

Table des matières

Introduction	4
Contextualisation	4
Conception du Robot	5
Conception/Architecture globale	6
2.1- Architecture	6
2.1.1- Présentation de l'architecture IOT	6
2.1.2- Architecture général de notre projet	7
2.2- Conception	8
Robot	8
3.1. Présentation	9
3.2- Environnement de développement	10
3.2.1- Architecture projet PlatformIO	11
3.3- ESP principal	11
3.3.1- Modules	12
3.3.1.1- Traitement parallèle	12
3.3.1.2- Wifi	13
3.3.1.3- MQTT	15
3.3.1.4- Bluetooth	17
3.3.1.5- Modèle de données	20
3.3.1.6- Moteurs	20
3.3.1.7- Capteurs	22
3.3.2- Programme principal	22
3.4- ESP Caméra	29
3.4.1- Modules	29
3.4.1.1- Caméra	29
3.4.2- Programme principale	30
Serveur Applicatif	33
4.1- Présentation	33
4.2- Environnement de développement	33
4.3- Architecture globale	33

4.4- Modèles	35
4.4.1- Modèle de données robot	35
4.4.2- Modèle de données d'authentification	35
4.4.2.1- Modèle Database interne à l'application	35
4.4.2.2- Modèle pour l'authentification et identité	36
4.4.3- Modèle de données d'autorisation	36
4.4.4- Modèle de données de connexion	36
4.5- Configuration	37
4.6- Couche d'accès aux données (Data Access Layer)	38
4.6.1- Contexte d'accès aux données	38
4.6.2 - Services Database	39
4.7- Services (Business Layer)	40
4.7.1- Authentification	41
4.7.2.1- Génération du token	41
4.7.2.2- Décodage du token	43
4.7.2- Autorisation	43
4.7.3- Cache de connexion et données	44
4.8- Service d'échange de données temps réel	46
4.9- API	50
4.10- Services hébergés	51
4.10.1- Service MQTT	51
4.10.2- Service TCP	55
4.11- Injection de données en base	57
Application mobile	59
5.1- Présentation	59
5.2- Environnement de développement	59
Contrôle du robot	60
6.1- Présentation	60
6.2- Application WPF (manette/kinect)	60
6.2.1- Environnement de développement	60
6.2.2- Architecture	61
6.2.3- Service MQTT Client	62

6.2.4- Présentation du dashboard	63
6.2.5- Contrôle par manette	64
6.2.6- Contrôle par Kinect	67
6.3- Contrôle Open CV	70
Interface Web de visualisation	76
7.1- Présentation	76
7.2- Environnement de développement	76
7.3- Dashboard principal	77
7.4- Visualisation d'un équipement	78
Sécurité	82
8.1- SSL/TLS pour HTTPS	82
8.2- Identification, Authentification et autorisation	84
Déploiement de l'application	86
9.1- Docker	86
9.1.1- Docker Serveur applicatif	87
9.1.2- Docker Angular	88
9.2- Docker Compose	89
9.3- Installation du VPS	91
9.4- Déploiement Docker-compose	92
Bilan	93
Résultats	93
Difficultés rencontrées	93
Evolutions	93
Conclusion	94
Bibliographie	95
Annexes	96

Introduction

Au début du second semestre du master 1 MIAGE, tous les élèves sont amenés à réaliser un projet libre dans le domaine de l'IOT.

Tout d'abord, nous avons eu en lien avec cette matière 6 heures de cours et 12 heures de TD d'introduction pour nous guider et nous donner une trame afin de commencer notre projet.

Les heures de travail n'étaient pas comprises dans l'emploi du temps, il a fallu concevoir ce projet en dehors de nos heures de cours et d'alternance sur une durée d'environ 3 mois.

Cela nous permet d'acquérir de nouvelles compétences et de nous perfectionner dans plusieurs domaines tels que la programmation, la connectivité, d'apprendre de nouveaux langages et découvrir de nouveaux logiciels.

Contextualisation

Le projet sur lequel nous avons travaillé a été créé par notre groupe de 6 personnes et a pour objectif d'être réutilisé par les futur élèves présent en licence 3 MIAGE.

Notre projet consiste à créer un petit robot et de déployer un grand nombre de fonctionnalités contrôlés ou autonomes.

Afin de mener à terme ce projet, nous avons pu trouver ou acheter du matériel comme la manette et kinect version 1 XBOX, deux batteries externes, boitier 9V en complément la faculté nous a mis à disposition un robot à monter, avec 2 ESP, une caméra, plaques, cables,...

1- Conception du Robot

Le matériel dont nous disposions au départ pour le robot était le suivant :

- ESP32 x6
- LED x6
- Photo sensor x6 (3V)
- Temperature sensor x6 (5V)
- Caméra ESP32 x1
- Kinect v1 x1
- Manette x1
- Joy car de joy it composé de :
 - Speed sensor x2
 - Ultrasonic sensor (4,5 - 5,5V)
 - *Line Tracking sensor x3*
 - Obstacle sensor x2 (3,3V - 5V)
 - Servo Motor x2 (3,3V - 5V)
 - Motor x2
 - Bridge Motor x1 à ajouter (6V et 12V)
 - Battery case (où on peut insérer des piles dont la capacité totale d'alimentation est de 6V)

Ici, vous trouverez un fichier de conception pour les besoins des différents sensors et outils :

[https://docs.google.com/presentation/d/1OnFTx-b8yxkE_prlXsskJKvHg2HNMI9ueH_oP87oO0/edit?
usp=sharing](https://docs.google.com/presentation/d/1OnFTx-b8yxkE_prlXsskJKvHg2HNMI9ueH_oP87oO0/edit?usp=sharing)

Cependant, nous avons dû prendre du matériel supplémentaire, car lors de l'étude de chacun des sensors et outils, nous nous sommes rendu compte de plusieurs problèmes :

- Source de courant insuffisante

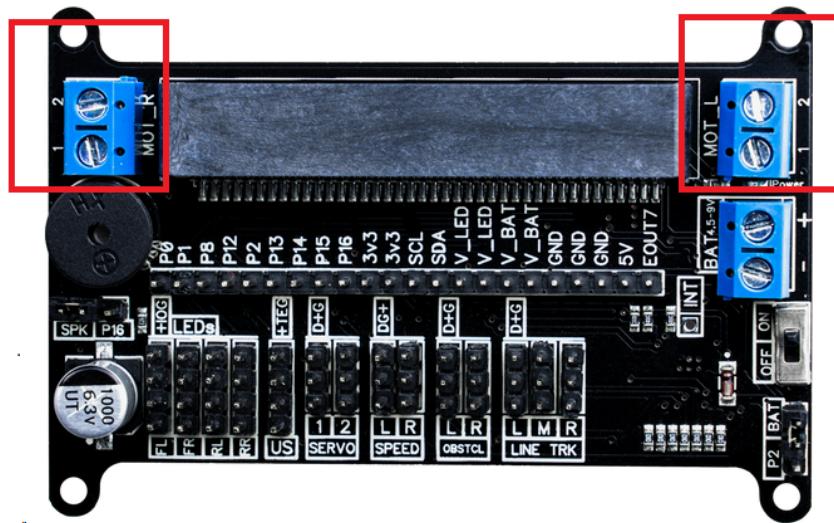
La source de courant était un problème, car nous disposions d'une boîte, permettant d'y mettre des piles qui génèrent au max **6 volts** (toutes cumulées). Hors, les moteurs ont besoin **de 3V chacun**, cela veut dire que cette alimentation ne pourrait faire fonctionner que les moteurs seuls, mais certainement pas l'ESP 32 CAM avec (qui était un élément important de notre réalisation).

Nous avons donc choisi d'utiliser en plus **une batterie externe** (personnelle), qui permettra d'alimenter l'ESP 32, et de laisser les piles pour les moteurs. Malgré tous ces changements, lors des tests, un des moteurs semblait moins puissant que l'autre. Nous avons donc compris qu'il y avait de la perte d'énergie - ce qui est en fait logique - et que nous n'avions finalement pas la quantité minimum pour chaque moteur, mais un peu moins (car nous avions 3v par moteur, et avec la perte d'énergie, nous avions en fait moins que cela).

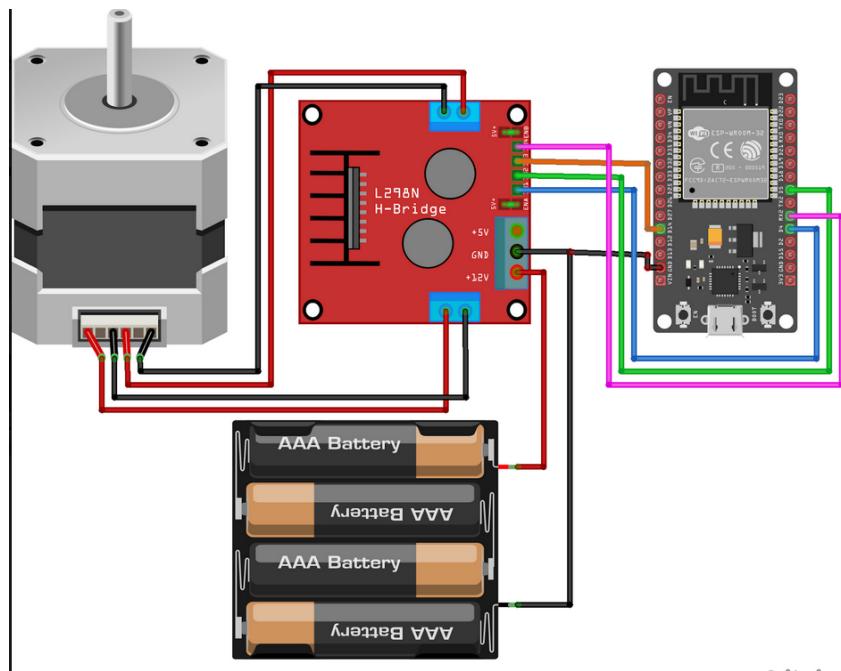
Pour résoudre ce problème, nous avons donc acheté un boîtier et une pile de 9V. Cela a changé considérablement la donne, notre robot est bien plus puissant, et cette puissance est parfaitement équilibrée.

- Bridge pour les moteurs

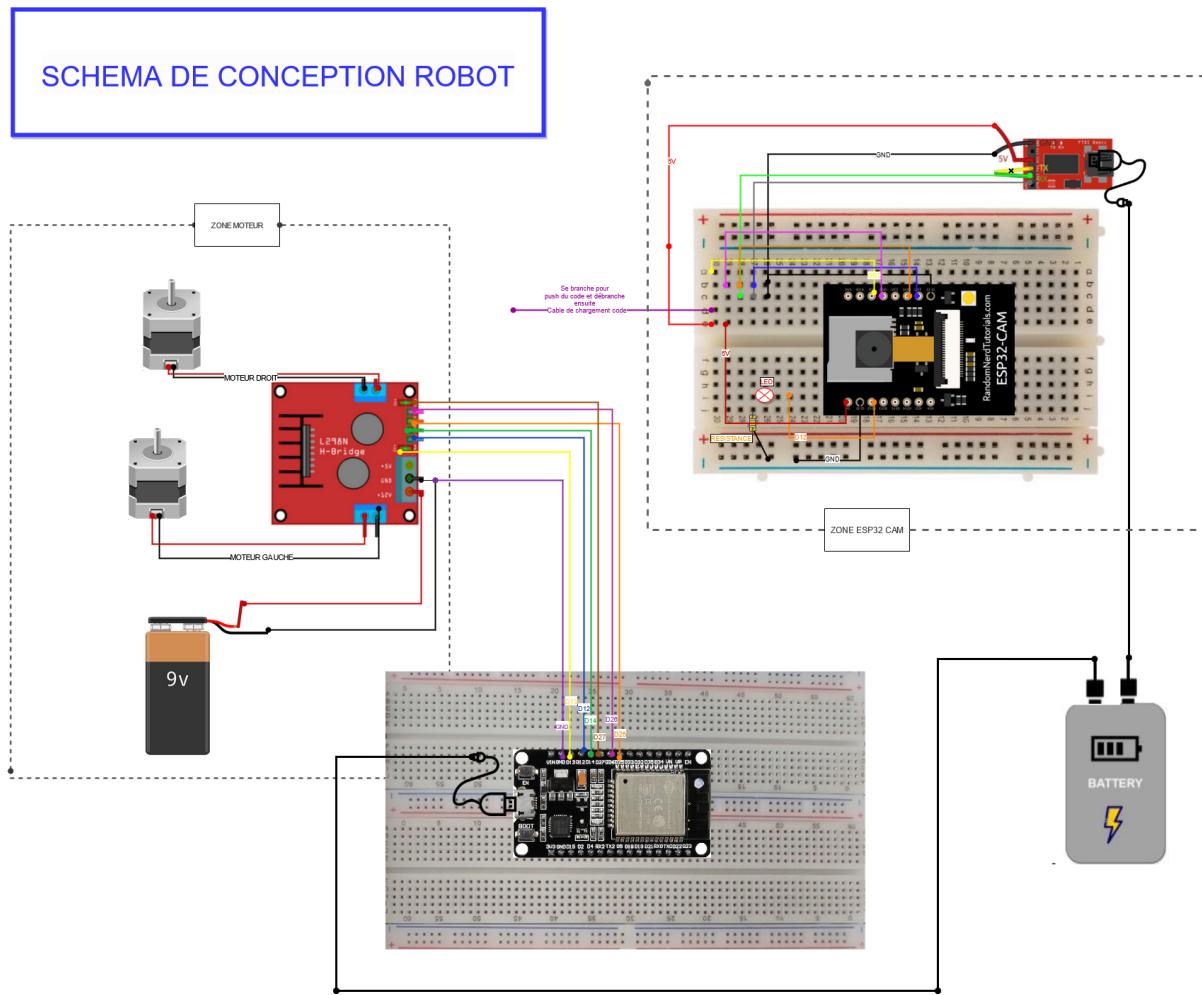
Lors des recherches sur l'assemblage des moteurs, nous avons réalisé que sans la carte mère fournie par "Joy Car" (the mainboard), nous allions avoir un problème pour contrôler les moteurs. Sur la carte mère de base, il y a des zones et des pins exclusivement réservé à la gestion et l'alimentation des moteurs (ici en rouge la liaison des moteurs sur la carte mère de base) :



Cependant, sur notre ESP 32, il n'y a pas cette possibilité. C'est pourquoi nous devons acheter un bridge, afin de faire le lien entre les moteurs et l'ESP 32. Nous devrions obtenir quelque chose comme ça pour contrôler les moteurs :



La conception du robot a donc été modifié plusieurs fois, mais voici la version final du montage :

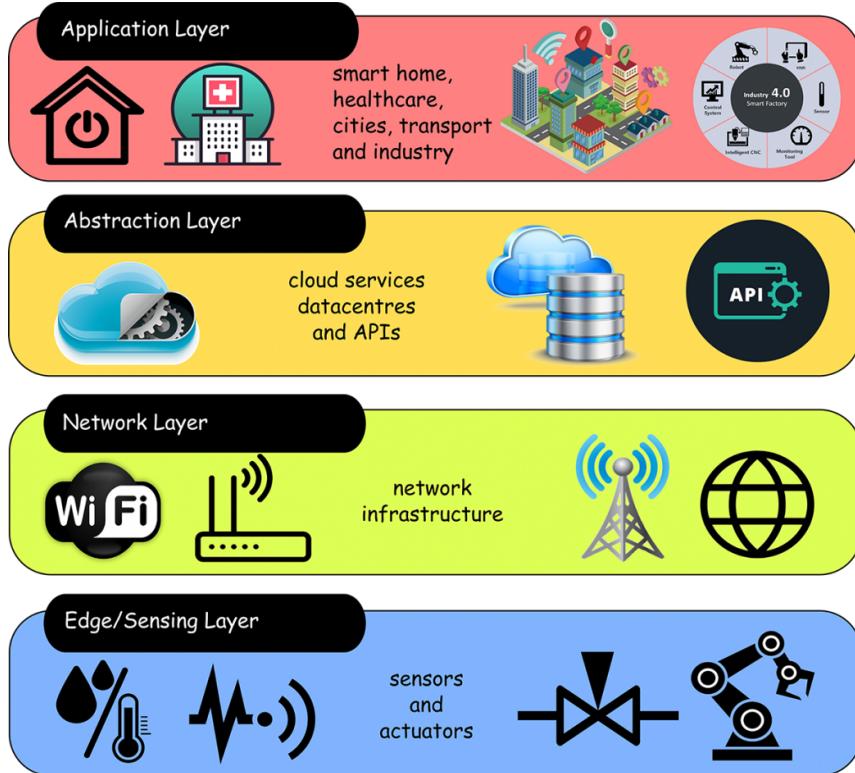


2- Conception/Architecture globale

2.1- Architecture

2.1.1- Présentation de l'architecture IoT

Toute application IoT peut être décomposée en (plus ou moins) quatre grandes couches, comme le montre la figure ci-dessous.



Couche des capteurs

Il s'agit de la couche la plus basse de l'architecture IOT, qui se compose de réseaux de capteurs, systèmes embarqués, étiquettes et lecteurs RFID ou d'autres capteurs souples qui sont des formes différentes de capteurs déployés sur le terrain.

Passerelle d'accès et couche réseau

Cette couche est responsable du transfert des informations recueillies par les capteurs à la couche suivante. Elle doit prendre en charge un protocole universel évolutif, flexible et normalisé pour le transfert de données à partir de dispositifs hétérogènes (différents types de nœuds de capteurs). Cette couche doit disposer d'un réseau performant et un réseau robuste. Elle doit également permettre à plusieurs organisations de communiquer indépendamment.

Couche de service de gestion

Cette couche sert d'interface entre la couche réseau et la couche application, en mode bidirectionnel. Elle est responsable de la gestion des dispositifs et de la gestion de l'information et responsable de la capture d'une grande quantité de données brutes et de l'extraction d'informations pertinentes des données stockées ainsi que des données en temps réel. La sécurité et la confidentialité des données doivent être assurées.

Couche d'application

Il s'agit de la couche la plus élevée de l'IOT qui fournit une interface utilisateur pour accéder à diverses applications à différents utilisateurs. Une interface utilisateur pour accéder à diverses applications pour différents utilisateurs.

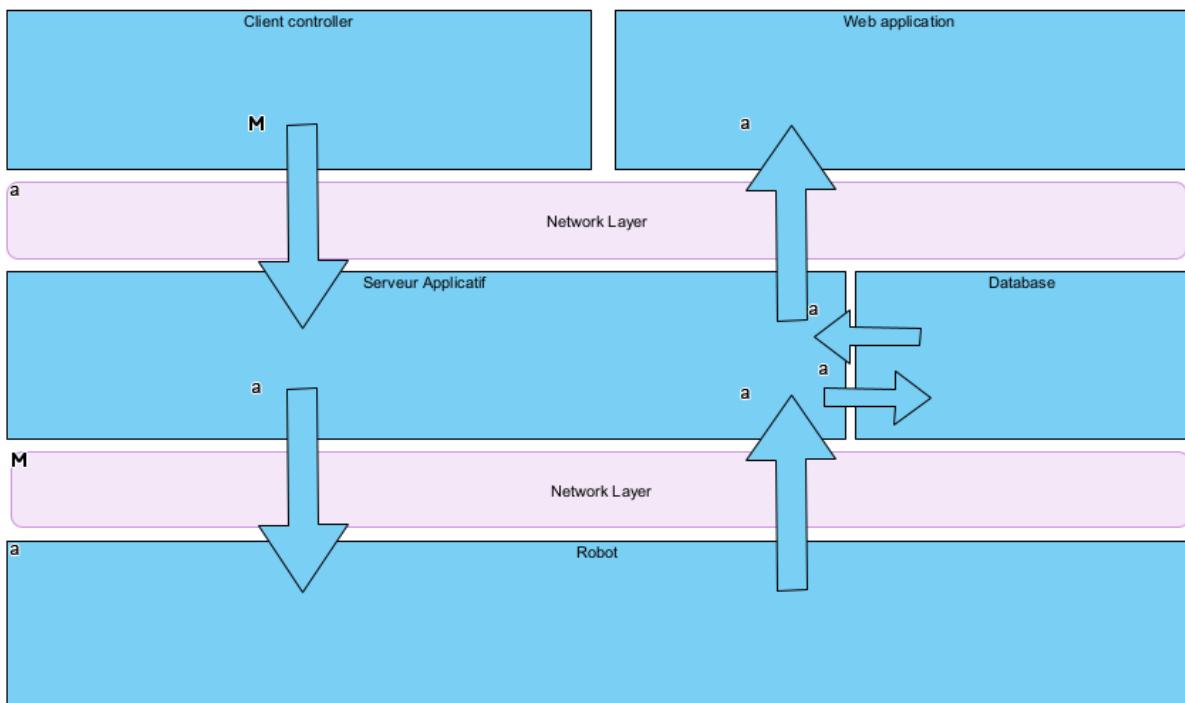
Les applications peuvent être utilisées dans divers secteurs tels que le transport, les soins de santé, l'agriculture, la chaîne d'approvisionnement, le gouvernement, le commerce de détail, etc.

2.1.2- Architecture général de notre projet

Nous aurons donc notre robot (couche des capteurs) qui utilisera la couche réseau pour pouvoir communiquer et interagir avec le Cloud.

Quand on dit Cloud, c'est en fait la couche d'abstraction, la couche contenant le middleware et donc notre serveur applicatif permettant de recevoir et fournir des données à la couche basse via la couche réseau.

Enfin, notre application web de visualisation (située dans la couche des applications) utilisera le middleware pour permettre une visualisation des données. Notre application de contrôle des robots se situe aussi dans cette couche et utilise le middleware pour envoyer les données nécessaire au contrôle du robot.



Nos controller enverrons des données à notre middleware en utilisant la couche réseau via un protocole de transmission de données. Le middleware enverra ensuite ces données par la couche réseau avec un protocole de transmission de données au robot.

Quant à lui, le robot enverra différentes données capteurs au middleware, il pourra les transformer ou non, les faire persister en base de données et aussi les fournir à notre application web pour une visualisation des données robot.

Nous utiliserons comme principalement dans la couche network les protocoles de transmissions de données MQTT et TCP.

2.2- Conception

A partir de cette architecture, nous avons déterminé tous les flux entre les différentes couches. Et nous avons en plus associé des fonctionnalités à l'entrée et sortie des flux de chaque application détaillée ci-dessous avec un diagramme de cas d'utilisation.

Pour un souci de lisibilité, le diagramme est accessible en cliquant sur le lien suivant :

<https://cdn.discordapp.com/attachments/330782151702478849/852526624679526450/unknown.png>

3- Robot

3.1. Présentation

Le robot doit être doté de plusieurs fonctionnalités. Pour commencer il pourra être utilisé comme un scanner pour sonder son environnement, c'est-à-dire récupérer la luminosité, la température à une position qui peut nous permettre, par la suite, de faire une cartographie qui montrent l'exposition à la lumière et à la chaleur d'une certaine zone.

Nous pouvons alors faire le parallèle par exemple avec l'envoi d'un robot pour effectuer une cartographie du niveau de radiation dans Tchernobyl.

Pour pouvoir sonder une zone, il est alors nécessaire que l'on puisse contrôler le robot à distance ou en présentiel dans des zones non critiques et dangereuses. Il est alors obligatoire d'avoir une caméra embarquée pour voir son environnement dans le cas du contrôle à distance.

A partir de ce constat le robot doit pouvoir réaliser plusieurs fonctionnalités :

- A partir de capteurs de luminosité et température, le robot est capable de renvoyer les données de n'importe quelle zone. Ces valeurs sont consultables sur une interface web.
- Grâce à la "kinect" ainsi qu'à une manette de jeu, le robot peut contrôler à distance les déplacements du robot par des gestes ou par les joysticks.

- Le robot détecte une personne grâce à la reconnaissance d'image.
- Le robot est localisable via une balise GPS (cela permet aussi d'obtenir la localisation des données récupérées par les capteurs).
- Visualisation en temps réel via la caméra embarquée du robot.

Pour pouvoir réaliser cela le robot à besoin de différents composants matériels :

- Microcontrôleur
- Deux moteurs pour les déplacements avec roue libre
- Capteurs de luminosité et de température
- Balise GPS
- Caméra embarqué

Nous devons alors doté le robot de microcontrôleur ESP32 lui permettant de réaliser des tâches. Nous avons fait le choix d'utiliser deux ESP pour différentes raisons.

