

        this.previousSourceVerte = null; // Valeur précédente de source verte

        this.previoustabPowerBox = null; // Valeur précédente des états des PowerBox

        this.previousDate = null // Valeur précédente de la date

        this.config = this.loadConfig(); // Configuration chargée depuis le fichier

        this.dataAPISolarPannel = []; // Données pour l'API des panneaux solaires

        this.dataAPILumi = []; // Données pour l'API de luminosité

        this.dataAPIBox = []; // Données pour l'API des PowerBox

        this.dataAPIAcces = []; // Données pour l'API d'accès

        this.MAX\_SIZE\_API = 3; // Taille maximale du tableau des données API

        this.reconnectionIntervals = {}; // Intervalles de reconnexion aux API

        // Variables pour le suivi du temps permettants de calculer les proportions

        this.firstTimer = null; // Premier temps enregistré

        this.timer = null; // Temps actuel

        this.previousTimer = null; // Temps précédent

        this.timerVert = 0; // Temps passé avec source verte active

        this.timerTotal = 0; // Temps total écoulé

        this.MoyennetempsPuissanceBox = new Array(8).fill(0); // Moyenne des temps verts pour chaque PowerBox

        this.tempsPuissanceBox = new Array(8).fill(0); // Temps VERT écoulé pour chaque PowerBox

        this.ratio = null // Valeur du ratio de temps vert

    }

Dossier de tests et de validation (extrait) :

Les tests suivants permettent de vérifier la bonne récupération des données des deux programmes C++ en amont.

|  |  |  |
| --- | --- | --- |
| Réf. : FE3.2 : Serveur Node.js / Récupérer les données des programmes C++ | | |
| **ENVIRONNEMENT DU TEST** | | |
| Le serveur Node.js sera testé sur une machine virtuelle (VM) configurée à l’avance dans un mode de fonctionnement normal. Les tests seront effectués à l’intérieur d’un programme Javascript. | | |
| **DESCRIPTION** | **CRITERES D'ACCEPTATION** | **RESULTAT** |
| Récupération des informations. | Les informations sont récupérées et sont utilisables (sans pertes ou fausses informations). |  |
| Erreur de récupération des informations | Un message d’erreur apparait pour indiquer une erreur lors de la récupération des informations et indiquer de quel programme C++ l’erreur vient. |  |
| Reformatage des informations. | Les informations sont correctement reformatées et prêtent à être utilisées. |  |
| **ETAT DU TEST** | | |
| Etat du test : Accepté  Refusé  Accepté sous Réserve | | |
| Observations : | | |