Pour pouvoir obtenir des images du robot, nous devons utiliser un ESP particulier appelé ESP32-CAM. C'est une version dotée d'une caméra Wifi. Il permet de créer des projets de caméra IP pour le streaming vidéo avec différentes résolutions.

Cependant cette carte est limitée en broche, et ne permet pas de placer tous les autres composants que l'on à besoin sur le robot, c'est-à-dire 2 moteurs, 2 capteurs (luminosité, température). **Nous n'aurons pas d'autre choix que d'utiliser un deuxième ESP traditionnel pour les autres fonctionnalités.**

Enfin, nous n'avons pas de baliseur GPS, nous avons fait le choix de connecter l'ESP à un téléphone portable (ayant le rôle de balise GPS) afin qu'il puisse partager ses informations GPS. Nous placerons le téléphone portable sur le robot.

Notre robot est alors composé au total de deux ESP :

- ESP principale ayant pour rôle :
 - Pilotage des moteurs : Il reçoit des données permettant la rotation des moteurs depuis le Cloud venant d'une application.
 - Partage des données capteurs : Il envoi sur le cloud les valeurs des différents capteurs qui le composent à instant T.
 - Il utilise une connexion WiFi et le protocole de transfert de données MQTT pour l'échange de données.
- ESP Cam ayant pour rôle :
 - Envoi des images de la caméra vers le Cloud.
 - Il utilise une connexion WiFi et le protocole de transfert de données TCP pour l'échange de données.

Nous avons conçu au total deux applications, une pour chaque ESP. Les applications sont développées grâce à l'IDE PlatformIO dans le langage C++.

3.2- Environnement de développement

La philosophie unique de PlatformIO sur le marché de l'embarqué offre aux développeurs un environnement de développement intégré moderne (Cloud & Desktop IDE) qui fonctionne sur plusieurs plates-formes, prend en charge de nombreux kits de développement logiciel (SDK) ou Frameworks différents, et comprend un débogage sophistiqué, des tests unitaires, analyse de code automatisée (analyse de code statique) et gestion à distance (développement à distance). Il est conçu pour maximiser la flexibilité et le choix des développeurs, qui peuvent utiliser des éditeurs graphiques ou en ligne de commande (PlatformIO Core CLI), ou les deux.

PlatformIO est un outil incontournable pour les ingénieurs de systèmes embarqués professionnels qui développent des solutions sur plusieurs plates-formes spécifiques. De plus, en ayant une architecture décentralisée, PlatformIO offre aux développeurs nouveaux et existants une voie d'intégration rapide pour développer des produits prêts pour le commerce, et réduit le temps global de mise sur le marché.

Il fonctionne sur n'importe lequel des systèmes d'exploitation modernes (macOS, MS Windows, Linux, FreeBSD).

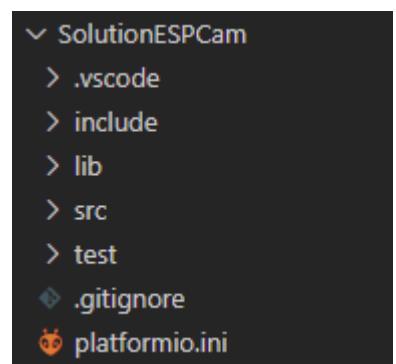
Il est compatible avec 25 Frameworks tel que Arduino ou ESP-IDF et plus de 1000 modèles de cartes.

Nous allons détailler l'architecture d'un projet PlatformIO.

3.2.1- Architecture projet PlatformIO

Pour commencer, le projet est composé d'un fichier "platformio.ini", c'est le fichier de configuration du projet. Nous pouvons configurer plusieurs environnements ciblant différents boards et frameworks. Mais aussi des arguments d'entrées pour l'application, et spécifier l'utilisation de différentes librairies.

Ensuite nous avons le dossier "src" contenant l'entrée de l'application, c'est-à-dire le programme principal. Nous pouvons écrire d'autres fichiers à utiliser dans le programme principal dans le dossier "lib", mais encore des librairies télécharger. Enfin, toutes les librairies spécifier dans le fichier de configuration sont placés dans le dossier "include".



3.3- ESP principal

Pour l'ESP principal du robot nous utilisons la plateforme "expressif32" ainsi que deux frameworks, Arduino et ESPIDF pour profiter des avantages de chacun. Cela est configuré dans le fichier de configuration de PlatformIO « platformio.ini ».

L'objectif de cet ESP est de permettre la réception de commande moteur à distance ainsi que de partager en temps réel les données des capteurs du Robot tel que la luminosité, la température à une position donnée à un instant T.

Pour permettre au robot d'interagir avec le monde extérieur, nous utilisons WiFi comme protocole de communication réseau, ainsi que MQTT comme protocole de transmission de données. Il pourra ainsi partager ses données mais aussi recevoir des commandes pour ses déplacements à distance.

Pour faciliter la localisation du robot, nous utilisons comme balise GPS un téléphone portable qui partage ses coordonnées GPS au robot, via une application en utilisant Bluetooth comme protocole de communication.

3.3.1- Modules

Le programme est divisé en plusieurs modules utilisables par le programme principal pour mieux diviser la responsabilité des différentes tâches qu'il doit réaliser. Nous avons donc les modules suivants :

- Gestion parallèle des tâches
- Gestion de la connexion WiFi
- Gestion MQTT
- Gestion Bluetooth
- Gestion des moteurs
- Gestion des capteurs
- Représentation des données utilisés

3.3.1.1- Traitement parallèle

La plupart des systèmes d'exploitation semblent autoriser l'exécution simultanée de plusieurs programmes ou threads. C'est ce qu'on appelle le multitâche. En réalité, chaque cœur de processeur ne peut exécuter qu'un seul programme à un moment donné. Une partie du système d'exploitation appelée le planificateur est chargée de décider quel programme

exécuter et à quel moment, et fournit l'illusion d'une exécution simultanée en passant rapidement d'un programme à l'autre.

Le planificateur dans un système d'exploitation en temps réel (RTOS) est conçu pour fournir un modèle d'exécution prévisible (normalement décrit comme déterministe). Ceci est particulièrement intéressant pour les systèmes embarqués, comme les appareils Arduino, car les systèmes embarqués ont souvent des exigences en temps réel.

Les **ordonnanceurs traditionnels** en temps réel, tels que l'ordonnanceur utilisé dans FreeRTOS, atteignent le déterminisme en permettant à l'utilisateur d'attribuer **une priorité à chaque thread d'exécution**. Le planificateur utilise ensuite la priorité pour savoir quel thread d'exécution exécuter ensuite. **Dans FreeRTOS, un thread d'exécution est appelé Task.**



L'ESP est composé de 2 Coeurs. Il est alors possible de répartir certaines tâches entre les deux avec une certaine priorité. Nous avons donc créé une abstraction supplémentaire comme module pour nous permettre d'écrire des tâches qui s'exécutent simultanément.

Toutefois, si plusieurs tâches sont exécutées sur le même cœur, alors le niveau de priorité de la tâche déterminera quelle tâche aura la priorité d'exécution.

Cela nous est principalement utile pour obtenir plusieurs tâches comme par exemple :

- Tâche ayant pour responsabilité la gestion WiFi
- Tâche ayant pour responsabilité la gestion MQTT

Nous avons donc créé **une classe « TaskRobot »** ayant pour rôle d'exécuter du code avant le lancement de la tâche, pendant celle-ci et avant la fin de la tâche.

Il permet de créer une tâche assignée à un cœur en particulier de l'ESP, avec un certain nom et **une certaine stackDepth**. Il permet de démarrer la tâche et de l'arrêter.

Cela permet à d'autres classes d'hériter de « TaskRobot » pour écrire un traitement qui s'exécutera de façon parallèle.

3.3.1.2- Wifi

L'ESP doit disposer d'un protocole de communication afin de lui permettre d'interagir avec d'autres applications dans le Cloud. Nous avons choisi **WiFi** car c'est le protocole le plus facile à mettre en œuvre avec peu de moyen, il suffit simplement de disposer d'un point d'accès WiFi connecter à un routeur (principalement une box internet qui joue ce rôle).

Nous avons donc écrit un module centralisant toute la gestion de la connectivité WiFi du programme qui utilise lui-même **la librairie « WiFi » d'Arduino**.

Sachant que le robot peut se déplacer sur certaines distances, nous avons choisi d'utiliser aussi la librairie « WiFiMulti » pour permettre au robot de trouver un nouveau point d'accès le plus proche. S'il perd la connexion il pourra ainsi se **reconnecter à un AP** (point d'accès) le plus proche afin qu'il soit toujours accessible sur le Cloud.

Nous avons donc créé une classe « WifiESP » qui hérite de « TaskRobot » permettant d'exécuter le code de façon parallèle :

Il suffira de lui donner un tableau d'Identifiants Wifi composé d'un couple SSID et password, pour lister tous les points d'accès auxquels il pourra essayer de se connecter.

Nous pouvons aussi spécifier deux rappels lors de la réussite de la connexion ou lors d'une déconnexion (OnConnected et OnDisconnected). Cela permet au programme utilisant cette classe de réagir lors des deux événements.

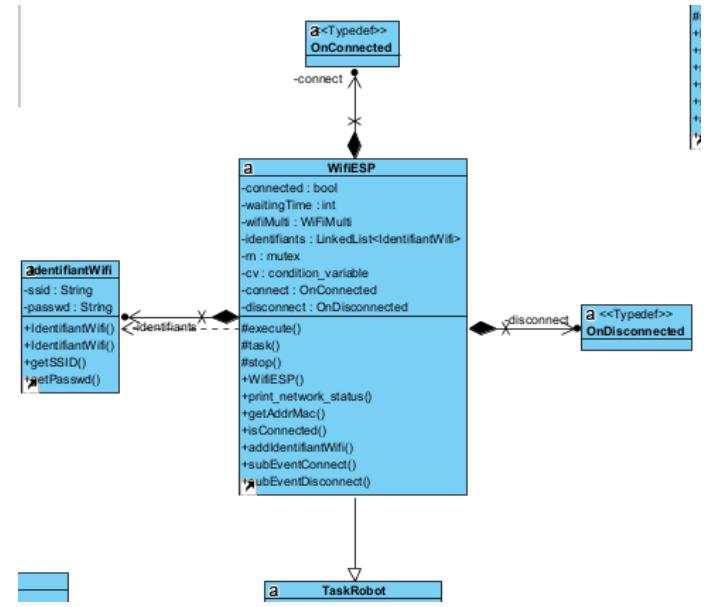
Au démarrage de la tâche, il ajoute au wifi multi AP toute la liste des identifiants et s'abonne aux différents événements de la librairie « WiFi » d'Arduino :

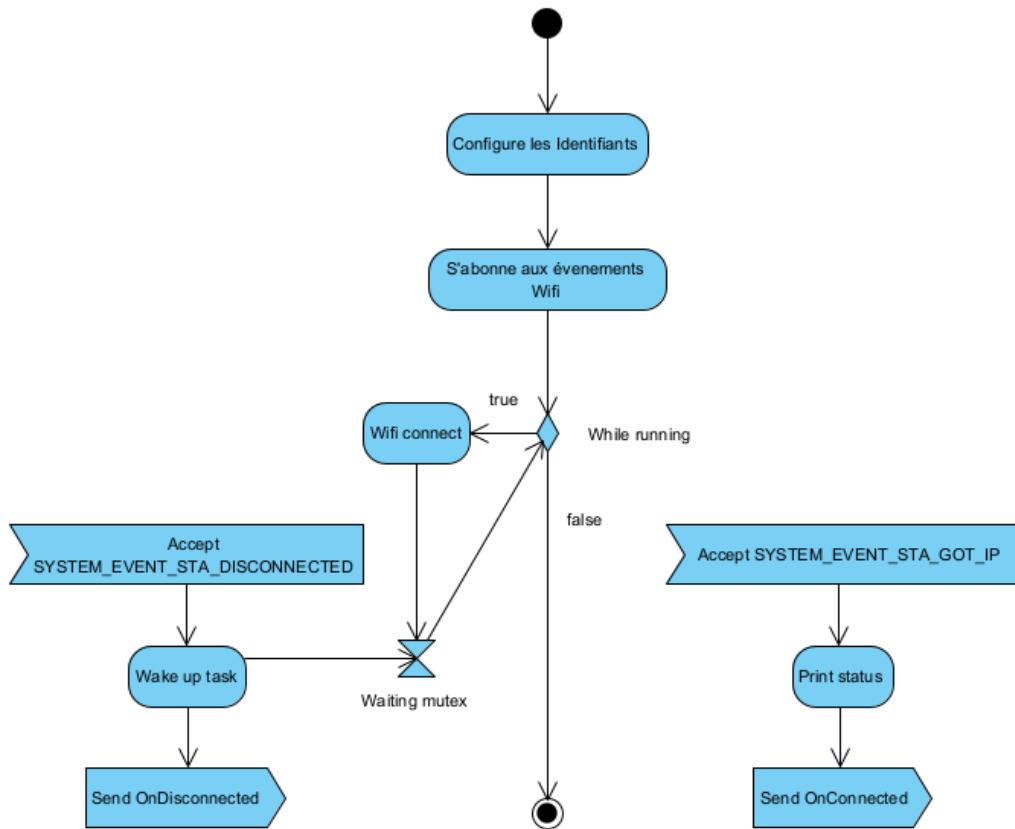
- SYSTEM_EVENT_STA_DISCONNECTED
- SYSTEM_EVENT_STA_GOT_IP

La tâche principale aura pour rôle d'essayer de se connecter à un point d'accès listé au-dessus, puis s'endort.

- A la réception de l'événement « SYSTEM_EVENT_STA_GOT_IP », il affichera le statut de la connexion et appellera le callback « OnConnected » auquel le programme principal s'est abonné.
- A la réception de l'événement « SYSTEM_EVENT_STA_DISCONNECTED », il réveillera la tâche principale grâce à l'utilisation de mutex et condition_variable, qui essaiera de nouveau à se connecter à un nouveau point d'accès pour s'endormir après.

Si on décide d'arrêter la tâche alors on déconnecte le Wifi et on réveille la boucle principale pour qu'elle s'arrête.

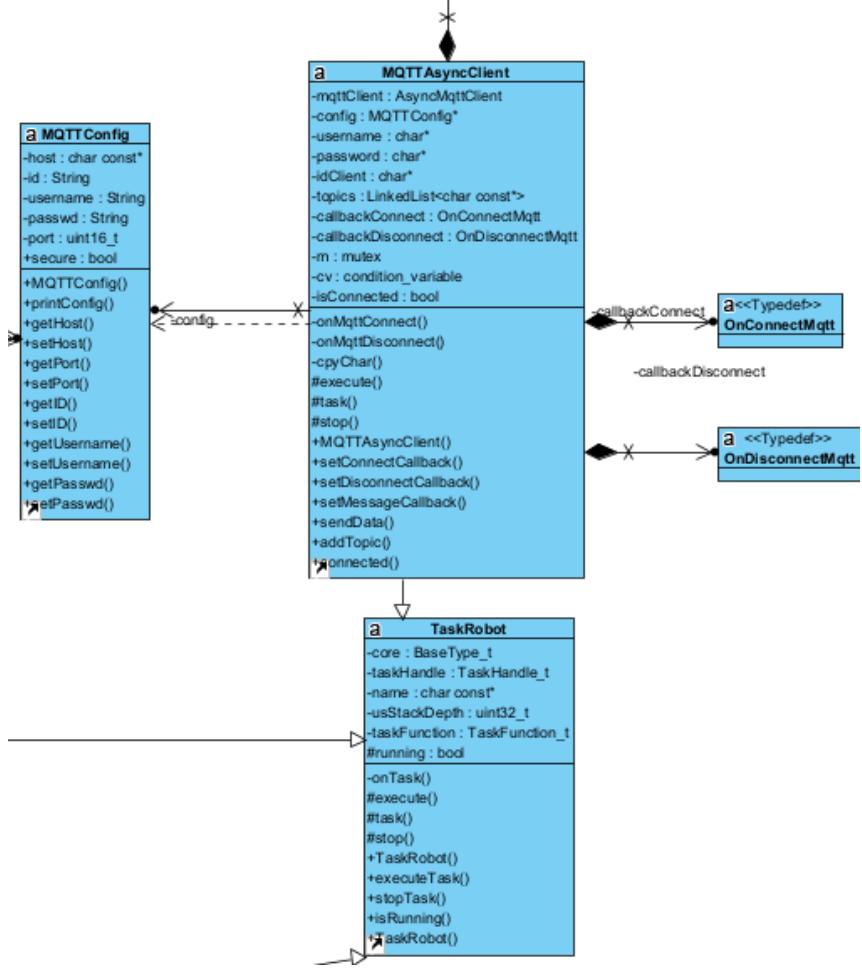




3.3.1.3- MQTT

L'ESP doit pouvoir envoyer ou recevoir des données venant du Cloud via la connexion WiFi. Nous utilisons alors le **protocole d'échange de données MQTT basé sur un système d'abonnement et de publication sur des topics**.

Comme pour le WiFi nous avons donc créé un module qui gère l'ensemble de l'interaction avec un broker MQTT. Il implémente « TaskRobot » pour lui permettre une exécution parallèle. Ce module utilise la librairie « **AsyncMqttClient** ».



Configuration

Nous avons un objet « **MQTTConfig** » qui contient toute la configuration utilisée pour pouvoir se connecter au broker MQTT.

Elle contient le nom d'hôte, le port du broker ainsi que l'identifiant client, un nom d'utilisateur et un mot de passe.

Dans notre cas l'identifiant client sera l'adresse MAC de la carte WiFi de l'ESP. Pour pouvoir s'authentifier auprès du broker, nous utiliserons un token d'authentification que nous ajouterons au champ nom d'utilisateur lors de tentative de connexion. Concernant l'authentification, nous verrons cela plus tard dans la partie du serveur applicatif.

Tâche MOTT

La classe de gestion MQTT s'appelle « **MQTTAsyncClient** ». C'est un pont entre la librairie « **AsyncMqttClient** » et notre programme principal.

La librairie à un fonctionnement entièrement asynchrone. Nous pouvons réagir avec elle avec plusieurs événements :

- A la réussite de la connexion avec le broker
- A la déconnexion du broker
- A la réception d'un message venant du broker

Nous nous abonnons alors à tous les événements à l'instanciation de notre objet « **MQTTAsyncClient** ».

Avant le lancement de la tâche le programme principal peut s'abonner à deux événements :

- **OnConnectMqtt** : pour que le programme principal soit averti quand la connexion a réussi.
- **OnDisconnectMqtt** : pour que le programme principal soit averti quand il y a déconnexion du broker MQTT.
- **OnMessageMqtt** : pour que le programme principal puisse traiter le nouveau message reçu venant du broker MQTT.

Cela permet au programme principal de réagir en fonction des événements principaux MQTT de façon asynchrone.

Il va par ailleurs, avant le lancement de la tâche, pouvoir ajouter des topics à une liste de l'objet « **MQTTAsyncClient** » pour qu'il puisse s'abonner aux différents topics du broker en cas de réussite de la connexion.

Au lancement de la tâche, grâce à la classe de configuration fournit au constructeur de « **MQTTAsyncClient** » par le programme principal, il défini le nom d'hôte et le port du broker MQTT. Mais aussi l'identifiant du client et le token d'authentification dans le champ nom d'utilisateur. Enfin nous désirons une nouvelle session propre ainsi qu'un temps d'expiration de la connexion de 30 secondes.

Une fois la configuration établie, nous lançons la boucle d'exécution. La boucle tente une connexion au broker MQTT, puis elle s'endort avec un mécanisme de synchronisation.

- En cas de réussite de connexion, l'objet s'abonne aux différents topics ajouter dans sa liste, puis appelle le callback « **OnConnectMqtt** » du programme principal afin qu'il puisse réagir après la réussite de la connexion.
- En cas de déconnexion, ou de non connexion, il appelle le callback « **OnDisconnectMqtt** » du programme principal afin qu'il puisse réagir à l'événement qui vient de se produire. Puis nous débloquons la boucle principale de la tâche pour tenter une reconnexion au broker MQTT.

3.3.1.4- Bluetooth

Comme nous l'avons dit précédemment, notre ESP32 n'est pas doté d'une balise GPS. Nous avons fait le choix de récupérer les coordonnées GPS d'un téléphone via le protocole Bluetooth.

Pour cela, l'ESP agit comme serveur Bluetooth afin que le téléphone puisse se connecter et écrire les coordonnées GPS. Pour cela nous avons écrit un module Bluetooth utilisant le Bluetooth Low Energy.

Bluetooth Low Energy, BLE en abrégé, est une variante de Bluetooth à économie d'énergie. L'application principale du BLE est la transmission à courte distance de petites quantités de données (faible bande passante). Contrairement au Bluetooth qui est toujours activé, le BLE reste en mode veille en permanence, sauf lorsqu'une connexion est initiée.

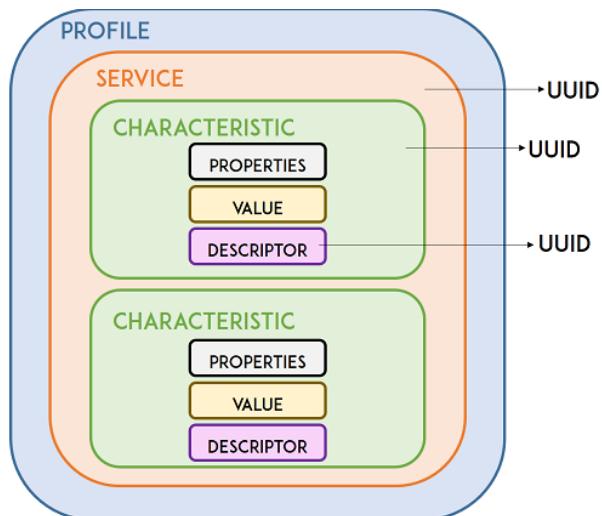
Cela fait qu'il consomme très peu d'énergie. BLE consomme environ 100 fois moins d'énergie que Bluetooth (selon le cas d'utilisation).

Avec Bluetooth Low Energy, il existe deux types d'appareils : le serveur et le client. L'ESP32 peut agir soit en tant que client, soit en tant que serveur.

Le serveur annonce son existence, afin qu'il puisse être trouvé par d'autres appareils, et contient les données que le client peut lire. Le client analyse les appareils à proximité et lorsqu'il trouve le serveur qu'il recherche, il établit une connexion et écoute les données entrantes. C'est ce qu'on appelle la communication point à point.

GATT

GATT signifie Attributs génériques et définit une structure de données hiérarchique qui est exposée aux appareils BLE connectés. Cela signifie que GATT définit la manière dont deux appareils BLE envoient et reçoivent des messages standard. Comprendre cette hiérarchie est important, car cela facilitera la compréhension de l'utilisation du BLE et de l'écriture de vos applications.



Le niveau supérieur de la hiérarchie est un profil, qui est composé d'un ou plusieurs services. Généralement, un appareil BLE contient plusieurs services.

Chaque service contient au moins une caractéristique, ou peut également référencer d'autres services. Un service est simplement une collection d'informations, comme les relevés de capteurs, par exemple.

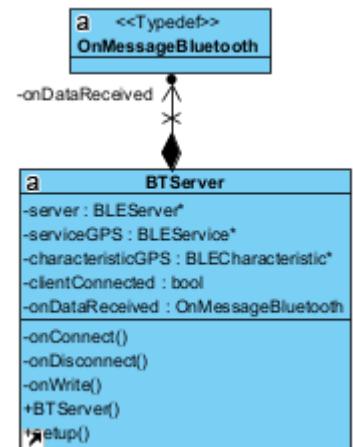
La caractéristique appartient toujours à un service et c'est là que les données réelles sont contenues dans la hiérarchie (valeur). La caractéristique a toujours deux attributs : la déclaration de caractéristique (qui fournit des métadonnées sur les données) et la valeur de caractéristique.

De plus, la valeur caractéristique peut être suivie de descripteurs, qui élargissent davantage les métadonnées contenues dans la déclaration de caractéristique.

Serveur Bluetooth ESP

Nous allons donc créer un service ainsi qu'une caractéristique sur le serveur Bluetooth nous permettant de pouvoir recevoir les coordonnées GPS du téléphone dans un nouveau module. Il prendra en paramètre une fonction de callback du programme principale pour réagir à la réception d'un nouveau message en Bluetooth du téléphone.

Dans l'initialisation, nous commençons à définir des UUID, un pour le service et un autre pour la caractéristique.



```

BLEDevice::init("ESP32-BLE");

this->server = BLEDevice::createServer();

this->server->setCallbacks(this);

this->serviceGPS = this->server->createService(SERVICE_UUID);

```

Premièrement nous créons un appareil BLE appelé ESP32-BLE. Puis nous le définissons en tant que serveur. Nous créons par la suite un service avec l'UUID précédemment créé. Enfin nous associons notre classe comme classe gérant les callbacks des événements serveur.

```

this->characteristicGPS = this->serviceGPS->createCharacteristic(
    CHARACTERISTIC_UUID,
    BLECharacteristic::PROPERTY_WRITE
);

this->characteristicGPS->setCallbacks(this);

```

Ensute, nous définissons la caractéristique pour ce service. Comme vous pouvez le voir, nous utilisons également l'UUID défini précédemment, et nous devons passer en arguments les propriétés de la caractéristique. Dans ce cas, c'est : ÉCRIRE.

Puis nous disons que notre classe réagit aux événements de la caractéristique créer en la définissant comme callback.

```

this->serviceGPS->start();

// BLEAdvertising *pAdvertising = pServer->getAdvertising(); // this still is working for backward compatibility
BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_UUID);
pAdvertising->setScanResponse(true);
pAdvertising->setMinPreferred(0x06); // functions that help with iPhone connections issue
pAdvertising->setMinPreferred(0x12);
BLEDevice::startAdvertising();

```

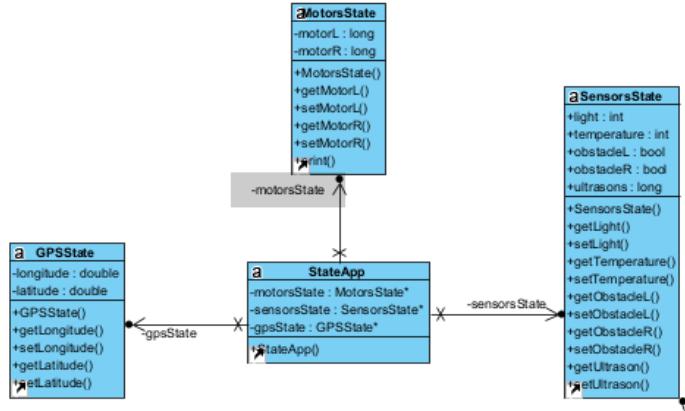
Enfin, vous pouvez démarrer le service et la publicité, afin que d'autres appareils BLE puissent analyser et trouver cet appareil BLE.

Callback caractéristique

Quand un client écrit alors sur la caractéristique, un événement d'écriture est trigger dans notre classe. Nous récupérons alors les données pour rediriger vers un callback que notre programme principale à définir au préalable pour pouvoir traiter les données GPS envoyées par le téléphone.

[3.3.1.5- Modèle de données](#)

Nous avons défini dans notre application, plusieurs objets permettant de faire persister les données actuelles (ou l'état de notre robot) des capteurs, des moteurs ainsi que des coordonnées GPS.



L'état de notre application est alors composé d'un état moteur, contenant la puissance des moteurs gauche et droit, d'un état contenant principalement les valeurs de lumière et températures des capteurs, et enfin un état contenant la latitude et la longitude du robot.

Les états seront définis et utilisés pour envoyer les valeurs du robot au broker MQTT, ou recevoir et définir la puissance des moteurs du robot.

3.3.1.6- Moteurs

Le gestionnaire moteur du robot a pour rôle de définir la puissance des moteurs gauche et droit du robot.

Au démarrage, il configure les différents PIN de sorties de l'ESP vers les moteurs :

- IN1 et IN2 : broche GPIO du moteur gauche
- IN3 et IN4 : broche GPIO du moteur droit
- ENA : broche PWM du moteur gauche
- ENB : broche PWM du moteur droit

MotorHandler
-IN1 : int
-IN2 : int
-ENA : int
-IN3 : int
-IN4 : int
-ENB : int
-dualMotor : bool
-mapVitesse()
+MotorHandler()
+MotorHandler()
+init()
+rotateMotor()
+rotateMotors()
+breakMotor()
+breakMotors()

Pour contrôler la direction du robot nous avons la table de vérité suivante :

DIRECTION	INPUT 1	INPUT 2	INPUT 3	INPUT 4
Forward	0	1	0	1
Backward	1	0	1	0
Right	0	1	0	0
Left	0	0	0	1
Stop	0	0	0	0

Cependant cette table de vérité est plutôt utilisée pour contrôler la direction à courant continu. Pour une meilleure précision des déplacements, nous allons utiliser un signal de modulation de largeur d'impulsion (PWM). Notre signal PWM prendra en entrée une valeur sur la plage [200 ;255] (résolution sur 8 bits).

SIGNAL ON THE ENABLE PIN	MOTOR STATE
HIGH	Motor enabled
LOW	Motor not enabled
PWM	Motor enabled: speed proportional to duty cycle

Afin de faire effectuer des rotations au moteur, notre objet dispose d'une méthode « rotateMotor », il prend en paramètre le numéro du moteur, ainsi qu'une valeur de puissance de moteur [-100 ;100].

Premièrement la méthode déterminera le sens de rotation du moteur, par exemple pour le moteur gauche :

- Valeur > 0 : IN1 = 0 et IN2 = 1
- Valeur < 0 : IN1 = 1 et IN2 = 0
- Valeur = 0 : IN1 = 0 et IN2 = 0

```
typedef enum {
    MOTOR1 = 1,
    MOTOR2 = 2
} motor_num_t;
```

Une fois le sens de rotation du moteur défini, nous allons ramener la puissance moteur de la plage [-100 ;100] dans la plage [200 ;255]. Nous récupérons alors la valeur absolue de la puissance en entrée pour mapper la plage [-100 ;100] dans la plage [0 ;100].

Ayant une valeur entre 0 et 100 nous pouvons trouver une nouvelle valeur entre 200 et 255 via l'opération suivante :

- $(\text{valeur} - \text{IN_MIN}) * (\text{OUT_MAX} - \text{OUT_MIN}) / (\text{IN_MAX} - \text{IN_MIN}) + \text{OUT_MIN}$

Enfin, une fois le sens et la puissance du signal PWM défini pour un moteur, nous écrivons les valeurs sur les broches pour définir la rotation du moteur.

[3.3.1.7- Capteurs](#)

Le gestionnaire des capteurs a pour rôle la récupération des données des différents capteurs du robot. Pour l'instant nous avons uniquement un capteur de luminosité, et un capteur de température. Mais il peut aussi gérer un capteur à ultrason, et des capteurs d'obstacles (que nous n'utilisons pas actuellement).

L'objet « SensorHandler » étant notre gestionnaire, il a une référence directe sur l'objet représentant l'état des capteurs que nous avons présenté plus tôt.

Il suffit pour récupérer les valeurs et les définir dans l'état, de l'initialiser avec le numéro des broches GPIO, puis d'appeler la méthode « `readValues()` ». Elle définit les valeurs actuelles dans l'objet state des capteurs. Il suffira ensuite de récupérer les valeurs dans l'objet « `SensorState` ».

Pour pouvoir récupérer la température nous utilisons la librairie « `OneWire` », ainsi que la librairie « `DallasTemperature` ».

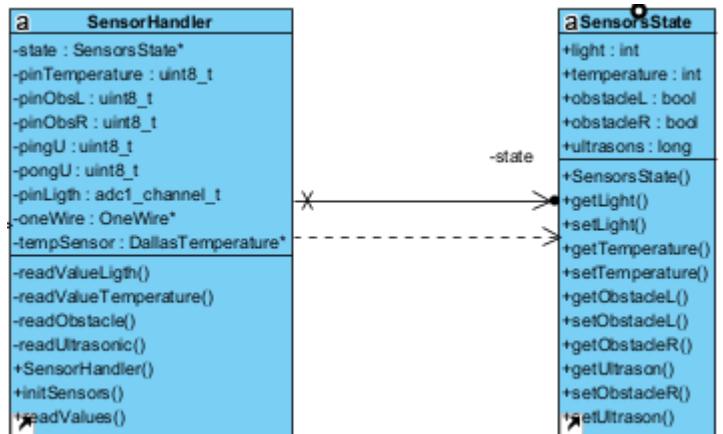
[3.3.2- Programme principal](#)

Le programme principal utilise toutes les librairies au-dessus ainsi que les objets états du robot.

Le rôle du programme est de se connecter au réseau WiFi, puis au broker MQTT pour recevoir les commandes moteurs (c'est-à-dire un objet contenant la puissance du moteur gauche et droit), mais aussi pour envoyer les données des capteurs et GPS.

Configuration

Le programme nécessite une certaine configuration à définir au préalable par l'utilisateur.



Pour commencer il est nécessaire de définir l'identifiant de l'équipement pour l'identification auprès du serveur.

```
#pragma region IdDefinition
const String idEquipment = "Esp32Robot";
#pragma endregion IdDefinition
```

Puis de définir en dur le JSON utilisé pour l'authentification.

```
{ "IdEquipment": "Esp32Robot", "Password": "25Zjqgr8AQxxZsyZ", "TypeEquipment": "Robot", "Role": "Station"}
```

De définir les SSID et password pour la connexion Wifi.

```
IdentifierWifi ids[] = {
    IdentifierWifi("Bbox-4DD70ADE", "551F54E2D72A27CA1EA44567F149E1"),
    IdentifierWifi("iPhone de Steven", "iobwgn7obkf91")
};
```

Nous définissons les données nécessaires pour la connexion MQTT ainsi que les topics d'envoi et de réception MQTT, ainsi que si nous utilisons une connexion chiffré ou non.

```
const char* HostMqtt = "62.35.150.64";
const char* DataTopicName = "IOT/Data";
const String ControlerTopicNameSub = "IOT/Controler/" + idEquipment;
```

```
#if ASYNC_TCP_SSL_ENABLED
bool mqttSecure = true;
const uint16_t PortMqtt = 8884;
#else
bool mqttSecure = false;
const uint16_t PortMqtt = 1883;
#endif
```

Enfin nous définissons le endpoint de l'API pour l'authentification.

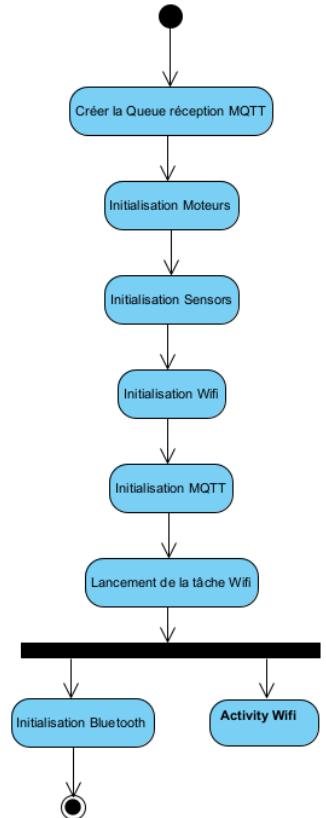
```
const String urlAuth = "http://62.35.150.64:8000/api/AuthEquipment/auth";
```

Setup

Pour commencer, le programme exécute la fonction “setup”. Elle a pour rôle l'initialisation de nos librairies précédentes et de lancer la tâche Wifi du programme. Nous créons aussi un tâche itérant sur une file d'attente pour le traitement des messages reçus du broker MQTT, nous verrons plus en détail le traitement plus tard.

L'initialisation :

- Moteurs et des capteurs pour attacher les différents PIN aux différentes broches de l'ESP.
- Wifi pour ajouter les identifiants multi AP ainsi que s'abonné aux différents events.
- MQTT pour définir les abonnements aux différents topics et s'abonner aux events.
- Bluetooth pour lancer le service et s'abonner à la réception de données.

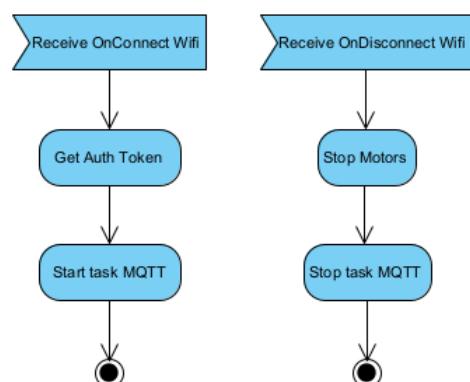


Événements Wifi

Le programme principale s'est abonné à deux événements lorsque la connexion Wifi réussie ou a échoué/déconnexion.

Lorsque la connexion réussie, il déclenche le début de la tentative de connexion MQTT. La tentative de connexion est la demande au serveur applicatif d'un jeton d'authentification, puis d'une tentative de connexion au broker avec ce même jeton.

Lorsque la connexion échoue ou se déconnecte, il arrête les moteurs et arrête la tâche MQTT.



Obtention du jeton d'authentification

Pour obtenir le jeton, il est nécessaire que l'ESP effectue une requête HTTP au serveur applicatif avec ses informations d'authentification.

Nous récupérons le timestamp actuel, puis le comparons au timestamp du temps d'expiration du token (si c'est la première fois il est égale à 0). Si le temps d'expiration est inférieur au timestamp actuel nous essayons alors d'obtenir un token d'authentification valide qui nous permettra de nous connecter au broker MQTT.

Nous demandons une authentification pour l'ESP du type d'équipement Robot ayant le rôle de "Station" du robot.

Si la requête retourne un code 200, alors le token à bien été généré, nous pouvons par la suite déserialiser le résultat en JSON pour récupérer le jeton d'authentification et le définir dans le champ username de l'objet de configuration MQTT pour ensuite faire une tentative de connexion au broker.

Ce token nous permettra de nous identifier lors de la connexion par le broker MQTT, puis nous permettra d'obtenir les autorisations nécessaires pour s'abonner aux topics ou nous avons les droits ainsi qu'envoyer des messages sur les topics où nous avons les droits aussi.

Concernant le fonctionnement du token, nous verrons cela dans la partie du serveur applicatif.

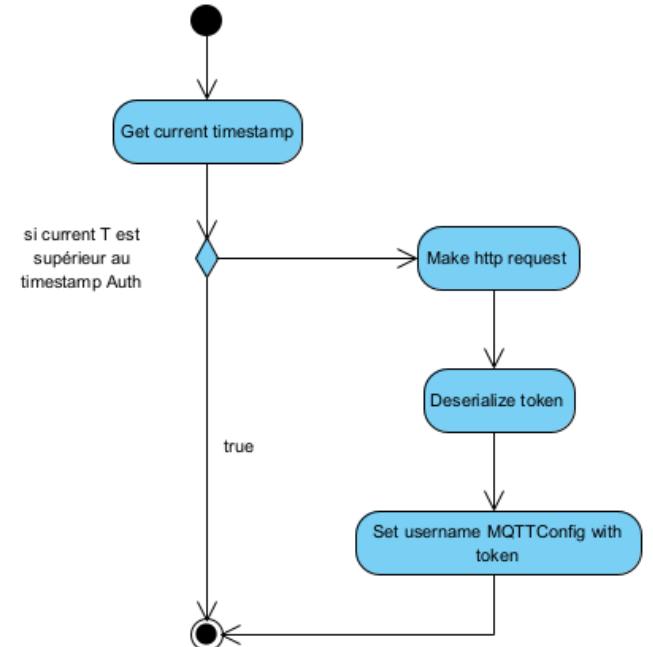
Tentative de connexion MQTT

Une fois l'obtention du token d'authentification, nous lançons la tâche MQTT.

Elle essaie ensuite de se connecter au broker avec la configuration que nous lui avons transmis par référence dans son constructeur.

Lorsque nous recevons l'événement de réussite de la connexion, nous ne faisons rien de particulier, nous attendons simplement un message à recevoir du broker.

Dans le cas de la réception de l'événement de déconnexion, nous arrêtons les moteurs et nous réessayons une tentative de connexion au broker MQTT, tout en vérifiant si le token est toujours valide ou pas pour l'authentification lors de la connexion.



Envoy des données capteurs au broker MQTT

L'envoi des données des capteurs se fait périodiquement dans la fonction "loop" de l'ESP.

Toutes les X secondes nous demandons au gestionnaire des capteurs de prendre les mesures des capteurs. Ils stockent les dernières mesures dans l'objet contenant l'état des capteurs. Ensuite nous sérialisons les dernières données (luminosité, température, géolocalisation) avec notre ID d'équipement au format JSON.

Dans le cas où nous sommes connectés au broker MQTT, nous lui envoyons le JSON précédemment obtenu. Il se chargera par la suite de stocker les données en base de données et d'avertir toutes les autres applications de la disponibilité de ces nouvelles données.

Réception des données moteurs du broker MQTT

A l'initialisation du client MQTT, nous lui avons défini une fonction de callback à déclencher lorsqu'il reçoit un nouveau message, avec le nom du topic ainsi que le contenu du message.

Pour pouvoir traiter cette réception nous utilisons un système de queue (concurrente) de façon parallèle. Précédemment dans l'initialisation nous avons dit que nous avions créé une queue de réception des messages MQTT.

Les files d'attente sont la principale forme de communication entre les tâches. Ils peuvent être utilisés pour envoyer des messages entre les tâches et entre les interruptions et les tâches. Dans la plupart des cas, ils sont utilisés comme des tampons FIFO (First In First Out) thread-safe avec de nouvelles données envoyées à l'arrière de la file d'attente, bien que les données puissent également être envoyées à l'avant. Nous utilisons alors les files d'attente implémentées dans FreeRTOS.

Les fonctions API de file d'attente permettent de spécifier un temps de blocage.

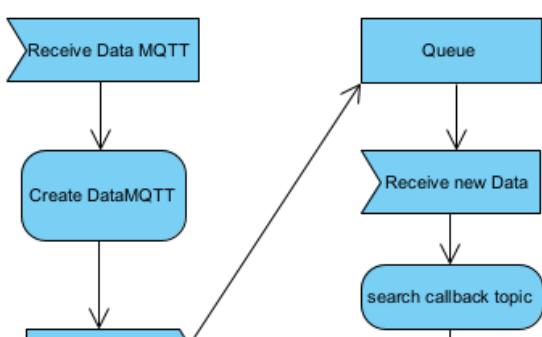
Lorsqu'une tâche tente de lire à partir d'une file d'attente vide, la tâche est placée dans l'état Bloqué (elle ne consomme donc pas de temps CPU et d'autres tâches peuvent s'exécuter) jusqu'à ce que les données soient disponibles dans la file d'attente ou que le temps de blocage expire.

Lorsqu'une tâche tente d'écrire dans une file d'attente complète, la tâche est placée dans l'état Bloqué (elle ne consomme donc pas de temps CPU et d'autres tâches peuvent s'exécuter) jusqu'à ce que de l'espace soit disponible dans la file d'attente ou que le temps de blocage expire.

Lors de la réception d'un nouveau message sur un topic, nous le définissons dans une structure de données "DataMQTT", contenant le topic source, le message, et la taille du message.

Puis nous ajoutons à la fin de la file d'attente la nouvelle structure de données obtenue.

```
typedef struct DataMQTT {
    char * topic;
    byte * message;
    unsigned int length;
} DataMQTT_t;
```



Une fois ajoutée, cela débloque la tâche itérant sur la file d'attente pour traiter les nouvelles entrées. La

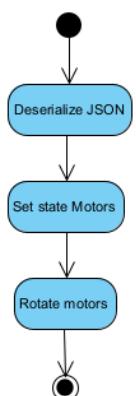
tâche récupère alors la structure de données à la tête de la file d'attente et itère sur une liste pour trouver la fonction de callback associée au topic, puis déclenche le callback avec le message reçu en paramètre.

```
void mqtt_pubcallback_async(char* topic, char* payload, AsyncMqttClientMessageProperties_t properties, void* userContext)
{
    size_t lenTopic = strlen(topic)+1;
    DataMQTT_t * data = reinterpret_cast<DataMQTT_t*>(pvPortMalloc(sizeof(DataMQTT_t)));
    data->length = len;
    data->topic = (char*)pvPortMalloc(lenTopic*sizeof(char));
    data->message = (byte*)pvPortMalloc(len*sizeof(byte));
    memcpy(data->message,payload,len);
    strcpy(data->topic,topic);
    xQueueSend(queue, (void*) &data, portMAX_DELAY);
}
```

```
void taskHandlePubCallback(void * args){
    DataMQTT_t *data;
    while(true){
        BaseType_t t = xQueueReceive(queue, &data, portMAX_DELAY);
        if(t == pdPASS)
        {
            for(int i = 0; i < actions.size(); i++){
                if(actions.get(i).getTopic() == data->topic){
                    actions.get(i).getHandler()((const byte*) data->message,data->length);
                    break;
                }
            }
            vPortFree(data);
        }
        vTaskDelete(NULL);
    }
}
```

Dans le cas de la réception des données moteurs sur le topic associé, nous avons associé le topic “IOT/Controller” à une fonction qui traite les messages destinés au contrôle des moteurs.

Elle tente une désérialisation des données sous le format JSON, puis extrait la puissance des moteurs pour définir les valeurs dans l'objet d'état des moteurs. Enfin elle fait appel à la fonction “rotateMotor” du gestionnaire des moteurs pour contrôler un certain moteur avec la valeur contenu dans l'objet d'état des moteurs. Le gestionnaire définit par la suite le sens de rotation des moteurs et la puissance pour contrôler le déplacement du robot.



Traitemennt des données GPS reçus en bluetooth

Dans l'initialisation du programme principale, nous nous sommes abonnés à la réception de messages bluetooth pour les coordonnées GPS.

A la réception nous sommes avertis par le déclenchement d'une fonction avec le message en paramètre. Nous effectuons une tentative de désérialisation du format JSON, et si elle réussit nous actualisons les valeurs de l'objet GPS. Il seront envoyés par la suite lors de la tâche d'envoi des données au broker MQTT.

```
/*
 * Callback when received data from Bluetooth client.
 *
 * @param message message received
 */
void receivedActionBluetooth(String *message){

    StaticJsonDocument<256> doc;
    DeserializationError error = deserializeJson(doc, message->c_str());

    if(error != DeserializationError::Ok)
        return;

    if(doc.containsKey(LatitudeJson) && doc.containsKey(LongitudeJson)){
        state.gpsState->setLongitude(doc[LongitudeJson]);
        state.gpsState->setLatitude(doc[LatitudeJson]);
    }
}
```

3.4- ESP Caméra

Pour l'ESP principale du robot nous utilisons la plateforme “expressif32” ainsi que le framework Arduino.

L'objectif de cet ESP est de permettre l'envoi d'images prises par la caméra au serveur applicatif afin que les utilisateurs puissent voir l'environnement du robot en temps réel.

Pour permettre au robot d'interagir avec le monde extérieur, nous utilisons WiFi comme protocole de communication réseau, ainsi que TCP comme protocole de transmission de données.

Nous avons utilisé TCP car au début nous utilisions une librairie MQTT qui limite la taille du payload. Cependant, pendant le projet nous avons changé de librairie MQTT qui ne limitait plus la taille, mais nous n'avons pas eu le temps d'effectuer des tests avec la nouvelle librairie.

3.4.1- Modules

Tout comme l'ESP principal, l'ESP Caméra utilise des modules. Cependant, cet ESP n'a pas beaucoup de tâches à exécuter. Il est donc principalement composés de deux modules :

- Le module de traitement parallèle présenté plus haut pour l'ESP principal
- Un module de récupération de l'image et traitement

Les modules de traitement parallèle et Wifi étant déjà présentés dans la partie du premier ESP, nous n'allons pas les présenter dans cette partie.

3.4.1.1- Caméra

Le rôle du module de la Caméra sera de configurer l'image que nous souhaitons obtenir en sortie, ainsi que d'exécuter une tâche nous permettant d'obtenir une image sous forme de bytes.

Configuration

Premièrement nous définissons dans la configuration que nous souhaitons une image au format JPEG avec une résolution de 480x320. Nous avons choisi une résolution basse par souci de consommation de bande passante ainsi que pour privilégier le nombre de frames par seconde. Plus l'image est grande, plus l'ESP mettra du temps à effectuer le traitement. Nous définissons toutes les configurations dans la méthode init de notre module.

Tâche

La tâche du module de caméra est d'obtenir l'image dans un buffer, puis de l'ajouter dans une queue FreeRTOS que nous avons vu précédemment dans la partie de l'ESP principale.

Pour cela nous avons créé une structure de données “DataVideo” contenant la taille de l'image ainsi que son buffer. Cette structure de données est ensuite rajoutée à la queue de la file d'attente qui sera traité par la suite par le programme principale.

```

typedef struct DataVideo {
    size_t size;
    const byte *buffer;
} DataVideo_t ;

```

```

void CameraStream::sendData(camera_fb_t * fb){
    DataVideo_t * data = reinterpret_cast<DataVideo*>(pvPortMalloc(sizeof(struct DataVideo)));
    data->size = fb->len;
    data->buffer = (const byte *)fb->buf;
    xQueueSend(queue, (void*) &data, portMAX_DELAY);
}

```

Nous avons choisi d'utiliser une file d'attente car lors de l'envoi d'un message il y a un temps d'envoi. Nous perdons alors des frames par seconde. Nous avons donc mis en place une file d'attente pour push l'image dans la file d'attente puis reboucler directement derrière afin de lire l'image suivante.

En parallèle l'autre tâche s'occupera de récupérer l'image à la tête de la file d'attente et de l'envoyer au serveur TCP. Ce qui n'entraîne plus de latence pour la lecture des images.

3.4.2- Programme principale

Le programme principal utilise toutes les librairies au-dessus.

Le rôle du programme est de se connecter au réseau WiFi, puis au serveur TCP afin d'envoyer les images de la caméra.

Il est à noter que nous avons mis en place un protocole simple de niveau applicatif pour l'échange de données en TCP. Il est nécessaire que l'ESP pour toutes données qu'il souhaite envoyer, d'envoyer au préalable la taille de cette données en byte (cela sera détaillé dans la partie du serveur applicatif).

Configuration

```

WifiESP wifi;
const IdentifiantWifi ids[] = {
    IdentifiantWifi("Bbox-4DD70ADE", "551F54E2D72A27CA1EA44567F149E1"),
    IdentifiantWifi("iPhone de Steven", "iobwgn7obkf91")
};
const char* SSID = "Bbox-4DD70ADE";
const char* SSID_PASSWORD = "551F54E2D72A27CA1EA44567F149E1";

String token = "";
long tokenExpiration = 0;
const uint16_t portTCP = 11000;
const char* urlTCP = "62.35.150.64";
const String urlAuth = "http://62.35.150.64:8000/api/AuthEquipment/auth";
const String jsonAuth = "{ \"IdEquipment\": \"Esp32Robot\", \"Password\": \"25Zjqgr8AQxxZsyz\", \"TypeEquipment\": \"Robot\", \"Role\": \"Camera\" }";

```

Setup

Au démarrage du programme on appelle la configuration du module ci dessus afin de définir le format de l'image en sortie que l'on souhaite obtenir. Nous créons ensuite une tâche afin de traiter les objets “DataVideo” dans la file d’attente. Puis nous nous connections en Wifi.

Une fois la connexion Wifi établie, comme pour l’ESP principale, nous devons demander un token d’authentification au serveur pour pouvoir se connecter au service TCP. Une fois le token obtenu nous tentons une connexion au serveur TCP.

La phase d’authentification auprès du serveur TCP consiste à envoyer en premier la taille du token obtenu, puis d’envoyer le token. Si l’authentification réussit alors nous pouvons transmettre les images sinon le serveur nous déconnecte automatiquement.

Gestion et envoie des images

Nous pouvons alors démarrer la tâche de stream, qui aura pour rôle d’ajouter à la file d’attente tous les buffers d’images obtenus.

En dépilant les images, nous vérifions ensuite si nous sommes connectés au serveur TCP. Si c’est bien le cas alors nous préparons le message à transmettre au serveur.

Les quatre premier bytes correspondent à la taille de l’image que nous voulons transmettre, nous allouons donc un buffer de la taille de l’image plus quatre bytes. Nous copions ensuite la taille dans le buffer, puis les données de l’image à la suite. Une fois les données finales obtenues nous écrivons le buffer dans le flux TCP.

Nous libérons ensuite les ressources en mémoire.

```
void taskQueue(void * args){

    DataVideo_t *data;

    while(true){

        BaseType_t t = xQueueReceive(stream.queue, &data, portMAX_DELAY);
        if(t == pdPASS)
        {
            if(client.connected()){
                size_t len = data->size;
                byte *message = (byte*) pvPortMalloc(4*sizeof(byte)+len*sizeof(byte));
                memcpy(message, &len, 4);
                memcpy(message + 4, data->buffer, len);
                client.write(message, len + 4);
                free(message);
            } else {
                ESP.restart();
            }
            //Serial.println(t);
            vPortFree(data);
        }
        vTaskDelete(NULL);
    }
}
```

4- Serveur Applicatif

4.1- Présentation

Le serveur d'application est le cœur du système. Il fait le lien entre toutes les autres applications. Il est utilisé par le site web, l'application de contrôle des robots, ainsi que pour tous les robots. Il est lui-même connecté à une base de données, dans notre cas MongoDB.

4.2- Environnement de développement

Comme environnement de développement pour le serveur applicatif, nous utilisons Visual Studio 2019. Le serveur est une application ASP.NET qui cible le Framework .NET 5 de Microsoft. Ce Framework nous permet de créer des applications web, client lourd, ou encore des applications mobiles.

ASP.NET Core est une infrastructure multiplateforme, à hautes performances et Open source pour la création d'applications modernes, basées sur le Cloud et connectées à Internet.

Avec ASP.NET Core, vous pouvez :

- Créez des applications et services Web, des applications d'Internet des objets (IOT) et des serveurs principaux mobiles.
- Utiliser vos outils de développement préférés sur Windows, macOS et Linux.
- Déployer dans le cloud ou localement.
- Exécutez sur .net Core.

ASP .NET Core s'intègre parfaitement avec des Framework et librairies côté client comme Angular, React.

4.3- Architecture globale

Pour une meilleure maintenabilité et responsabilité des composants du serveur applicatif, nous allons mettre en place une architecture 3-tiers. Dans ce type d'architecture nous divisons l'application en différentes couches principalement : Présentation, Métier et Données, nous permettant de mettre en place une division des responsabilités.

Présentation

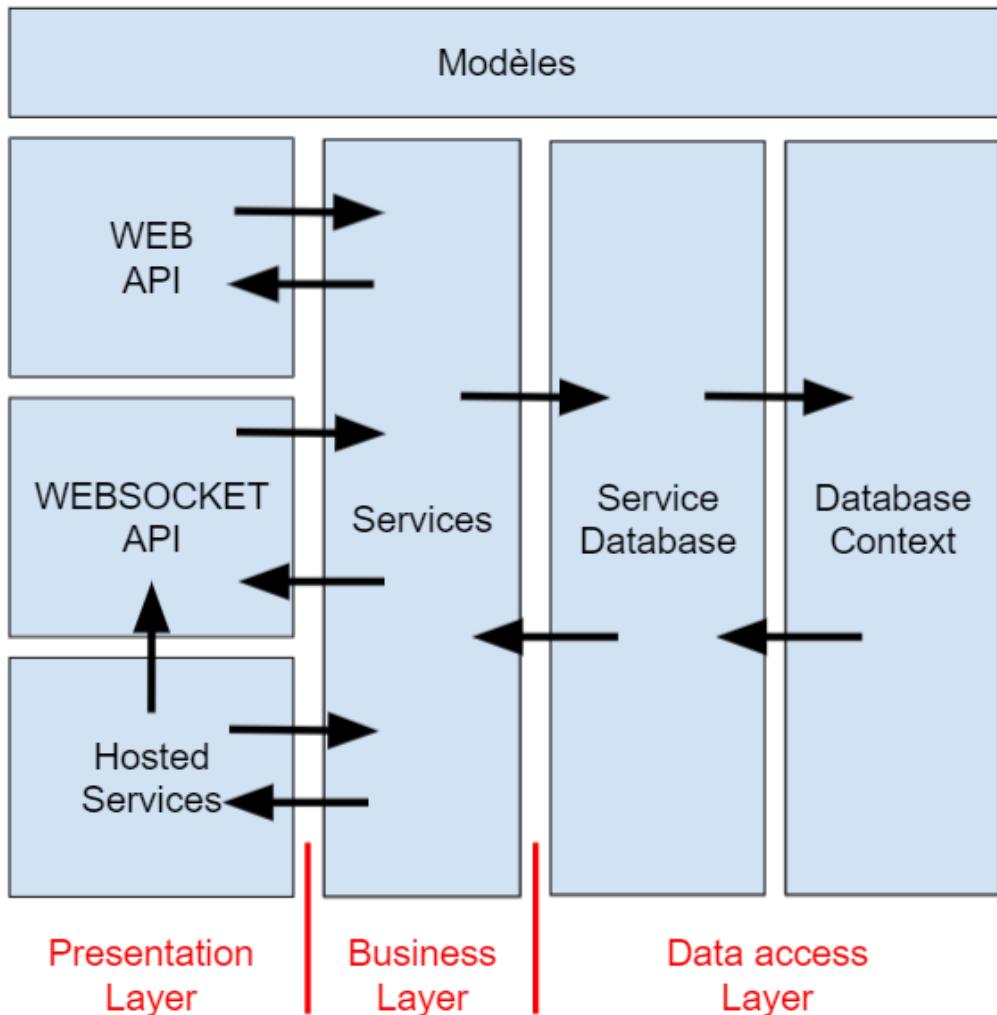
Son objectif principal est d'afficher des informations et de collecter des données auprès des utilisateurs. Il s'agit du niveau le plus élevé de l'architecture. Dans notre cas, ce sont les API et les services hébergés permettant l'envoi et la réception de données extérieures au système.

Métier

Il s'agit du cœur de l'application, mieux connu sous le nom de niveau logique ou niveau intermédiaire. Il coordonne toute la logique métier de l'application, prescrit la manière dont les objets métier interagissent les uns avec les autres. Il gère les informations collectées à partir du niveau Présentation. Au cours du processus, ce niveau peut avoir besoin d'accéder au niveau Données pour récupérer ou modifier les données. Dans une application à 3 niveaux, toutes les communications passent sans problème par le niveau Application. Le niveau Présentation et le niveau Données ne peuvent pas communiquer directement entre eux.

Données

Parfois appelé niveau base de données, où il stocke et gère les données traitées par le niveau application avec un connecteur entre lui et la base de données.



4.4- Modèles

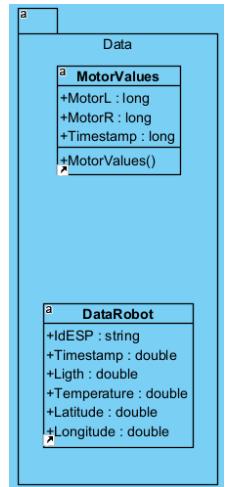
Le modèle des données est la description, la représentation des données dans notre application ainsi qu'en base de données.

4.4.1- Modèle de données robot

Le modèle des données robot sont des objets représentant la structure de données utilisée par le robot.

La classe « **MotorValues** » contient deux variables de type « long » représentant la puissance des moteurs du robot sur la plage [-100 ;100]. Cette classe représente la structure de données échangés entre l'application qui contrôle le robot et le robot lui-même.

La classe « **DataRobot** » contient plusieurs variables décrivant la luminosité, la température, la longitude, la latitude pour un robot donné à un instant T. C'est la structure de données envoyée par le robot et qui sera par la suite stockée en base de données.

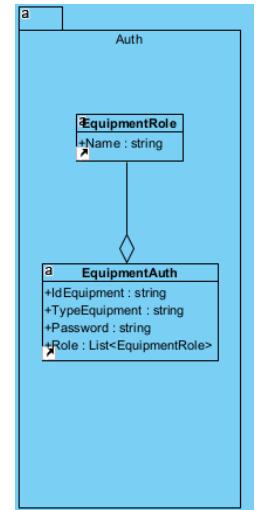


4.4.2- Modèle de données d'authentification

4.4.2.1- Modèle Database interne à l'application

Le premier modèle concerne une structure de données stockées en base de données. Cela énonce pour un robot en particulier ses informations le concernant :

- Son type (si c'est un robot, ou alors si c'est une autre application)
- Son rôle (dans le cas du robot si son rôle est de contrôler les moteurs, ou d'envoyer des images, dans le cas d'une application si son rôle est de contrôler un robot)
- Son mot de passe pour pouvoir s'authentifier



Cette classe nous servira pour stocker en base de données les revendications d'un de nos équipements et ses informations pour qu'ils puissent s'authentifier.

```
"EquipmentsAllowed": [
    "Allowed": [
        {
            "IdEquipment": "Esp32Robot",
            "TypeEquipment": "Robot",
            "Password": "25Zjqgr8AOxxZsyz",
            "Role": [
                { "Name": "Station" },
                { "Name": "Camera" }
            ]
        },
        {
            "IdEquipment": "WPFController",
            "TypeEquipment": "Application",
            "Password": "25Zjqgr8AQxxZsyz",
            "Role": [
                { "Name": "Controller" }
            ]
        }
    ],
    ...
],
```

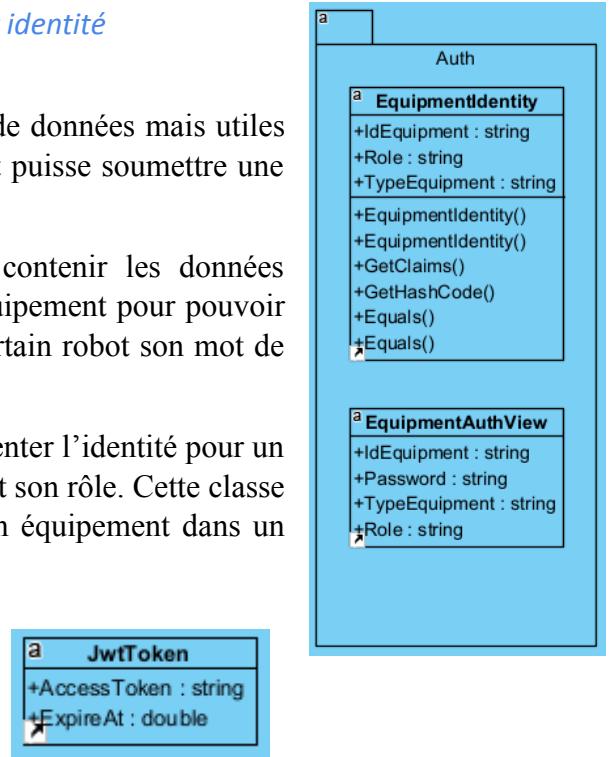
4.4.2.2- Modèle pour l'authentification et identité

Ce sont deux classes qui ne sont pas utilisées en base de données mais utiles en interne dans l'application ou pour qu'un équipement puisse soumettre une demande d'authentification.

La classe « **EquipmentAuthView** » à pour rôle de contenir les données d'authentification que l'on souhaite soumettre à un équipement pour pouvoir s'authentifier auprès du serveur. Il contient pour un certain robot son mot de passe, le type et le rôle qu'il revendique.

La classe « **EquipmentIdentity** » a pour rôle de représenter l'identité pour un certain équipement contenant son identifiant, son type et son rôle. Cette classe est utilisée par la suite stocker les revendications d'un équipement dans un token d'authentification.

La classe « **JwtToken** » contient le token d'authentification d'un équipement s'il est bien authentifié, ainsi que le temps d'expiration de l'authentification.



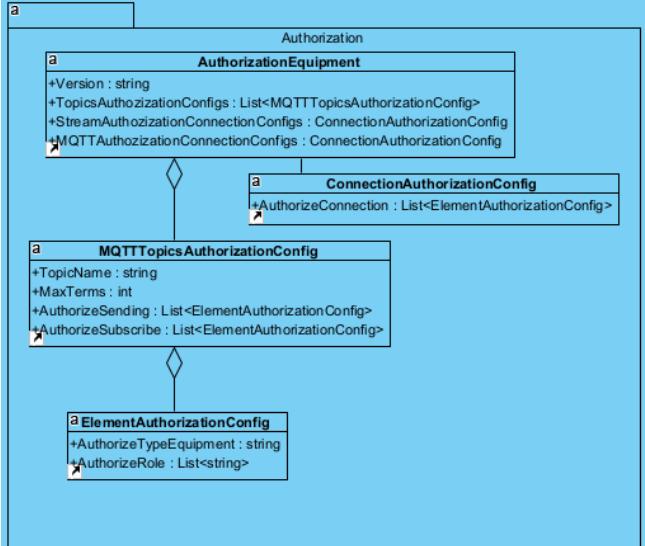
4.4.3- Modèle de données d'autorisation

Le modèle concernant les autorisations est stocké en base de données. Il contient toutes les autorisations d'accès à certaines ressources basées sur un couple type et rôle d'équipement.

Par exemple, nous avons toutes les autorisations concernant l'utilisation du broker MQTT ou du serveur TCP.

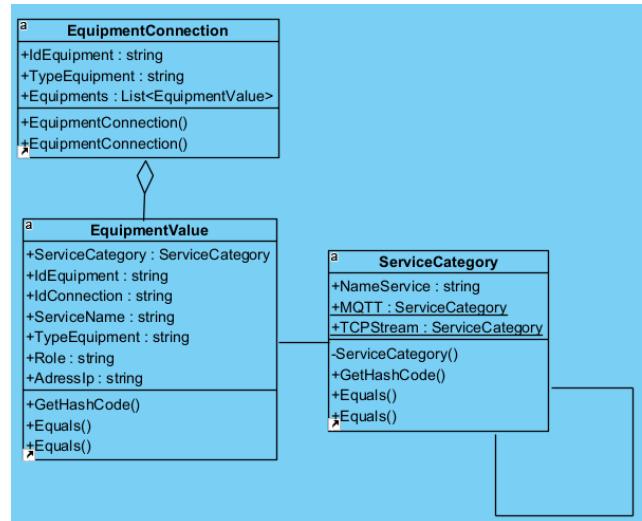
Nous avons alors plusieurs autorisations sur plusieurs événements :

- Sur les connexions
- Sur l'envoi de données sur un certain topic
- Réception de données sur un certain topic



4.4.4- Modèle de données de connexion

Le modèle de connexion à pour rôle de grouper toutes les connexions pour un équipement particulier ayant un certain type. Par exemple regrouper toutes les connexions du type d'équipement « Robot », avec chaque connexion dédiée à un rôle et à un service en particulier.



4.5- Configuration

Les fichiers de configuration définissent des réglages (affichage, langue, vitesse de transmission, protocoles de communication, prise en compte de certains périphériques, etc.) dans les applications, les services d'un serveur informatique ou les systèmes d'exploitation.

Cela est nécessaire et très important pour notre serveur applicatif afin qu'il puisse être configuré pour les propres besoins de chacun et de son environnement (développement ou production).

De cette façon, il n'y a aucune variable défini en dur dans l'application, et nous permet de les faire varier sans modifier la moindre ligne de code.

Nous avons donc un dossier contenant la configuration pour les différents services avec chacune deux versions, une pour le développement local et une autre pour la production.

```
0 références
public Startup(IWebHostEnvironment env)
{
    var basePath = $"{ env.ContentRootPath}/AppSettings";

    Configuration = ConfigurationBuilder(basePath, env.EnvironmentName, "");
    ConfigurationServices = ConfigurationBuilder(basePath, env.EnvironmentName, "Services");
    ConfigurationDatabase = ConfigurationBuilder(basePath, env.EnvironmentName, "Database");
    ConfigurationHostedServices = ConfigurationBuilder(basePath, env.EnvironmentName, "HostedServices");
}

4 références
private static IConfiguration ConfigurationBuilder(string basePath, string environmentName, string name)
{
    var path = string.IsNullOrEmpty(name) ? basePath : $"{basePath}/{name}";
    var file = string.IsNullOrEmpty(name) ? $"appsettings.{environmentName}.json" : $"appsettings.{name}.{environmentName}.json";

    return new ConfigurationBuilder()
        .SetBasePath(path)
        .AddJsonFile(file, optional: false, reloadOnChange: true)
        .AddEnvironmentVariables()
        .Build();
}
```

Nous chargeons les bonnes versions en fonction d'une variable d'environnement pour savoir si l'application est dans un environnement de développement ou de production.

Au démarrage du serveur, nous souhaitons alors chargé les configuration répartie dans différents dossiers pour chaque partie de notre application :

- La configuration général
- La configuration des services
- La configuration concernant la base de données
- La configuration des services hébergés

Grâce à la variable d'environnement contenu dans l'objet “IWebHostEnvironment”, nous pouvons déterminer si nous sommes en développement ou en production, nous pouvons chargé alors la version correspondante.

Les fichiers de configuration sont en format JSON et injectés au démarrage dans différent objet défini au préalable.

Nous détaillerons toutes les configurations des différentes parties plus tard dans le rapport.

4.6- Couche d'accès aux données (Data Access Layer)

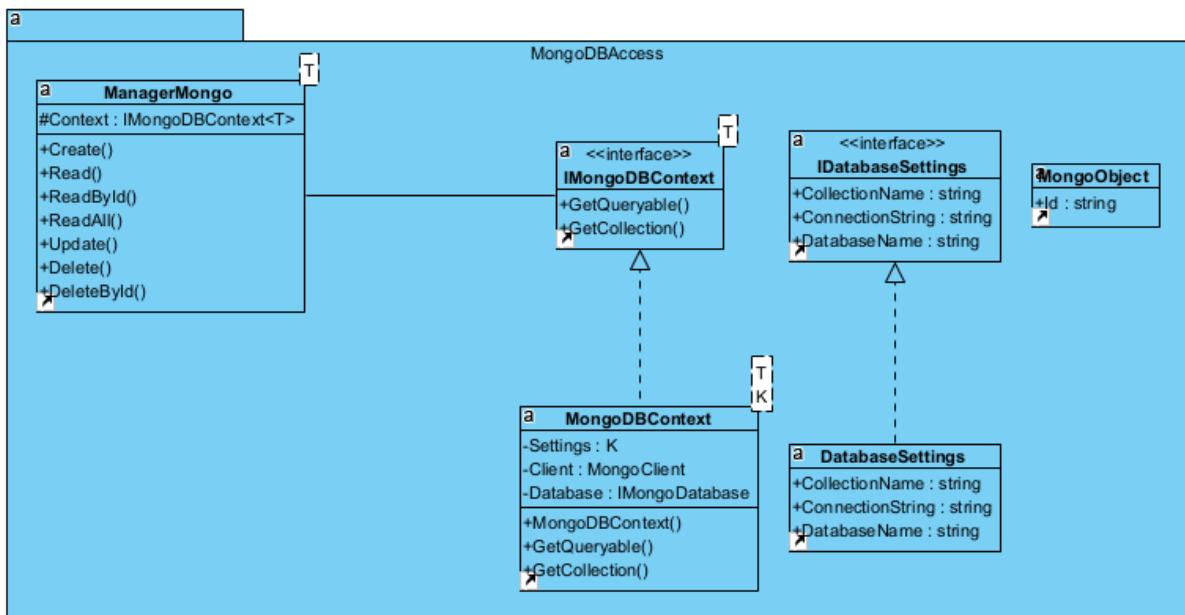
4.6.1- Contexte d'accès aux données

La couche d'accès aux données de l'application est composée de différentes classes.

Tout d'abord, nous avons une classe de configuration de la connexion à la base MongoDB. Elle contient la chaîne de connexion à la base, le nom de la Database ainsi que le nom de la collection appartenant à la Database.

Après cela nous avons la classe « **MongoDBContext** » qui utilise la classe de configuration pour se connecter à une base de données MongoDB sur une base et une collection particulière. C'est un contexte générique (T) qui prend n'importe quel type d'objet représentant les documents de la collection.

Pour finir nous avons une classe abstraite « **ManagerMongo** », qui utilise la classe précédente afin de se connecter à la base. Elle définit toutes les méthodes de base pour les opérations CRUD sur une certaine collection. Il suffit par la suite de créer une nouvelle classe qui hérite de « **ManagerMongo** » pour définir de nouvelles requêtes ou des opérations customs.



Configuration du contexte

L'objet “MongoDBContext” a besoin d'une configuration pour déterminer la chaîne de connexion à la base MongoDB , ainsi que le nom de la database et de sa collection.

Pour cela il utilise une interface qui s'appelle “IDatabaseSetting”.

Nous stockerons dans le dossier AppSettings/Database, les fichiers de configurations pour les différents objets utilisant le contexte de données.

```
7 références
public interface IDatabaseSettings
{
    7 références
    public string CollectionName { get; set; }
    7 références
    public string ConnectionString { get; set; }
    7 références
    public string DatabaseName { get; set; }
}
```

4.6.2 - Services Database

Nous avons principalement deux services :

- DataRobotService : données des différents robots
- UserService : données des différents utilisateurs de l'application

Chaque service est relié à une collection particulière de la base de données MongoDB via les objets de la couche d'accès aux données et plus particulièrement la classe « **ManagerMongo** ».

Simplement leurs rôles sont d'effectuer des opérations CRUD ou autres requêtes particulières sur la base afin de sauvegarder, mettre à jour, lire, ou supprimer des données en base.

Les deux services héritent d'une classe abstraite de la couche d'accès aux données en spécifiant un objet générique qui représente les documents de la collection MongoDB auquel le service est rattaché, par exemple :

- DataRobotService<DataRobot>
- UserService<EquipmentAuth>

Configuration

Les différents services liés à la base de données ont besoin de la configuration contenant les informations de connexion pour accéder au contexte de données. La configuration est défini dans le fichier ayant pour chemin

AppSettings/Database/appsettings.Database.json.

```
{
    "UsersDatabaseSettings": {
        "CollectionName": "users",
        "ConnectionString": "mongodb://mongoApp:27017",
        "DatabaseName": "IOT"
    },
    "DataDatabaseSettings": {
        "CollectionName": "data",
        "ConnectionString": "mongodb://mongoApp:27017",
        "DatabaseName": "IOT"
    },
    "AuthorizationDatabaseSettings": {
        "CollectionName": "authorization",
        "ConnectionString": "mongodb://mongoApp:27017",
        "DatabaseName": "IOT"
    }
}
```

4.7- Services (Business Layer)

L'application est composée de différents services. Généralement ce sont des composants de niveau intermédiaire, typiquement ce sont :

- Logique métier, ou validation de données
- Composants et logique d'accès aux données

Nous avons **principalement trois services** dans notre application, deux qui concernent **l'authentification et l'autorisation pour l'utilisation des ressources de notre application**. Enfin un dernier service, nous permettant de faire persister dans un cache, l'état des connexions actuel des clients de l'application ainsi que certaines données.

Configuration

La configuration des différents services est définie dans le fichier : **AppSettings/Service/appsettings.Services.json**, et contient tous les paramètres nécessaires pour les différents services.

Configuration du service d'authentification :

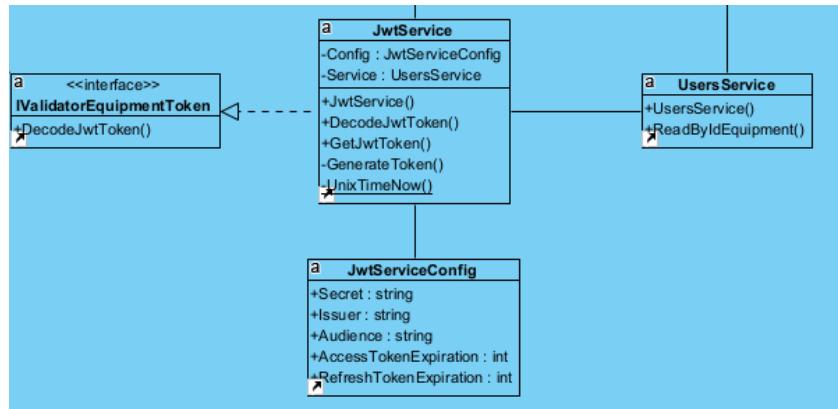
```
"JwtServiceConfig": {  
    "Secret": "XCAP05H6LoKvbRRa/QkqLNMI7c0HguaRyHzyg7n5qEkGjQmtBhz4SzYh4Fqwjyi3KJH1SXKPwVu2+bXr6CtpgQ==",  
    "Issuer": "",  
    "Audience": "",  
    "AccessTokenExpiration": 60,  
    "RefreshTokenExpiration": 10  
}
```

Configuration du service cache de connexion :

```
"EquipmentsConnectionCacheConfig": {  
    "TypesEquipments": [  
        "Robot"  
    ]  
},
```

[4.7.1- Authentification](#)

Le service d'authentification s'appelle « **JwtService** ». Il a pour rôle de générer des tokens d'authentification si un utilisateur fournit les bonnes informations d'authentification, mais aussi de les décoder par la suite. Les utilisateurs pourront alors s'authentifier auprès du service et utiliser les différentes fonctionnalités de l'application en fournissant le token généré.



[4.7.2.1- Génération du token](#)

Pour déterminer si les informations fournies par un utilisateur sont les bonnes, il se sert du **service « UserService »** afin de récupérer les informations en base et comparer le nom d'utilisateur et le mot de passe de ce dernier.

Pour prouver qu'un utilisateur est bien authentifié, nous lui fournissons un JWT token. JSON Web Token est une norme Internet proposée pour la création de données avec une signature facultative et/ou un cryptage facultatif dont la charge utile contient JSON qui revendique un certain nombre de revendications. Les jetons sont signés à l'aide d'un secret privé ou d'une clé publique/privée.

Pour générer le token, nous utilisons une clé symétrique définie dans un objet de configuration dédié pour le service d'authentification.

```

private string GenerateToken(EquipmentIdentity identity)
{
    var claims = identity.GetClaims();

    byte[] key = Convert.FromBase64String(this.Config.Secret);
    SymmetricSecurityKey securityKey = new(key);
    SecurityTokenDescriptor descriptor = new()
    {
        Subject = new ClaimsIdentity(claims),
        Expires = DateTime.UtcNow.AddSeconds(this.Config.RefreshTokenExpiration * 60),
        SigningCredentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256Signature)
    };

    JwtSecurityTokenHandler handler = new();
    JwtSecurityToken token = handler.CreateJwtSecurityToken(descriptor);
    return handler.WriteToken(token);
}

```

Nous rajoutons en plus certaines informations concernant l'utilisateur dans le token, ses rôles, son identifiant, adresse IP par exemple (ce sont les revendications).

Ensuite nous définissons **un temps d'expiration pour le token** (défini dans la configuration), nous permettant de nous assurer que le token est valide qu'un certain temps (si quelqu'un vole le token, il pourra s'en servir qu'un nombre de temps limité). Après l'expiration, l'utilisateur doit régénérer un nouveau token avec ses identifiants.

Une fois les revendications et le temps d'expiration définie, nous spécifions la clé symétrique ainsi que le protocole (HMAC avec la fonction de hachage SHA 256) pour signer le token (prouver qu'il est bien émis par nous).

HMAC avec la fonction de hachage SHA 256

Un HMAC, de l'anglais keyed-hash message authentication code (code d'authentification d'une empreinte cryptographique de message avec clé), **est un type de code d'authentification de message (CAM)**, ou MAC en anglais (message authentication code), calculé en utilisant une fonction de hachage cryptographique en combinaison avec une clé secrète. Comme avec n'importe quel CAM, il peut être utilisé pour vérifier simultanément l'intégrité de données et l'authenticité d'un message. N'importe quelle fonction itérative de hachage, comme MD5 ou SHA-1, peut être utilisée dans le calcul d'un HMAC ; le nom de l'algorithme résultant est HMAC-MD5 ou HMAC-SHA-1. La qualité cryptographique du HMAC dépend de la qualité cryptographique de la fonction de hachage et de la taille et la qualité de la clé.

Une fonction itérative de hachage découpe un message en blocs de taille fixe et itère dessus avec une fonction de compression. Par exemple, MD5 et SHA-1 opèrent sur des blocs de 512 bits. La taille de la sortie HMAC est la même que celle de la fonction de hachage (128 ou 160 bits dans les cas du MD5 et SHA-1), bien qu'elle puisse être tronquée si nécessaire.

Enfin nous générerons le token contenant diverses informations concernant l'utilisateur, nous pourrons par la suite vérifier l'intégrité et l'authenticité du token.

4.7.2.2- Décodage du token

L'utilisateur bien authentifié, peut ensuite nous fournir le token pour prouver son authentification. Pour vérifier la validité du token ou récupérer les informations de l'utilisateur contenu dans le token que nous devons décoder.

Pour cela il est possible de faire l'opération inverse, nous fournissons la clé symétrique à la fonction de déchiffrement, en vérifiant aussi si le token n'a pas expiré. Si l'opération réussit, alors le token est bien valide et nous pouvons extraire les informations de l'utilisateur, sinon le token n'est pas valide.

[4.7.2- Autorisation](#)

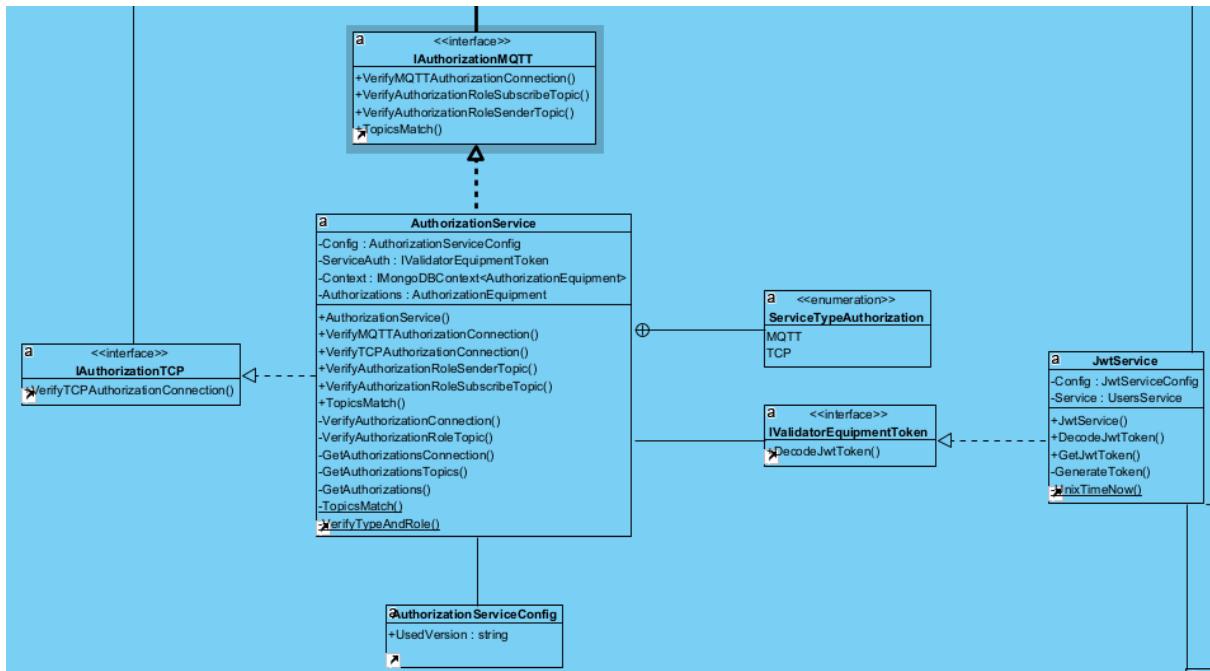
Le service d'autorisation d'utilisation des fonctionnalités s'appelle «AuthorizationService». Il a pour rôle de valider en fonction de certains critères si un utilisateur peut accéder à une ressource particulière de l'application. **Dans notre cas, un utilisateur à un type et un rôle.** Nous avons en base de données un objet qui contient pour chaque ressource quels sont les types d'utilisateur et quels rôles utilisateurs peuvent accéder à la ressource.

Le service récupère au démarrage la configuration des autorisations en base de données pour approuver par la suite l'utilisation d'une ressource.

Pour vérifier l'autorisation, l'utilisateur au préalable fournit le token d'authentification. Le service d'autorisation récupère alors les informations de l'utilisateur tel que son rôle, son type stocker dans le token, s'il est valide, grâce au service d'authentification, qui essaye de décoder le token.

Enfin, connaissant le rôle et le type de l'utilisateur, il peut comparer ses informations à l'objet contenant les autorisations, et renvoi si oui ou non l'utilisateur peut accéder à la ressource.

Principalement nous contrôlons les autorisations pour les ressources MQTT et TCP que nous verrons plus tard.

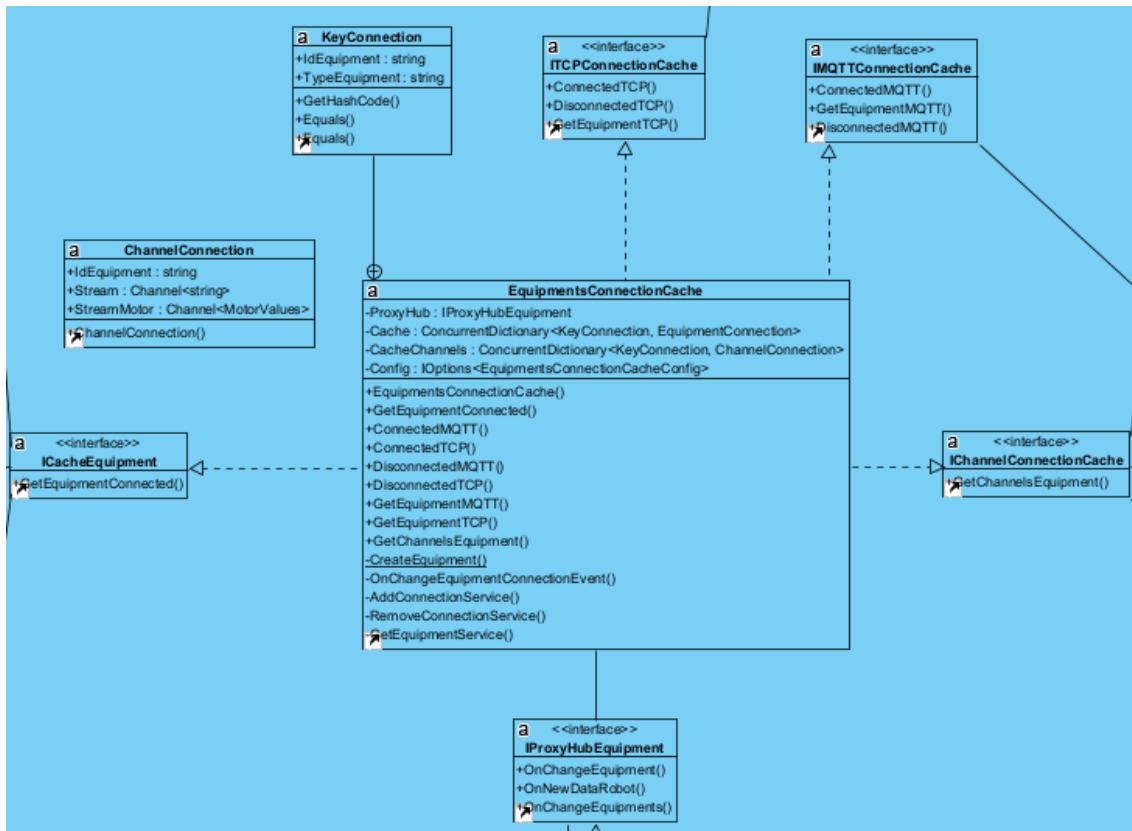


4.7.3- Cache de connexion et données

Le service de cache de connexion s'appelle « **EquipmentConnectionCache** ». Il a pour rôle de faire persister et de centraliser les connexions des deux ressources MQTT et TCP. Un utilisateur peut très bien être connecté plusieurs fois à la ressource MQTT et TCP en même temps. Le cache centralise alors quel utilisateur avec un certain type et un certain rôle est connecté à quoi.

Ce service utilise des collections thread safe car il est principalement utilisé dans un environnement multi-tâche.

Par exemple, prenons le cas d'un robot. Un robot est composé de deux ESP, les deux ont chacun un rôle différent mais le même type "Robot" car ils font un tout avec le robot. Cependant l'ESP principale aura plutôt un rôle de station pour le robot tandis que l'ESP Camera aura le rôle de caméra. Les deux peuvent utiliser MQTT et/ou TCP, et nous nous retrouvons avec plusieurs connexions pour un couple ID robot et son type. **C'est pourquoi le rôle du cache est de centraliser les connexions et de regrouper les connexions ayant le même ID et type.**



Comme nous pouvons le voir sur l'image au dessus, “EquipentConnectionCache” implémente différentes interfaces qu'utilisent les autres services.

- **ICacheEquipment** : pour récupérer la liste de tous les équipements connectés des types défini dans le fichier de configuration que l'on a vu plus haut, par exemple “Robot”.
- **ITCPConnectionCache** : Être le gestionnaire cache des connexions TCP.
- **IMQTTConnectionCache** : Être le gestionnaire cache des connexions MQTT.
- **IChannelConnectionCache** : Exposer les différents Channels des différents équipements connectés.
-

Centralisation des fluxs de données (Channels)

Il permet aussi **d'associer chaque utilisateur/équipement à des channels**. Ils permettent de **fournir et écrire des données dans des flux ou propager des événements en temps réel** concernant les utilisateurs à d'autres ressources de l'application.

Un Channel est une structure de données qui permet à un thread de communiquer avec un autre thread. Dans .NET, cela se faisait généralement en utilisant une variable partagée qui prend en charge la concurrence (en implémentant un mécanisme de synchronisation/verrouillage). Les canaux, en revanche, peuvent être utilisés pour envoyer des messages directement entre les threads sans aucune synchronisation ou verrouillage externe requis. Ces messages sont envoyés dans l'ordre FIFO (premier entré, premier sorti).

Nous les utiliserons plus tard particulièrement pour transmettre des données tel que les images envoyées par le robot mais encore les commandes envoyées au robot à des clients connecter en temps réel à l'application, tel que l'application Front-end de visualisation des robots grâce à SignalR.

A la déconnexion d'un équipement/utilisateur nous fermons le Channel signalant la fin de la transmission du stream.

4.8- Service d'échange de données temps réel

Nous avons voulu rajouter à notre application un service temps réel d'échange de données pour pouvoir transmettre ou recevoir instantanément des données aux clients connectés dès que celles-ci sont disponibles, plutôt que de laisser le serveur attendre qu'un client demande de nouvelles données.

Présentation

Nous utilisons la librairie open-source SignalR d'ASP.NET. C'est une bibliothèque qui simplifie le processus d'ajout de fonctionnalités Web en temps réel aux applications. SignalR permet également de tout nouveaux types d'applications Web qui nécessitent des mises à jour à haute fréquence du serveur, par exemple, les jeux en temps réel.

SignalR gère automatiquement la gestion des connexions et nous permet de diffuser des messages à tous les clients connectés simultanément, comme une salle de discussion. Nous pouvons également envoyer des messages à des clients spécifiques. La connexion entre le client et le serveur est persistante, contrairement à une connexion HTTP classique, qui est rétablie à chaque communication.

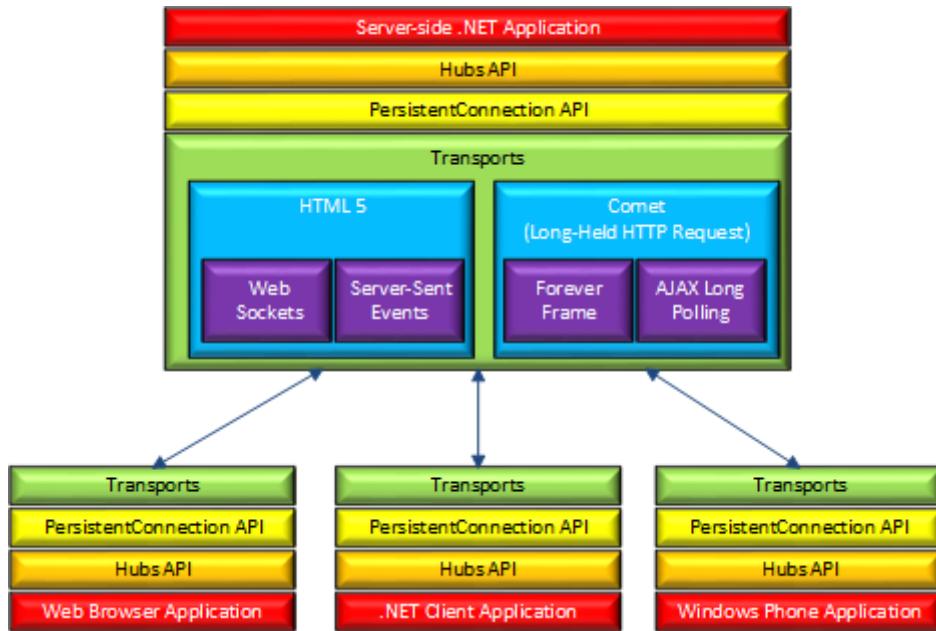
SignalR prend en charge la fonctionnalité « poussage du serveur », dans laquelle le code du serveur peut appeler le code client dans le navigateur à l'aide d'appels de procédure à distance (RPC), plutôt que le modèle de demande-réponse courant sur le Web aujourd'hui.

Transport

SignalR est une abstraction sur certains des transports nécessaires pour effectuer un travail en temps réel entre le client et le serveur. SignalR tente d'abord d'établir une connexion WebSocket si possible. WebSocket est le transport optimal pour SignalR car il a :

- Utilisation la plus efficace de la mémoire serveur.
- Latence plus faible.
- Fonctionnalités les plus sous-jacentes, tel que la communication en duplex intégral entre le client et le serveur.

Le diagramme suivant montre la relation entre les concentrateurs, les connexions persistantes et les technologies sous-jacentes utilisées pour les transports.



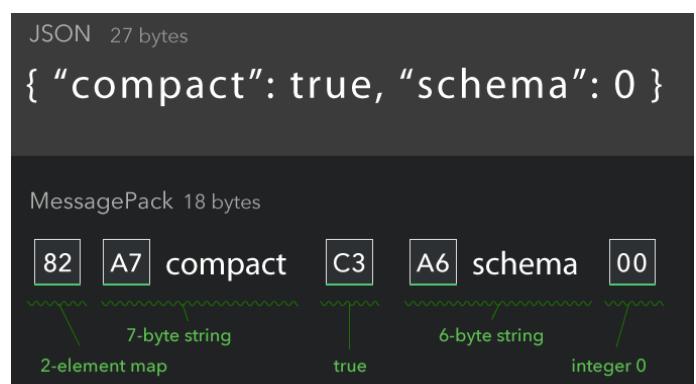
Fonctionnement des hubs

Un hub est un pipeline de plus haut niveau basé sur l'API de connexion qui permet à notre client et à notre serveur d'appeler directement des méthodes l'un sur l'autre. SignalR permet aux clients d'appeler des méthodes sur le serveur aussi facilement que des méthodes locales, et vice versa. L'utilisation d'un Hub nous permet également de passer des paramètres fortement typés aux méthodes, permettant la liaison de modèle.

Lorsque le code côté serveur appelle une méthode sur le client, un paquet est envoyé via le transport actif qui contient le nom et les paramètres de la méthode à appeler (lorsqu'un objet est envoyé en tant que paramètre de méthode, il est sérialisé à l'aide de JSON). Le client fait ensuite correspondre le nom de la méthode aux méthodes définies dans le code côté client. S'il y a une correspondance, la méthode client sera exécutée à l'aide des données de paramètre déserialisées.

Format de sérialisation

Comme il a été mentionné plus haut, nous utilisons le format de sérialisation JSON. Cependant, par exemple pour transmettre des images, les messages sous la forme de chaînes



caractères sont 33% plus volumineuse que sous le format binaire.

Pour cela nous utilisons en plus de JSON un autre format de sérialisation avec SignalR, MessagePack.

MessagePack est un format de sérialisation binaire efficace. Il vous permet d'échanger des données entre plusieurs langues comme JSON. Mais c'est plus rapide et plus petit. Les petits entiers sont codés dans un seul octet et les chaînes courtes typiques ne nécessitent qu'un seul octet supplémentaire en plus des chaînes elles-mêmes.

Diffusion en continu (stream) de données temps réel

ASP.NET Core SignalR prend en charge la diffusion en continu du client au serveur et du serveur au client. Ceci est utile pour les scénarios où des fragments de données arrivent au fil du temps. Lors du streaming, chaque fragment est envoyé au client ou au serveur dès qu'il devient disponible, plutôt que d'attendre que toutes les données soient disponibles.

Implémentation dans le serveur

Dans notre application nous possédons un total de trois hubs SignalR.

EquipmentsHub

Ce concentrateur à pour rôle de fournir à un client la liste de tous les équipements actuellement connectés à l'application. Il utilisera alors le service de cache de connexion pour récupérer toutes les informations et les transmettre à la demande du client.

EquipmentHub

```
4 références
public class EquipmentsHub : Hub
{
    private readonly ICacheEquipment Cache;

    0 références
    public EquipmentsHub(ICacheEquipment cache)
    {
        Cache = cache;
    }

    0 références
    public List<EquipmentConnection> EquipmentStatus()
    {
        return Cache.GetEquipmentConnected();
    }
}
```

Ce concentrateur à pour rôle de fournir des données à un client pour un équipement en particulier en fonction de son identifiant.

Pour cela à la connexion, **le client doit fournir en paramètre l'identifiant de l'équipement auquel il souhaite recevoir ou demander des informations.** Le concentrateur à la connexion récupère l'identifiant et vérifie si un équipement avec cet identifiant existe.

- S'il n'existe pas, il met simplement fin à la connexion du client.
- Si il existe alors le concentrateur le rajoute à un groupe de discussion dont le nom est l'identifiant de l'équipement (un topic en quelque sorte).

Avant de le rajouter au groupe, le concentrateur sauvegarde dans la variable de session du client l'identifiant de l'équipement auquel il s'est abonné. A la déconnexion, il supprimera le client du bon groupe via l'identifiant dans la variable de session.

Le client peut ainsi attendre plusieurs données venant du serveur concernant un équipement en particulier :

- Recevoir ou obtenir le statut de connexion de l'équipement, si il à changer il est avertie directement du changement.
- Recevoir les dernières données de l'équipement juste après qu'elle soit sauvegardé en base de données.
- Obtenir les N dernières données de l'équipement de la base de données.

Pour cela le concentrateur utilise les services “DataRobotService” et le cache de connexion pour récupérer les informations à la demande du client.

EquipementStreamHub

Ce concentrateur plus spécifique que celui au-dessus à pour rôle de délivré des données en continu (stream) d'un équipement en particulier. Ce sont des données soit lourdes, soit qui sont très souvent transmises, c'est pourquoi ce hub utilise le format de MessagePack.

Les clients peuvent s'abonner à un stream d'un équipement en particulier en fournissant l'identifiant de celui-ci. En retour, le concentrateur fournit le ChannelReader de l'équipement. Dès qu'une nouvelle donnée est écrite dans le Channel de l'équipement, tous les clients via SignalR la recevront grâce au ChannelReader auquel ils se sont abonnés.

Le concentrateur récupère le Channel d'un équipement en particulier grâce au service de cache de connexion contenant les Channels des équipements déjà connectés. Si le client demande un Channel d'équipement qui n'est pas connecté nous soulevons une erreur mettant fin au stream de données.

```
private readonly IChannelConnectionCache ChannelHandler;
private readonly ILogger<EquipmentStreamHub> Logger;

0 références
public EquipmentStreamHub(IChannelConnectionCache channelHandler, ILogger<EquipmentStreamHub> logger)
{
    Logger = logger;
    ChannelHandler = channelHandler;
}

0 références
public ChannelReader<string> EquipmentStream(string idEquipment)
{
    return ChannelHandler.GetChannelsEquipment(idEquipment)?.Stream.Reader;
}

0 références
public ChannelReader<MotorValues> EquipmentStreamMotor(string idEquipment)
{
    return ChannelHandler.GetChannelsEquipment(idEquipment)?.StreamMotor.Reader;
}
```

ProxyHub

Jusqu'à maintenant les concentrateurs sont utilisés pour gérer les demandes des clients. Cependant le serveur peut fournir des données aux clients sur un nom de méthode auxquelles ils se sont abonnées.

Pour cela nous avons créé un proxy permettant d'envoyer des données de l'application serveur vers les clients sans aucune demande des clients.

Le proxy utilise les concentrateurs que nous avons défini au-dessus pour accéder au contexte des concentrateurs (accéder aux clients connecter à chaque hub). Ainsi, les services qui veulent avertir les clients connecter via Signal utiliserons le proxy pour l'envoi des données.

```
2 références
public Task OnChangeEquipment(EquipmentConnection data)
{
    return EquipmentHub.Clients.Group(data.IdEquipment).SendAsync("onChangeEquipment", data);
}

2 références
public Task OnChangeEquipments(List<EquipmentConnection> data)
{
    return EquipmentsHub.Clients.All.SendAsync("onChangeEquipments", data);
}

2 références
public Task OnNewDataRobot(DataRobot data)
{
    return EquipmentHub.Clients.Group(data.IdESP).SendAsync("onDataEquipment", data);
}
```

Comme nous pouvons le voir sur l'image au dessus, si un service souhaite avertir, par exemple les clients abonné à un robot en particulier, de la réception de nouvelles données capteurs, alors il utilisera la méthode “OnNewDataRobot” qui enverra à tous les clients du groupe de l'identifiant, les nouvelles données via l'événement “onDataEquipment” gérée par le client.

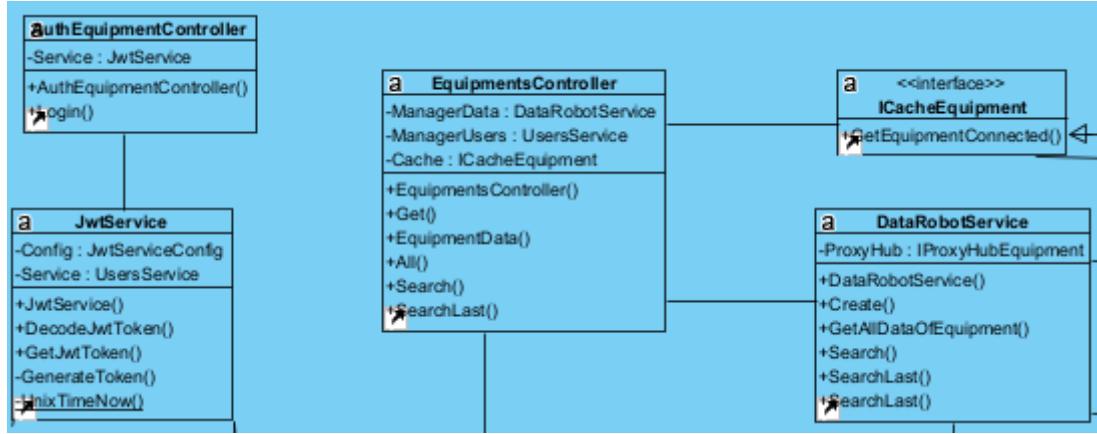
4.9- API

L'application dispose d'une API web permettant de recevoir des requêtes HTTP et d'y répondre. L'API permet de s'authentifier à l'application en fournissant des identifiants, mais aussi accéder aux données des robots.

Elle dispose de deux contrôleurs :

- AuthEquipmentController : Rôle d'authentifier et fournir un JWT token à l'utilisateur s'il fournit les bonnes informations de connexion. Il utilise le service d'authentification pour générer un token d'authentification.

- EquipmentController : Pour obtenir par exemple tous les robots qui sont actuellement online, mais aussi récupérer toutes les datas des robots situé en base de données. Il utilise le service “DataRobotService” et “ICacheEquipment”.



4.10- Services hébergés

Les services hébergés dans l'application sont en quelque sorte des sous applications/serveurs. Nous avons principalement 3 services hébergés :

- Broker MQTT : c'est un serveur MQTT permettant à des clients de s'abonner et publier des messages. Principalement pour contrôler les robots ou recevoir les données des capteurs des robots.
- Serveur TCP : c'est un serveur permettant de recevoir les images en temps réel des robots.
- SignalR : permet à d'autres applications de se connecter à l'application pour recevoir en temps réel des informations concernant les robots, tel qu'un robot vient de se connecter, ou alors un robot vient d'envoyer une image.

4.10.1- Service MQTT

Le service MQTT est un service hébergé dans le serveur applicatif. C'est un broker MQTT permettant la connexion d'autres applications pour permettre l'échange de données entre elles.

Configuration

La configuration du service hébergé est disponible dans le fichier : **AppSettings/HostedServices/appsettings.HostedServices.json**. Il contient la définitions

pour les ports du broker (port non sécurisé et port sécurisé), mais aussi le nom des topics sur lesquels il doit pouvoir effectuer des actions (nous verrons cela plus bas dans la même partie).

```
{  
    "MQTTServiceConfig": {  
        "MQTTConfigBroker": {  
            "Port": 1883,  
            "EncryptedPort": 8884  
        },  
        "TopicNameDataRobot": "IOT/Data",  
        "TopicNameMotorDataRobot": "IOT/Controler"  
    },  
}
```

Fonctionnement

Le service est lancé au démarrage du service applicatif.

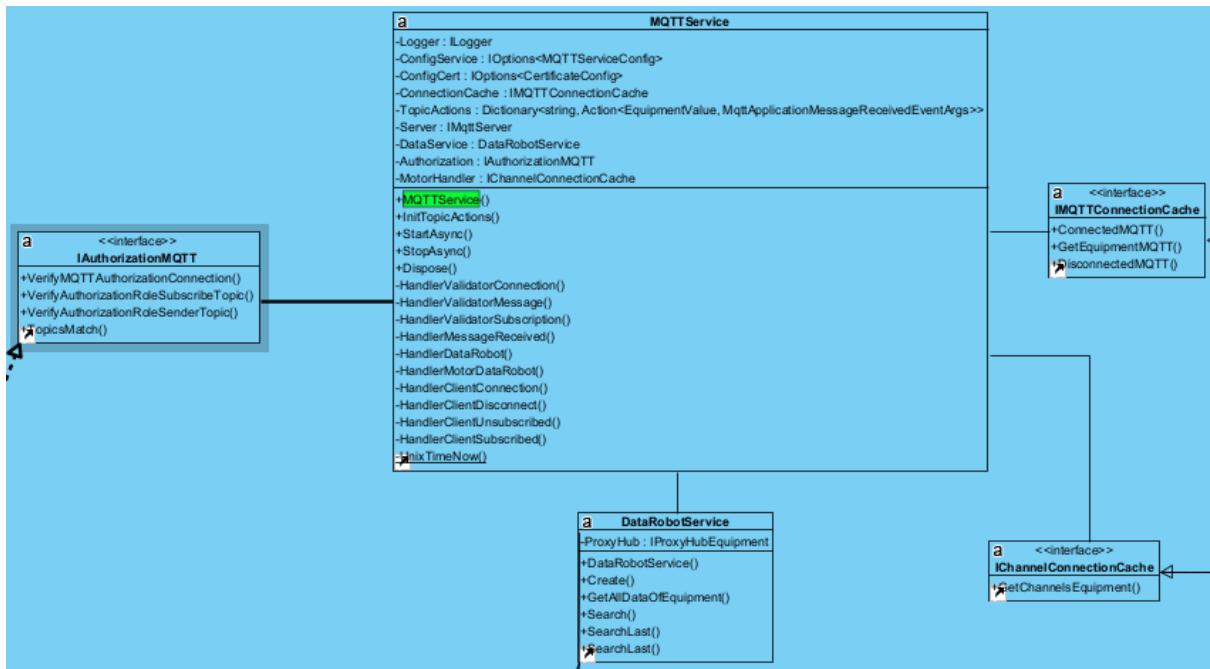
Le service MQTT contrôle la connexion, l'abonnement et l'envoi de messages sur des topics. Cela est possible en gérant tous les événements disponibles d'un broker MQTT. Il traite les événements suivants :

- Gestionnaire et Validateur de réception de message
- Gestionnaire et Validateur de connexion
- Gestionnaire de déconnexion
- Gestionnaire et Valideur d'abonnement
- Gestionnaire de désabonnement

Les gestionnaires permettent pour chaque type d'événement d'exécuter du code additionnel. Par exemple, je souhaite exécuter une tâche particulière lorsqu'un certain client se connecte à moi. Tandis que les valideurs ont plutôt un rôle de contrôle, d'autorisation, et peuvent accepter ou refuser une action d'un client sur le broker.

Le service MQTT utilise plusieurs sous service que nous avons définie plus tôt. Il utilise :

- DataRobotService : sauvegarde des données robot à la réception
- IChannelConnectionCache : pour publier les commandes moteurs reçus pour un robot dans un stream
- IMQTTConnectionCache : pour faire persister les informations d'un client connecter.
- IAuthorizationMQTT : pour déterminer les autorisations des clients sur le broker.



Gestionnaire et validateur de connexion

Lors de la tentative de connexion au service MQTT, le client doit spécifier un token d'authentification qu'il a obtenu précédemment en s'identifiant auprès du serveur via l'API. Ce token doit être défini dans le champ `username` d'une demande de connexion.

La connexion du client déclenche alors dans le service un callback de validation de la connexion. La connexion sera refusée dans plusieurs cas :

- Si le service d'autorisation n'autorise pas la connexion car :
 - Token d'authentification non valide.
 - Client fournissant le token n'a pas le droit de connexion MQTT.
- Si il existe déjà une connexion pour un client ayant les mêmes informations définies dans le token d'authentification. Dans ce cas, il doit attendre que le client précédent soit déconnecté du service.

Si le validateur constate que le token est valide, qu'il a bien le droit de connexion au broker et qu'il n'existe déjà pas un client connecté avec les mêmes informations contenues dans le token, alors il accepte la connexion du client.

Quant au gestionnaire, si le validateur accepte la connexion, il ajoute au cache de connexion, une nouvelle connexion au service MQTT avec l'identité (ID, type et rôle) du client.

Gestionnaire et validateur d'abonnement

Une fois qu'un client est bien connecté au broker, il peut s'abonner à plusieurs topics. Cependant notre broker n'autorise pas l'abonnement de tous les topics à n'importe qui.

Pour cela le validateur d'abonnement récupère l'identité du client dans le cache de connexion grâce à l'ID client MQTT. Ainsi il peut ensuite utiliser le service d'autorisation pour valider que l'identité du client a le droit de pouvoir s'abonner au topic qu'il demande. Dans le cas contraire, il refuse l'abonnement du client et il ne recevra pas les messages publiés sur ce topic.

Par exemple, l'application de contrôle des robots n'a pas à avoir accès au topic de publication des données des robots, cela n'a pas de sens, tout comme l'ESP Caméra n'a pas à avoir accès aux données publiées sur les topics de contrôle des robots.

Gestionnaire et validateur de message

Tout comme le validateur d'abonnement, le validateur de message permet d'autoriser ou non un client d'envoyer des messages sur un certain topic. Il utilise lui aussi le service d'autorisation en récupérant l'identité du client dans le cache de connexion avec son ID.

Quant à lui, le gestionnaire de message permet d'effectuer des actions supplémentaires à la réception d'un message dans un certain topic. Il existe dans le service MQTT un dictionnaire associant un topic comme clef, et une fonction de callback comme valeur pour effectuer une certaine action lors de la réception d'un message d'un certain topic.

Nous utilisons le gestionnaire de message pour pouvoir par exemple faire persister en base de données les données des capteurs des ESP.

A la réception du message contenant les données, nous récupérons l'identité du client et nous exécutons le callback associé au topic "IOT/Data". Le callback aura pour rôle de déserialiser le message JSON en objet du modèle présenté plus haut "DataRobot". Il ajoutera en plus le timestamp de réception des données. Enfin il utilisera le service "DataRobotService" pour ajouter les données en base.

Nous utilisons aussi le gestionnaire de message pour propager les données de contrôle des robots. Par exemple à la réception d'un message sur le topic "IOT/Controler", nous déserialisons le message en un objet "MotorValue" pour enfin pousser les données dans le stream moteur du client associé à ce contrôle. Cela permet par exemple une visualisation temps réel des contrôles envoyés sur un dashboard (nous verrons cela dans plus tard dans le rapport).

Lorsqu'il y a publication de nouvelles données sur les topics "IOT/Data" et "IOT/Controler" alors le service MQTT propage donc les données vers les clients connectés à SignalR (que nous avons vu précédemment).

- IOT/Data : il récupère le proxy hub et signale grâce à lui qu'un nouvel objet "DataRobot" a été reçu pour un certain équipement.
- IOT/Controler : Il récupère le Channel de l'équipement concerné et écrit l'objet dans le flux. Le client connecter en stream sur ce channel recevra alors le nouvel objet.

4.10.2- Service TCP

Le service TCP est un service hébergé dans le serveur applicatif. C'est un serveur permettant la réception de données d'images d'autres applications, plus particulièrement l'application de l'ESP Caméra. Le serveur est entièrement codé en mode asynchrone.

Configuration

La configuration du service hébergé est disponible dans le fichier : **AppSettings/HostedServices/appsettings.HostedServices.json**. Il contient la définition du port à utiliser pour accepter les connexions clientes.

```
"TCPServiceConfig": {
    "Port": 11000
}
```

Fonctionnement

Le serveur attend une connexion client sur un certain port. Une fois qu'une connexion client est acceptée, il sauvegarde dans un cache interne au serveur TCP la socket cliente, et commence la lecture de la socket de façon asynchrone.

Le protocole TCP est orienté flux, c'est à dire que lorsque l'on reçoit des données, il n'est pas assuré que toutes les données ont été reçues d'un coup, tout comme les bytes reçus peuvent contenir plusieurs données (de paquet différents). Il est donc nécessaire de mettre en place un protocole de niveau applicatif pour "parser l'échange de données".

Parseur

Dans notre cas nous ferons au plus simple, nous utiliserons un délimiteur entre les paquets. Le client avant d'envoyer ce qu'il souhaite transmettre, enverra la taille des bytes totales du paquet contenant les données totales. La taille est un entier codé sur 32 bits, il sera alors de taille 4 bytes.

Le serveur TCP lira alors les 4 premiers bytes pour savoir combien de bytes il doit lire avant d'être assuré d'avoir obtenu les données complètes.

Toutefois, ce protocole applicatif n'est pas sûr à 100%. Le serveur peut se désynchroniser du client et ainsi lire les 4 mauvais bytes et les considérer comme la taille du prochain paquet. Pour cela nous détectons la désynchronisation, et nous déconnectons alors le client. Il lui suffira, pour lui par la suite, de se reconnecter au serveur pour se resynchroniser avec lui.

Nous avons écrit un petit parser afin de faciliter la gestion du protocole applicatif. Il utilise une structure de données pour faire persister les bytes du paquet en train d'être reçu.

```
7 références
public interface IStateBuffer
{
    7 références
    public int CurrentSizeAttemping { get; set; }
    5 références
    public int RestSize { get => CurrentSizeAttemping - CurrentSizeBuffer; }
    7 références
    public int CurrentSizeBuffer { get; set; }
    6 références
    public byte[] Buffer { get; set; }
    8 références
    public byte[] Temp { get; set; }
}
```

Il contient la taille des bytes que l'on s'attend à recevoir pour considérer le paquet comme complet. La taille actuelle de données reçue, ainsi que le buffer contenant les données du paquet ainsi qu'un buffer temporaire.

Si le RestSize est égale à 0, alors nous devons extraire les quatres premiers bytes pour définir la taille totale du paquet. Si par malheur nous n'avons pas reçu la totalité des quatres premiers octets alors nous stockons les bytes dans le buffer temporaire.

Une fois les quatres bytes de la taille du paquet reçus, nous convertissons en un entier sur 32 bits et nous sauvegardons cette taille dans CurrentSizeAttemping.

Si le RestSize, donc la différence entre le nombre de bytes dans le buffer et la taille du paquet n'est pas égale à 0 alors nous rajoutons les nouveaux bytes dans le buffer et on incrémente la taille actuelle du buffer.

Dans le cas où la taille des bytes reçus dépasse la taille que nous attendons encore de recevoir, nous récupérons seulement les bytes pour combler la fin du paquet et mettons en attente les autres bytes.

Si à la fin de l'itération le RestSize est égale à 0 alors nous avons reçu le paquet entier et appelons une fonction de callback qui traitera le paquet.

Si toutefois il reste des bytes en attente après alors nous rappelons la même méthodes qui les traitera.

Une fois le parseur écrit nous pouvons commencer à recevoir les données du client.

Validateur de connexion

Pour que le client puisse commencer à envoyer des images, il est nécessaire au préalable qu'il s'authentifie auprès du service TCP. Pour cela, il doit envoyer au préalable son token d'authentification obtenu avec l'API.

Il fournit ensuite le token au service d'autorisation qui validera ou non la connexion, c'est-à-dire il déchiffra le token et validera les accès au service TCP avec les revendications du token.

Si le client n'a pas l'autorisation, alors le serveur ferme la connexion et supprime la socket de son cache interne.

Autrement, il génère un ID de connexion pour ce client, vérifie qu'une autre connexion n'existe pas avec la même identité et l'ajoute au cache de connexions. Puis commence à recevoir les données des images.

Traitement des données

A chaque fois que le serveur TCP reçoit une nouvelle image, il l'ajoute à une file d'attente. En parallèle, une autre tâche traite les images de la file d'attente. La file d'attente est thread-safe et nous utilisons un mécanisme de synchronisation pour réveiller la tâche quand il y a de nouvelles données dans la file d'attente.

La tâche de traitement de la file d'attente itère sur la queue jusqu'à ce qu'il n'y est plus de données d'image disponible dans la file en dépliant le premier élément à chaque fois. Pour chaque image, il convertit les bytes en Base64 et pousse l'image sous la forme Base64 dans le stream (Channel) du client associé à l'image en récupérant le channel via le service de cache de connexion.

La tâche de traitement de la file d'attente est un autre service hébergé à part lancé au démarrage de l'application.

4.11- Injection de données en base

Quand nous lançons une version vide de l'application, il n'y a aucune donnée dans la base Mongo. Cependant nous avons besoin d'initialiser plusieurs données concernant les identifiants ainsi que les informations d'authentification des utilisateurs/équipement de notre application mais aussi toutes les autorisations.

C'est pourquoi nous avons un petit tool qui s'exécute dans un programme en ligne de commande.

```
"EquipmentsAllowed": {
  "Allowed": [
    {
      "IdEquipment": "Esp32Robot",
      "TypeEquipment": "Robot",
      "Password": "25Zjqgr8AQxxZsyz",
      "Role": [
        { "Name": "Station" },
        { "Name": "Camera" }
      ]
    },
    {
      "IdEquipment": "WPFController",
      "TypeEquipment": "Application",
      "Password": "25Zjqgr8AQxxZsyz",
      "Role": [
        { "Name": "Controller" }
      ]
    }
  ]
},
```

Il a pour rôle au lancement des applications de se connecter à la base Mongo et d'insérer les données en base.

Les données sont contenues dans un fichier de configuration dans le projet “MongoSeeding”.

Si nous voulons par défaut ajouter de nouveaux équipements, ou de nouvelles autorisation il suffit de modifier le JSON ayant pour nom “appsettings.json” dans le projet “MongoSeeding”.

A droite nous avons les données d'authentification et d'identification par défaut qui sont insérées en base.

Données par défaut des autorisations pour les utilisateurs insérer en base :

```
    "AuthorizationEquipment": {
        "Version": "1",
        "TopicsAuthzationConfigs": [
            {
                "TopicName": "IOT/Data",
                "MaxTerms": 2,
                "AuthorizeSending": [
                    {
                        "AuthorizeTypeEquipment": "Robot",
                        "AuthorizeRole": [ "Station" ]
                    }
                ],
                "AuthorizeSubscribe": []
            },
            {
                "TopicName": "IOT/Controler",
                "MaxTerms": 3,
                "AuthorizeSending": [
                    {
                        "AuthorizeTypeEquipment": "Application",
                        "AuthorizeRole": [ "Controler" ]
                    }
                ],
                "AuthorizeSubscribe": [
                    {
                        "AuthorizeTypeEquipment": "Robot",
                        "AuthorizeRole": [ "Station" ]
                    }
                ]
            }
        ],
        "StreamAuthziationConnectionConfigs": {
            "AuthorizeConnection": [
                {
                    "AuthorizeTypeEquipment": "Robot",
                    "AuthorizeRole": [ "Camera" ]
                }
            ]
        },
        "MQTTAuthziationConnectionConfigs": {
            "AuthorizeConnection": [
                {
                    "AuthorizeTypeEquipment": "Robot",
                    "AuthorizeRole": [ "Station" ]
                },
                {
                    "AuthorizeTypeEquipment": "Application",
                    "AuthorizeRole": [ "Controler" ]
                }
            ]
        }
    }
```

5- Application mobile

5.1- Présentation

Cette application a pour vocation de transmettre à l'ESP principal, via Bluetooth, les coordonnées du téléphone (Latitude, Longitude uniquement). Nous faisons ceci afin que le téléphone se substitue comme une balise GPS pour le robot.

L'application est assez rudimentaire, elle permet entre autre de :

- Déetecter si le bluetooth du téléphone est actif
- Scanner les alentours pour découvrir les périphériques bluetooth
- Se connecter à un périphérique
- Démarrer le partage de la position

Une fois le service de partage de position enclenché, l'application transmettra toutes les 5 secondes la localisation du téléphone. Pour la mise en œuvre nous utilisons l'ESP comme serveur bluetooth et le téléphone comme client de ce serveur. L'application mobile utilise les capteurs Bluetooth et GPS du téléphone en plus de certains plugins liés à l'environnement de développement.

En ce qui concerne la transmission des données, nous utilisons du JSON pour sérialiser la latitude et la longitude, suite à ça nous encodons les données dans une séquence d'octet que nous stockons dans un tableau et que nous envoyons via Bluetooth en nous connectant au service dédié par le serveur bluetooth qu'est l'ESP.

<https://www.bluetooth.com/blog/a-developers-guide-to-bluetooth/> (pour les L3, pour comprendre le fonctionnement service/characteristic du Bluetooth)

5.2- Environnement de développement

Pour le développement nous nous sommes tournés vers Xamarin pour sa solution cross platform assez simple à mettre en œuvre. L'application en écrite en C# via Xamarin.Forms, qui permet de créer des applications natives en utilisant un kit d'outils d'interface utilisateur multi plateforme .NET qui cible les facteurs de forme mobile, tablette et poste de travail sur Android, iOS, etc.

A cette solution nous avons ajouté plusieurs plugins utile pour le développement :

- Newtonsoft : pour la sérialisation en JSON des données
- Geolocator : bien que Xamarin fournit une bibliothèque permettant de connaître la position, Geolocator fournit la même chose en plus d'une fonction d'écoute utile pour transmettre les données toutes les X secondes
- BLE : c'est le plugin le plus important puisqu'il fournit toutes les fonctions pour scanner, se connecter et envoyer des données via Bluetooth

Pour le rendu graphique de l'application Xamarin utilise le langage XAML ; en ce qui concerne l'IDE nous utilisons Visual Studio 2019 et le framework .NET.

Pour finir, l'exécution de l'application se fait via un Samsung Galaxy J3 6 (Android 5.1 API 22) pour Android. La partie Apple et iOS n'est pas active du fait des contraintes liées à Apple (Mac et iPhone) cependant du fait que le code sur Xamarin soit commun au 2 environnements celle-ci peut être implémentée sans trop de problèmes.

6- Contrôle du robot

6.1- Présentation

Ces applications ont pour but de pouvoir contrôler des robots à distance via le Cloud grâce à notre serveur applicatif (middleware). Chacune des applications est cliente de notre middleware.

Nous avons développé plusieurs façons de contrôler un robot, par manette, kinect, mais encore piloter grâce à de la reconnaissance d'image fournie par le robot. Nous avons donc deux applications :

- Application de contrôle par manette / kinect
- Application de contrôle par reconnaissance d'images

Chaque application reçoit des données en entrée (données de manette, kinect, détection), et les convertit en données exploitables pour le contrôle des moteurs du robot. Les données des moteurs sont par la suite envoyées au middleware qui se chargera de délivrer les commandes au robot grâce au protocole de transmission de données MQTT.

Nous avons convenu ensemble d'utiliser une plage de valeurs [-100;100] pour représenter la puissance et le sens des différents moteurs pour toutes les applications.

Nous allons détailler le fonctionnement des deux applications.

6.2- Application WPF (manette/kinect)

6.2.1- Environnement de développement

Windows Presentation Foundation (WPF) est une infrastructure d'interface utilisateur qui crée des applications clientes de bureau. La plate-forme de développement WPF prend en charge un large éventail de fonctionnalités de développement d'applications, notamment un modèle d'application, des ressources, des contrôles, des graphiques, une mise en page, une liaison de données, des documents et la sécurité. Le framework fait partie de .NET. WPF utilise le langage XAML (Extensible Application Markup Language) pour fournir un modèle déclaratif pour la programmation d'applications.

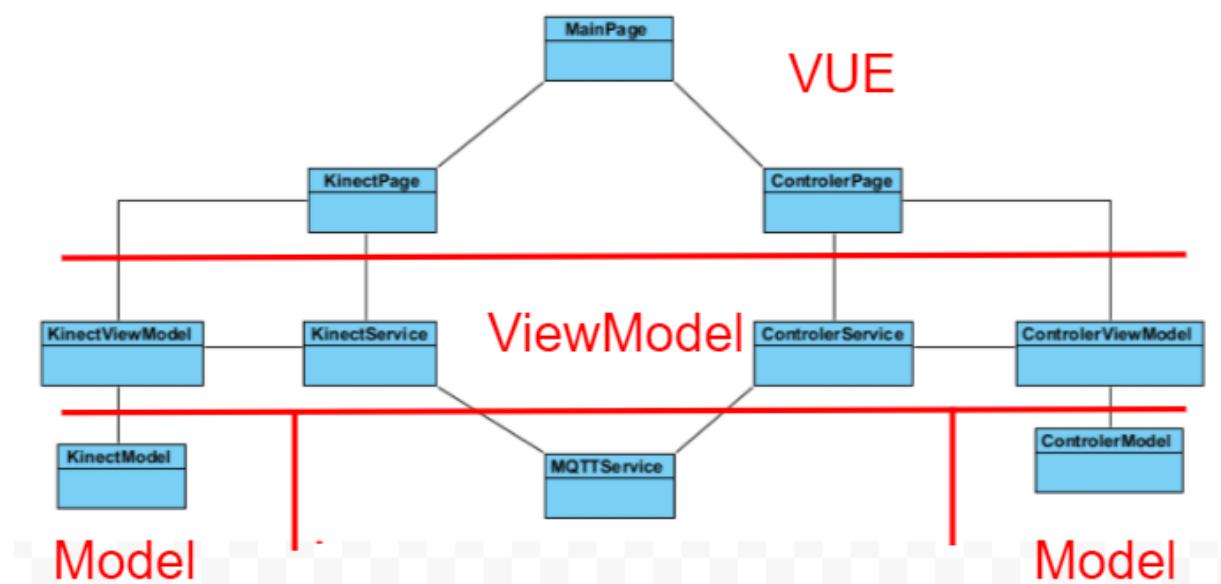
Nous utilisons Visual Studio 2019. L'application cible le Framework .NET framework 4.8 de Microsoft. Nous utilisons pour ce projet l'ancien framework (.NET framework) car nous utilisons une Kinect V1. Le SDK Kinect est uniquement disponible pour les projets utilisant .NET framework et n'est pas portable dans les versions .NET Core.

6.2.2- Architecture

Les applications WPF utilisent une architecture logiciel MVVM. Le Model View ViewModel (MVVM) est un modèle architectural utilisé en génie logiciel qui provient de Microsoft qui est spécialisé dans le modèle de conception de modèle de présentation. Il est basé sur le modèle MVC (Modèle-vue-contrôleur) et s'adresse aux plates-formes de développement d'interface utilisateur modernes (WPF et Silverlight) dans lesquelles un développeur UX a des exigences différentes de celles d'un développeur plus traditionnel. MVVM est un moyen de créer des applications clientes qui exploite les fonctionnalités de base de la plate-forme WPF, permet des tests unitaires simples des fonctionnalités de l'application et aide les développeurs et les concepteurs à travailler ensemble avec moins de difficultés techniques.

- **VUE** : Une vue est définie en XAML et ne doit avoir aucune logique dans le code-behind. Il se lie au modèle de vue en utilisant uniquement la liaison de données.
- **MODÈLE** : un modèle est chargé d'exposer les données d'une manière qui est facilement utilisable par WPF.
- **VIEWMODEL** : Un ViewModel est un modèle pour une vue dans l'application ou on peut dire comme une abstraction de la vue. Il expose les données pertinentes pour la vue et expose les comportements des vues, généralement avec des commandes.

Nous obtenons alors l'architecture simplifié suivante en appliquant le principe MVVM :



Nous rajoutons simplement des Services permettant de centraliser la logique métier des ViewModeles s'occupant d'exposer les données du modèle aux différentes vues. Les ViewModeles implémentent “INotifyPropertyChanged” pour signaler aux différents composants des vues binder les données que celle ci ont changé, permettant un refresh de l'interface.

De plus nous avons un service “MQTTService” ayant pour rôle de transmettre les données des moteurs calculées par les autres services au serveur applicatif dans le Cloud.

6.2.3- Service MQTT Client

Le service MQTT est un simple client MQTT de notre serveur applicatif. Il a pour rôle de se connecter au serveur et d'envoyer les données moteurs sur un topic. Le serveur applicatif redirigera les données vers le bon robot.

Cependant comme nous l'avons vu dans les parties précédentes, pour pouvoir utiliser le broker MQTT, le service doit premièrement d'authentifier au serveur. Au lancement il exécute alors une requête HTTP avec ses identifiants pour obtenir un jeton d'authentification qu'il enverra dans le champ username lors de la connexion avec le broker. S'il est déconnecté du broker, il vérifie l'expiration du token et si nécessaire il fera une nouvelle demande d'authentification obtenant alors un nouveau token.

Configuration MQTT

Le client MQTT possède une classe de configuration charger à partir d'un json au démarrage de l'application.

```
"MQTTServiceConfig": {  
    "TopicSendControl": "IOT/Controler/Esp32Robot",  
    "ApiURL": "http://62.35.150.64:8000/api/AuthEquipment/auth",  
    "EquipmentAuth": {  
        "IdEquipment": "WPFCcontroler",  
        "TypeEquipment": "Application",  
        "Password": "25Zjqgr8AQxxZsyz",  
        "Role": "Controler"  
    }  
}
```

Il permet de spécifier l'URL du serveur applicatif pour l'authentification, contient les identifiants d'authentification à transmettre au serveur, ainsi que le nom du topic suivi de l'identifiant du robot qu'il souhaite contrôler. Par souci de temps nous n'avons pas pu rajouter dans l'interface une sélection de contrôle pour pouvoir choisir quel robot nous souhaitons contrôler, dans ce cas il enverra les commandes au robot ayant l'identifiant “Esp32Robot”.

Évidemment pour qu'il puisse se connecter à MQTT et publier des données sur un certain topic nous devons ajouter les permissions dans la base du serveur applicatif (via le fichier de configuration serveur).

```
"MQTTAuthorizationConnectionConfigs": {  
    "AuthorizeConnection": [  
        {  
            "AuthorizeTypeEquipment": "Robot",  
            "AuthorizeRole": [ "Station" ]  
        },  
        {  
            "AuthorizeTypeEquipment": "Application",  
            "AuthorizeRole": [ "Controler" ]  
        }  
    ]  
}
```

```
{  
    "TopicName": "IOT/Controler",  
    "MaxTerms": 3,  
    "AuthorizeSending": [  
        {  
            "AuthorizeTypeEquipment": "Application",  
            "AuthorizeRole": [ "Controler" ]  
        }  
    ],  
}
```

6.2.4- Présentation du dashboard

Le dashboard est composé d'une sidebar ainsi qu'une zone d'affichage.

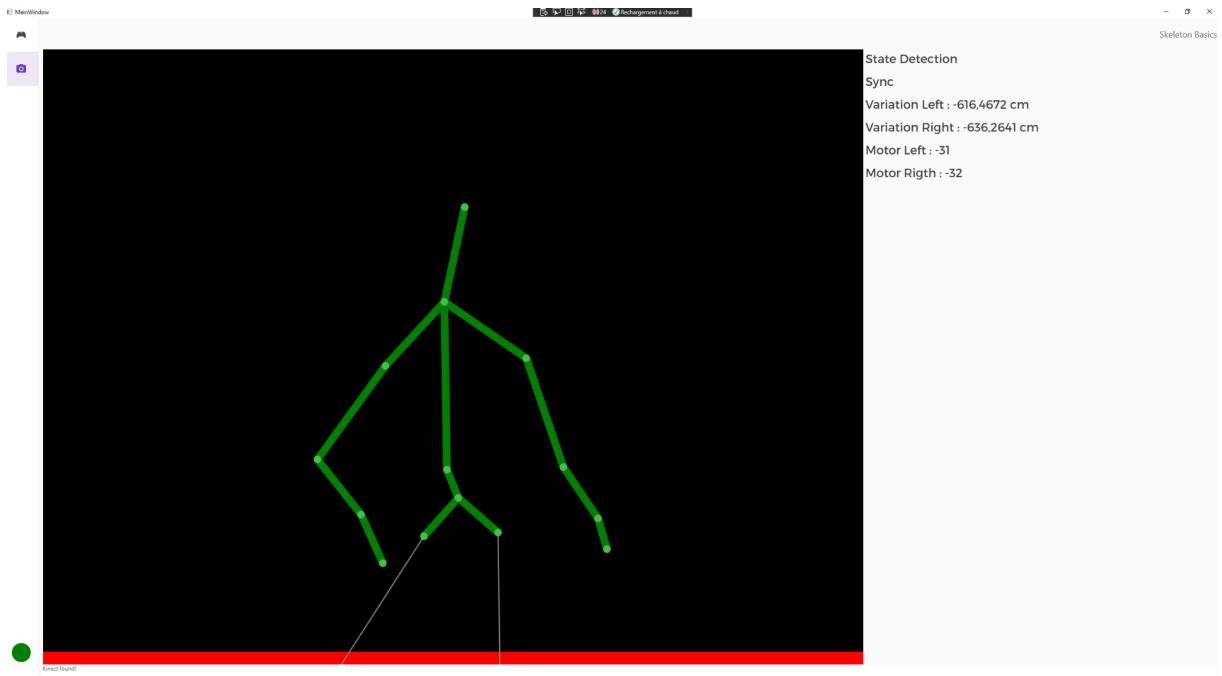
Sidebar

La sidebar est une sorte de menu permettant à l'utilisateur de switcher entre les différents modes de contrôle, manette ou kinect. Elle dispose aussi d'une pastille permettant la visualisation de la connexion MQTT avec le middleware.

Zone d'affichage

La zone d'affichage à pour but d'afficher la scène courante de l'application. Nous avons uniquement deux scènes, une pour la gestion des contrôles d'une manette, l'autre pour la Kinect.

Affichage Kinect



Affichage Manette



6.2.5- Contrôle par manette

Au lancement de la scène de la manette nous initialisons un service “ControllerService” qui aura pour rôle de définir la manette courante et par la suite à un intervalle de temps, récupérer les données de la manette pour les transformer en données exploitable pour les moteurs.

Pour actualiser les valeurs d’input de la manette nous lançons en arrière plan une tâche qui s'exécute toutes les 50 millisecondes. Nous avons défini ce temps pour ne pas trop échantillonner les données trop souvent.

Pour pouvoir interagir avec les manettes connectées le service “ControllerService” utilise l’API “XInput”.

Fonctionnement du contrôle

Nous avons choisi afin de contrôler les moteurs des robots d’associer chaque joystick de la manette à un moteur. Ainsi le joystick gauche contrôlera le moteur gauche, le joysticks droit le moteur droit. L’axe Y des joysticks représente en pourcentage la puissance des moteurs du robot. Nous signons le pourcentage de façon à obtenir en plus de la puissance, un sens de rotation, nous obtenons alors la plage de valeur [-100;100].



De plus, pour que l’utilisateur bénéficie d’une expérience cohérente, nous devons implémenter une zone morte (par exemple que les joysticks sont centrer, représenter sur l’image ci-dessus par les cercles de chaque joystick).

Les applications doivent utiliser des « zones mortes » sur les entrées analogiques (déclencheurs, bâtons) pour indiquer qu'un mouvement est suffisamment fait sur le bâton ou le déclencheur pour être considéré comme valide.

API XInput

La console Xbox utilise un contrôleur de jeu qui est compatible avec Windows. Les applications peuvent utiliser l'API XInput pour communiquer avec ces contrôleurs lorsqu'elles sont connectées à un PC Windows (jusqu'à quatre contrôleurs uniques peuvent être branchés à la fois).

À l'aide de cette API, tout contrôleur Xbox connecté peut être interrogé pour déterminer son état et les effets vibratoires peuvent être définis. Les contrôleurs qui disposent du casque peuvent également être interrogés pour les périphériques d'entrée et de sortie audio qui peuvent être utilisés avec le casque pour le traitement vocal.

Sélection de la manette

Comme nous avons pu le voir sur la présentation du dashboard de la manette, nous avons un sélecteur pour pouvoir sélectionner la manette courante car il est possible d'utiliser un total maximum de quatre manettes.

Avant chaque lecture de la manette, nous essayons aussi de détecter des changements qui concernent la déconnexion d'une manette, la connexion d'une manette.

Nous utilisons alors une méthode pour pouvoir “refresh” l'état actuel des manettes.

Pour ce faire nous récupérons :

- Les identifiants de toutes les manettes qui sont connectés actuellement (itération N).
- Les identifiants des nouvelles manettes connectés (ce sont les manettes qui ne sont pas dans la liste de manettes connectés à l'itération N-1).
- Les identifiants des manettes déconnectés (ce sont les manettes qui étaient présentes dans la liste de manettes connectés à l'itération N-1, mais qui ne sont plus présentes dans l'itération N).

Ensuite nous pouvons donc déterminer un éventuel changement d'état des manettes. Si nous détectons un changement (c'est-à-dire si de nouvelles manettes sont connectées ou déconnectées), alors nous actualisons la liste de manettes actuellement connectées, ce qui permet aussi de refresh le sélecteur de manettes dans l'interface.

Si il y avait une manette sélectionnée et qu'elle n'existe plus à l'itération actuelle alors nous la définissons la manette comme *null*, ce qui a pour effet de ne pas actualiser les valeurs de l'input dans la suite de l'échantillonnage.

Si l'utilisateur sélectionne une manette présente dans le sélecteur de l'interface, alors cela met à jour l'identifiant de la manette en cours d'utilisation dans le modèle du service.

Récupération des valeurs et transposition

Une fois que nous avons mis à jour l'état des manettes, nous pouvons déterminer à l'instant T les valeurs de la puissance des moteurs du robot dans le cas où une manette est sélectionnée par l'utilisateur.

Premièrement nous récupérons l'état de la manette sélectionnée, pour récupérer un objet "State" de l'API "XInput". A partir de cet objet là, nous récupérons la valeur actuelle pour Y et X des deux joysticks. Les valeurs de retour sont dans la plage [-32768;32767].

Une fois que nous obtenons les coordonnées de la position du joystick, nous appliquons le calcul de dead zone aux coordonnées.

- On détermine la magnitude = $\text{Math.Sqrt}((x * x) + y * y)$
- Si la magnitude < à la dead zone définie alors les coordonnées sont égales à 0
- Sinon on ajuste la magnitude en déterminant la fin de la dead zone
- Enfin nous normalisons la magnitude entre 0 et 100 (par la suite nous nous servons pas de la magnitude)

Cela calcule le vecteur de direction du contrôleur et l'avancement du vecteur que le contrôleur a fait l'objet d'un push. Cela permet l'application d'un deadzone circulaire en vérifiant simplement si l'amplitude du contrôleur est supérieure à la valeur deadzone. En outre, le code normalise l'amplitude du contrôleur, qui peut ensuite être multiplié par un facteur spécifique à un jeu pour convertir la position du contrôleur en unités pertinentes pour le jeu.

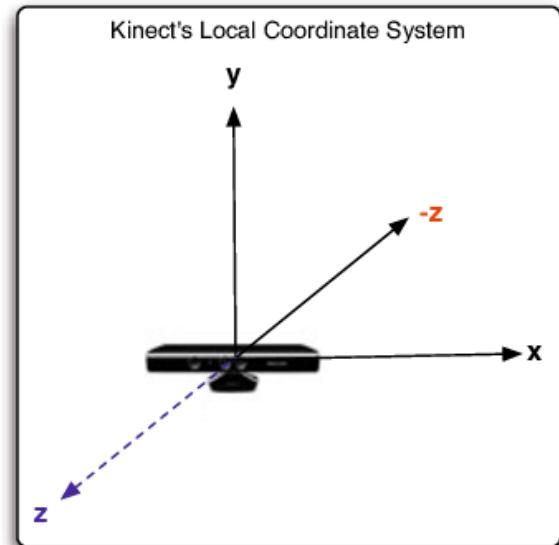
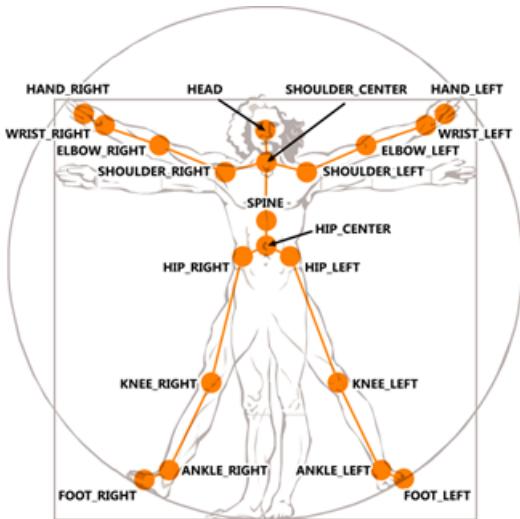
Enfin, nous comparons les valeurs de Y obtenues pour les deux joysticks à la valeur précédemment calculée. Si nous constatons une différence alors nous appliquons les nouvelles valeurs à notre objet "MoteurValue" puis l'envoyons au middleware grâce au service MQTT.

6.2.6- Contrôle par Kinect

Au lancement de la scène Kinect nous initialisons un service "KinectService" qui aura pour rôle de récupérer les frames de la Kinect pour les transformer en données exploitable pour les moteurs.

Puis nous initialisons la Kinect, nous vérifions qu'elle est bien connectée, puis nous récupérons un objet "KinectSensor". Avant de démarrer la Kinect nous lui spécifions une fonction de callback pour l'événement "SkeletonFrameReady" (événement pour signifier quand une frame est disponible) et on active les frames dédiées au squelette, puis démarrons le capteur de la Kinect.

La fonction de callback prend en paramètre un tableau d'objet Skeleton. L'objet Skeleton représente toutes les coordonnées (x,y,z) pour différents points du corps humain. Une position d'un point est comprise entre [-1;1].



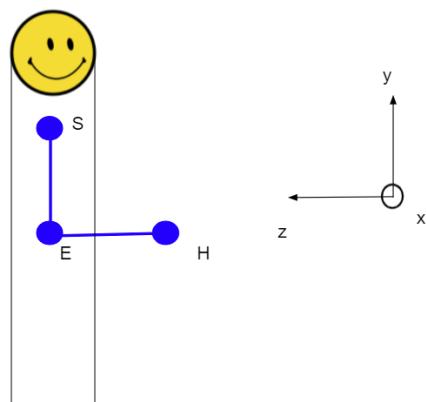
Fonctionnement du contrôle

Nous avons choisi afin de contrôler les moteurs des robots d'associer le moteur gauche au bras gauche, et le moteur droit au bras droit.

Nous avons alors comme données d'entrée les valeurs x, y, z de plusieurs points. Nous allons donc associer des mouvements à des valeurs entre [-100;100] pour les moteurs.

Pour commencer, il est nécessaire de trouver une position de synchronisation. C'est-à-dire une position qui permettra de dire à l'application "je suis prêt à contrôler le robot". Ce sera la position pour laquelle les valeurs des moteurs seront égales à 0. Ensuite nous devons trouver un mouvement nous permettant de donner un sens et une puissance aux différents moteurs.

Nous avons fixer la position suivante comme position de synchronisation (ce sont les bras d'une personne) :



Les points S,E,H sont les points qui nous intéressent dans le squelette. Avec ce schéma, nous pouvons facilement déduire les conditions pour pouvoir détecter la position :

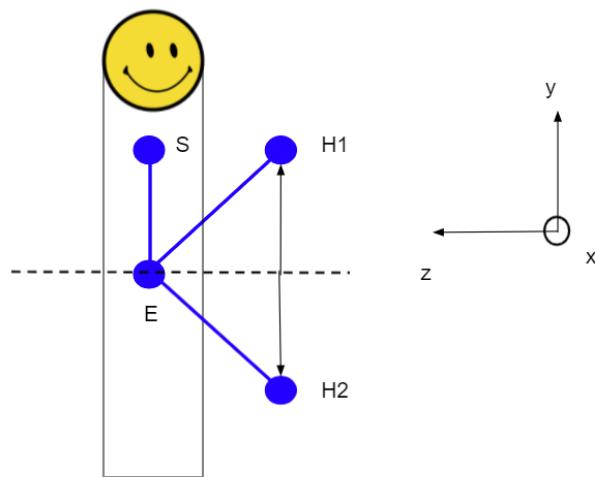
- Alignement du point S.Z et E.Z
- Alignement du point H.X et E.X
- Alignement du point H.Y et E.Y
- Alignement du point S.X et E.X

Alors si la position n'est pas au préalable dans un état synchronisé et toutes les conditions ci-dessus sont remplies, alors nous pouvons considérer que la position est synchronisée.

A partir de ce constat nous devons définir aussi des conditions qui permettra à l'application de déterminer la désynchronisation de la position :

- Plus d'alignement du point S.Z et E.Z
- Plus d'alignement du point H.X et E.X
- Plus d'alignement du point S.X et E.X

Par rapport à la détection de la première position nous voyons qu'il n'y a qu'une condition qui n'est pas présente. Effectivement ce sera le mouvement que la personne fera pour faire varier le sens et la puissance des moteurs c'est-à-dire la différence entre les points : H.Y et E.Y comme ci-dessous :



Nous avons donc maintenant, dans une position synchronisée la valeur du point H.Y et E.Y que nous devons transformer en une valeur dans la plage [-100;100].

Pour cela, nous allons calculer la différence entre les deux points. Nous obtenons alors une valeur de différence comprise entre [-1;1]. Puis nous fixons une valeur Min et Max pour la plage, c'est deux valeurs sont de la différence entre les points S.Y et E.Y. Nous obtenons alors une valeur comprise entre [-Max;Max]. Il suffit alors avec une fonction que l'on a déjà vu précédemment de rapporter cette valeur proportionnellement dans la plage [-100;100].

Traitement des frames et envois des données

Lors de la réception d'une nouvelle frame, nous ajoutons le squelette dans une queue thread-safe. Un autre thread à pour but de dépiler la queue afin de traiter les différentes frames arrivantes.

Nous avons alors dans notre application deux états :

- WAITING_SYNC
- SYNC

En fonction de l'état actuel l'application exécutera une fonction de détection de la position de synchronisation dans le cas où l'état actuel est WAITING_SYNC, sinon l'application récupérant la différences entre H.Y et E.Y afin de transposer dans la plage des moteurs.

Toutefois avant de transposer on regardera si la position actuelle est toujours synchronisée avec les conditions de l'état SYNC que nous avons vu plus haut. Si les conditions ne sont pas remplies, l'application retourne à un état de WAITING_SYNC.

Nous effectuons cela pour les deux bras, nous n'avons donc pas les points S,H,E, mais les points SL,HL,EL,SR,HR,ER. Nous obtenons alors une valeur dans la plage [-100;100] et si elles sont différentes des valeurs précédentes alors nous modifions l'objet MotorValues avec les nouvelles données et demandons au service MQTT d'envoyer les valeurs au serveur applicatif.

Dans le cas où la personne se synchronise avec l'application, c'est-à-dire à l'état WAITING_SYNC alors les valeurs des moteurs sont définies à 0 et le service MQTT envoie les valeurs au serveur pour arrêter les moteurs.

6.3- Contrôle Open CV

L'idée globale est de relier à notre robot un module d'identification de personnes et reconnaissance de celles-ci via OpenCV (en python). Le but final est de faire en sorte que le robot suive une personne.

Le manque de temps a joué en notre défaveur, car nous ne pouvions pas reconnaître une personne de dos (donc par conséquent on ne peut la suivre).

Nous devions donc reconnaître les corps des gens, hors dans les modèles déjà existant il n'y a que les "détections" de corps et pas la "reconnaissance". Il faut alors faire un apprentissage (qui prend des jours), pour créer un modèle qui reconnaîtrait les "corps" qu'il a déjà vu, donc qui apprend à reconnaître un corps aussi bien qu'il est capable de reconnaître un visage.

La deuxième possibilité était de croiser corps et visage afin de suivre le corps qui correspond au visage "cible". Cependant cela ne fonctionnait pas bien, car les corps sont difficiles à détecter, la cible ne pouvait pas être identifiée constamment à un corps.

Cette partie est donc en partie réalisée, et en partie à réaliser pour le futur.

a) OpenCV - Fonctionnement :

Initialement développée par Intel, OpenCV (Open Computer Vision) est une bibliothèque graphique. Elle est **spécialisée dans le traitement d'images**, que ce soit pour de la photo ou de la vidéo. Ce qui est totalement adapté à notre besoin de reconnaissance faciale via l'ESP 32 CAM.

Pour comprendre le fonctionnement d'OpenCV, il faut d'abord définir ce qu'est **la vision par ordinateur**.

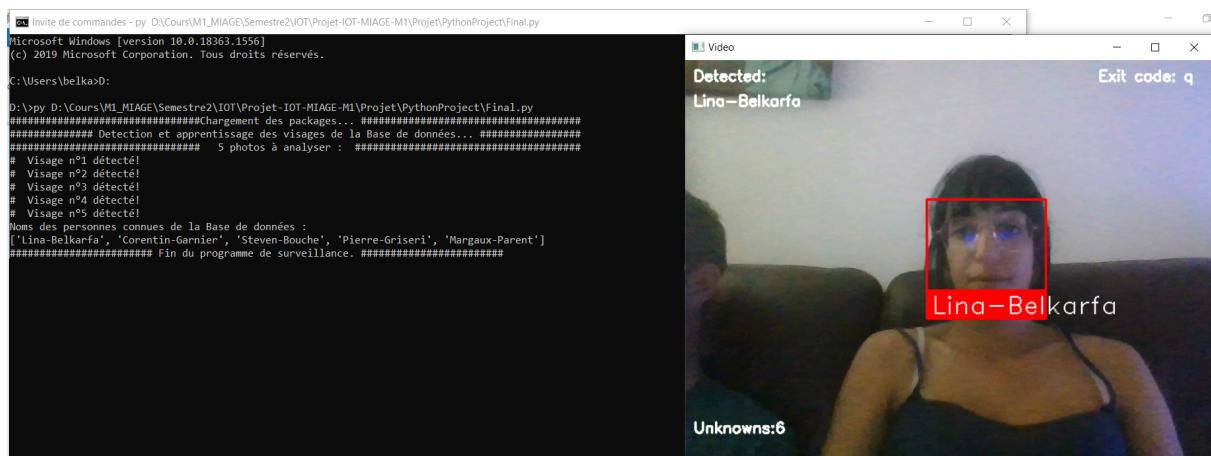
Branche de l'IA, la vision par ordinateur a pour principal but de permettre à une machine d'analyser, traiter et comprendre des images prises par un système (notre ESP32 par exemple).

OpenCV fournit un ensemble de **plus de 2500 algorithmes de vision par ordinateur, accessibles au travers d'API**. Ce qui permet d'effectuer tout un tas de traitements sur des images (extraction de couleurs, détection de visages, de formes, application de filtres,...).

Ces algorithmes se basent principalement sur **des calculs mathématiques complexes, concernant surtout les traitements sur les matrices**. Ces matrices sont en fait des **matrices de pixels**. En effet, la vidéo provenant du stream de l'ESP 32 CAM, est en fait un ensemble d'images, qui elles-mêmes sont des matrices de pixels. C'est donc ces pixels que la machine va utiliser pour l'apprentissage des visages et la détection des corps ou visages.

Il est donc important de comprendre que les principaux **régagements se feront sur les dimensions** des images, car cela va définir les dimensions des matrices. De plus, d'autres réglages doivent être fait selon les cas, la plupart étant liés à des **paramètres de sensibilité de détection**.

b) Le rendu final :



```
D:\>py D:\Cours\MIAGE\Semestre2\IOT\Projet-IOT-MIAGE-M1\Projet\PythonProject\Final.py
Microsoft Windows [version 10.0.18363.1556]
(c) 2019 Microsoft Corporation. Tous droits réservés.

C:\Users\belka>:

D:\>py D:\Cours\MIAGE\Semestre2\IOT\Projet-IOT-MIAGE-M1\Projet\PythonProject\Final.py
#####
##### Chargement des packages... #####
#####
##### Detection et apprentissage des visages de la Base de données... #####
#####
##### 5 photos à analyser : #####
#####
# Visage n°1 détecté!
# Visage n°2 détecté!
# Visage n°3 détecté!
# Visage n°4 détecté!
# Visage n°5 détecté!
Noms des personnes connues de la Base de données :
['Lina-Belkarfa', 'Corentin-Garnier', 'Steven-Bouche', 'Pierre-Griseri', 'Margaux-Parent']
##### Fin du programme de surveillance. #####
```

c) Exécution :

1) Lancement du programme avec la commande “py” :

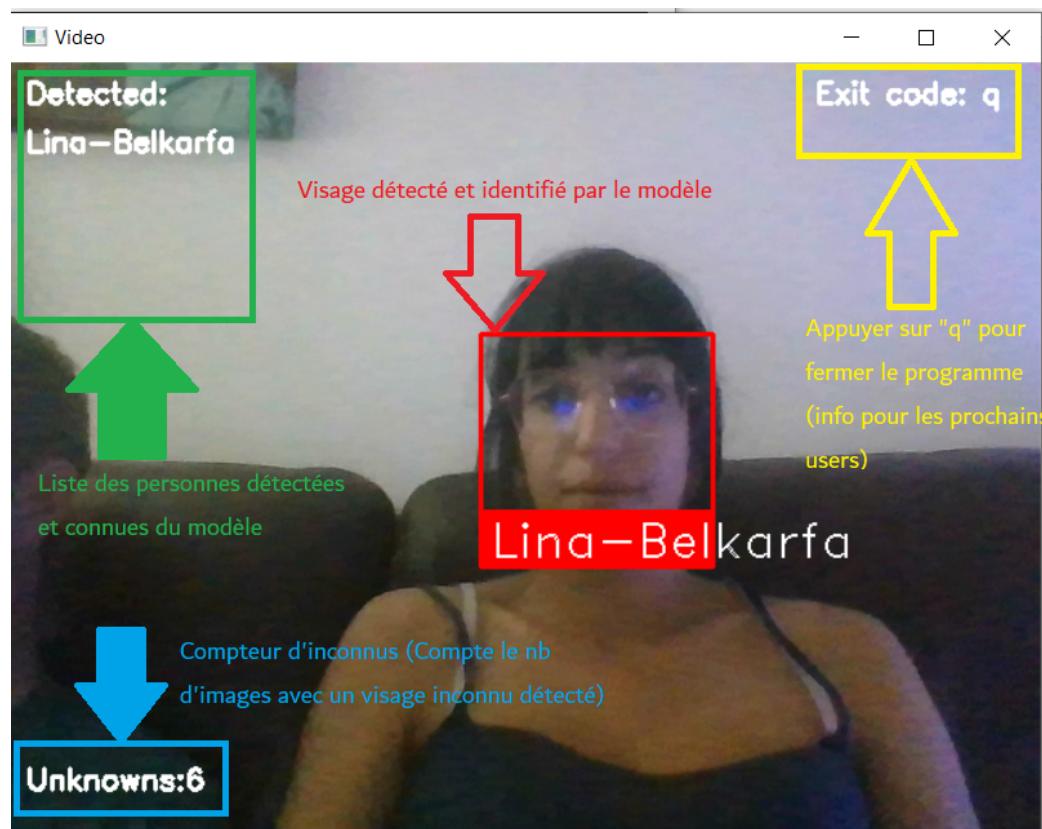
```
D:\>py D:\Cours\M1_MIAGE\Semestre2\IOT\Projet-IOT-MIAGE-M1\Projet\PythonProject\Final.py
```

2) Le programme charge les packages et images contenues dans le dossier “**Python Project**”, et c'est ainsi que l'analyse des images de personnes contenues dans le sous-dossier “**faces**” se lance. Dans la console est indiqué pour chaque image si le visage a bien été détecté. Si ce n'est pas le cas, la console affichera “pas de visage détecté”.

A la fin de ce traitement, le nom des visages connus s'affichent dans la console :

```
#####Chargement des packages... #####
##### Detection et apprentissage des visages de la Base de données... #####
##### 5 photos à analyser : #####
# Visage n°1 détecté!
# Visage n°2 détecté!
# Visage n°3 détecté!
# Visage n°4 détecté!
# Visage n°5 détecté!
Noms des personnes connues de la Base de données :
['Lina-Belkarfa', 'Corentin-Garnier', 'Steven-Bouche', 'Pierre-Griseri', 'Margaux-Parent']
##### Fin du programme de surveillance. #####
```

3) Enfin, une fenêtre s'ouvre avec le stream et la détection et identification en direct des visages :



d) Explication du code :

La plupart du code est commenté, mais voyons en détail comment ce script fonctionne. Tout d'abord, il faut faire une pré-installation grâce à la commande “pip install” du package “face_recognition”, et bien sûr “opencv-python”. Ces installations ne se font qu'une fois, contrairement aux “import” qui se lanceront à chaque exécution.

```
#Installation à faire au préalable :  
#pip install face_recognition  
#pip install opencv-python  
  
import face_recognition  
import cv2  
import numpy as np  
import os  
import glob  
import operator  
  
#Chargement du modèle pour détection de visage  
  
faceCascade = cv2.CascadeClassifier('C:/Users/belka/Documents/IOT_PROJECT_DATA/haarcascade_frontalface_default.xml')
```

Par la suite, nous récupérons les photos que nous souhaitons faire connaître aux modèles :

```
from cv2 import *  
  
#Prendre les photos des personnes connues du système  
  
faces_encodings = []  
faces_names = []  
  
list_of_files = ['D:/Cours/M1_MIAGE/Semestre2/IOT/Projet-IOT-MIAGE-M1/Projet/PythonProject/data/faces/Lina-Belkarfa.jpg',  
'D:/Cours/M1_MIAGE/Semestre2/IOT/Projet-IOT-MIAGE-M1/Projet/PythonProject/data/faces/Corentin-Garnier.jpg',  
'D:/Cours/M1_MIAGE/Semestre2/IOT/Projet-IOT-MIAGE-M1/Projet/PythonProject/data/faces/Steven-Bouche.jpg',  
'D:/Cours/M1_MIAGE/Semestre2/IOT/Projet-IOT-MIAGE-M1/Projet/PythonProject/data/faces/Pierre-Griseri.jpg',  
'D:/Cours/M1_MIAGE/Semestre2/IOT/Projet-IOT-MIAGE-M1/Projet/PythonProject/data/faces/Margaux-Parent.jpg']  
names = list_of_files.copy()  
number_files = len(list_of_files)  
cur_direc = 'D:/Cours/M1_MIAGE/Semestre2/IOT/Projet-IOT-MIAGE-M1/Projet/PythonProject/data/faces/'
```

Il est important de nommer les photos aux noms des personnes qui correspondent. Cela permettra d'utiliser le nom de l'image pour associer la reconnaissance à une identité. `cur_direc` permet de retirer le chemin complet de l'image afin de n'avoir que le nom qui apparaît.

La partie suivante du script permet de détecter le visage présent dans chaque image, et de mémoriser les caractéristiques de chacun d'entre eux.

```

#Chercher les visages dans les photos

print('##### Detection et apprentissage des visages de la Base de données... ##### ')
print('##### '+str(number_files)+' photos à analyser : #####')
for i in range(number_files):

    globals()['image_{}'.format(i)] = face_recognition.load_image_file(list_of_files[i])

    if not(face_recognition.face_encodings(globals()['image_{}'.format(i)])):
        print('Pas de visage détecté!')
        names[i] = names[i].replace(cur_direc, "No detected ")

    else:
        globals()['image_encoding_{}'.format(i)] = face_recognition.face_encodings(globals()['image_{}'.format(i)]) [0]
        faces_encodings.append(globals()['image_encoding_{}'.format(i)])

        names[i] = names[i].replace(cur_direc, "")
        names[i] = names[i].replace(".jpg", "")
        faces_names.append(names[i])
        print('# Visage n°'+str(i+1)+ ' détecté!')


#Voir les noms des visages reconnus
print('Noms des personnes connues de la Base de données :\n'+str(names))
names

```

Nous avons mis en place une boucle dans laquelle l'algorithme “face_recognition”, s'il y a un visage dans l'image, va enregistrer ses caractéristiques et renvoyer “Visage détecté”. Dans le cas contraire, on renvoie “pas de visage détecté”. Enfin, nous renvoyons le nom des personnes détectées.

Par la suite, nous avons la vidéo capture qui se lance. Nous avons été contraints de développer notre script à partir de nos webcam, car le travail à distance nous a empêché de travailler quand nous le voulions sur le robot. C'était très difficile, notamment avec la caméra qui était un outil important de notre rendu. Le robot en lui-même était également nécessaire pour la suite (déplacement en fonction de la personne qui se déplace).

Voici des détails un peu plus parlant du code :

```

face_locations = []
face_encodings = []
face_names = []
process_this_frame = True

#Lancement du programme de reconnaissance faciale :
video_capture = cv2.VideoCapture(0) Permet le lancement de la caméra et du stream que l'on stock dans une variable. Partie à remplacer par le lancement de l'ESP32 CAM

detection=[]
nbInconnus=0
name=""

while True:
    ret, frame = video_capture.read() Boucle dans laquelle tout le script de reconnaissance faciale s'exécute

    small_frame = cv2.resize(frame, (0, 0), fx=0.25, fy=0.25)
    rgb_small_frame = small_frame[:, :, ::-1] Traitement des données pour face_recognition

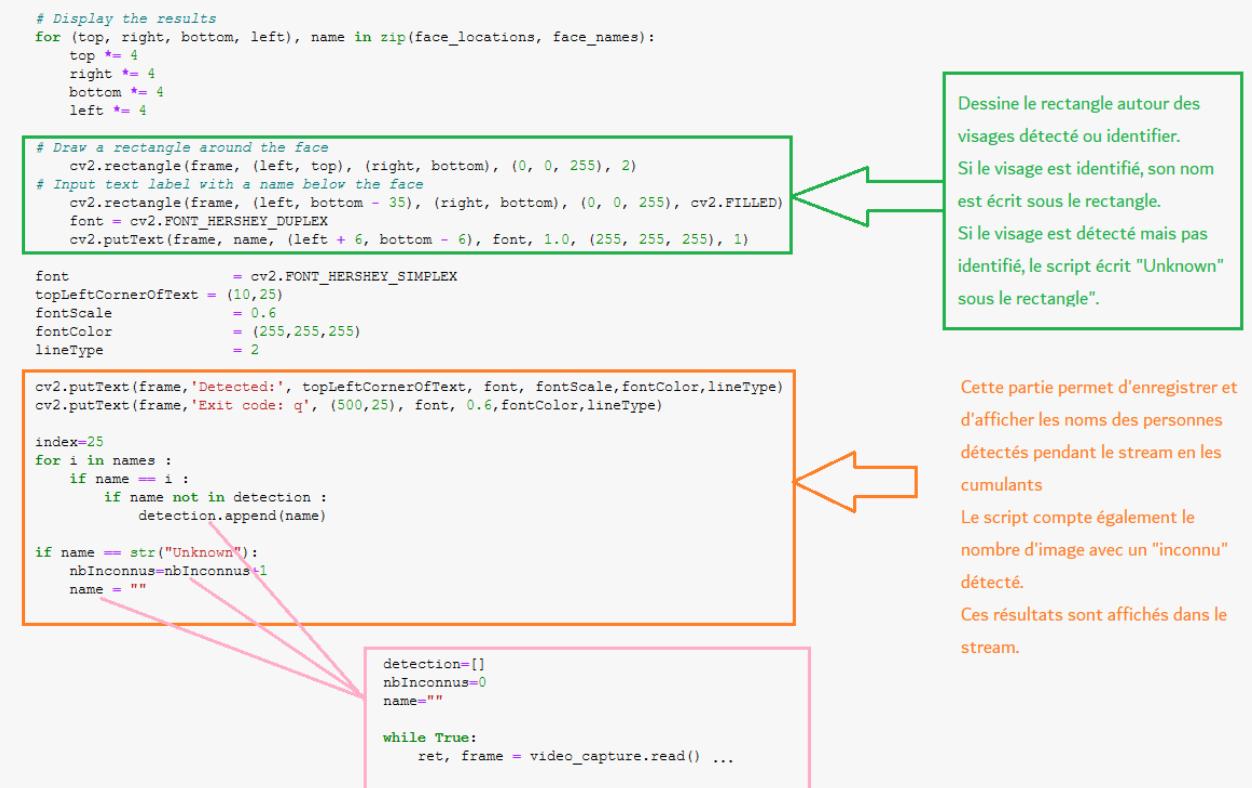
    if process_this_frame:
        face_locations = face_recognition.face_locations(rgb_small_frame)
        face_encodings = face_recognition.face_encodings(rgb_small_frame, face_locations)

```



Les réglages restent importants sur la partie traitement des données des images. C'est avec l'ESP32 CAM qu'il faut réaliser les réglages. De plus, pour la sensibilité à la détection, cela dépendra également du type d'image, du nombre de pixels etc...

Ci-dessous, le code tel qu'il est réglé pour la caméra de l'ordinateur de Lina. N'importe qui peut le lancer, mais son efficacité sera moins bien car les réglages ont été faits depuis l'ordinateur portable.



```

detec=""
for i in detection :
    index=index+30
    cv2.putText(frame,i, (10,index), font, fontScale,fontColor,lineType)
    cv2.putText(frame,"Unknowns:"+str(nbInconnus), (10,450), font, fontScale,fontColor,lineType)

# Display the resulting image
cv2.imshow('Video', frame)
# Hit 'q' on the keyboard to quit!
if cv2.waitKey(1) & 0xFF == ord('q'):
    break

cv2.destroyAllWindows()

print('##### Fin du programme de surveillance. #####')

```

Pour chaque nom des personnes détectés, ce script va les écrire sur le stream, ainsi que le nombre d'inconnus

Le script est donc fonctionnel pour la reconnaissance faciale, la détection de visages, et la détection d'inconnus.

Pour qu'il puisse être utilisé en suivie de personne, il nous faut nous connecter à la websocket du serveur applicatif, appliquer l'algorithme python sur l'image en temps réelle, et en sortie avoir une connection au serveur MQTT qui retourne une valeur pour chaque moteur (entre -100 et 100). Cette valeur fera avancer ou reculer le robot.

La valeur entre -100 et 100 retournée peut être calculée de la manière suivante grâce aux coordonnées fournies par OpenCV avec "face_locations".

7- Interface Web de visualisation

7.1- Présentation

Nous avons réalisé une interface web permettant de visualiser les données des équipements en temps réel (en l'occurrence les ESP32). Celle-ci nous permet d'afficher les données suivantes :

- température de l'environnement,
- luminosité de l'environnement,
- caméra en direct du robot,
- localisation du robot via une carte,
- ESP connectés en direct,
- utilisation des moteurs de l'ESP en direct.

Cette interface sert alors de dashboard, accessible à l'adresse suivante : <https://51.38.226.68/dashboard>

7.2- Environnement de développement

Le site web a été codé en TypeScript sous le framework Angular 11. Nous avons choisi cette technologie pour plusieurs raisons ; d'abord, Angular est souvent de pair avec

.NET (techno back-end) et il s'est avéré que nous utilisions déjà .NET, rendant Angular plus attrayant. Aussi certaines personnes de l'équipe, fort de leur expérience, ont de solides bases en Angular. Pour ces deux raisons, Angular apparaissait alors comme le meilleur choix et de fait ce framework nous a permis de coder efficacement et facilement notre interface utilisateur. Finalement, cette interface a été réalisée avec l'IDE Visual Studio Code.

Il est intéressant de noter que l'application permet une configuration facile pour changer d'environnement de déploiement ; nous avons plusieurs adresses API en fonction de l'environnement (local, dev, prod). Nous avons alors mis en place des scripts permettant de générer les fichiers compilés avec l'adresse que l'on souhaite (local, localSSL, dockerSSL, dockerSslProd).

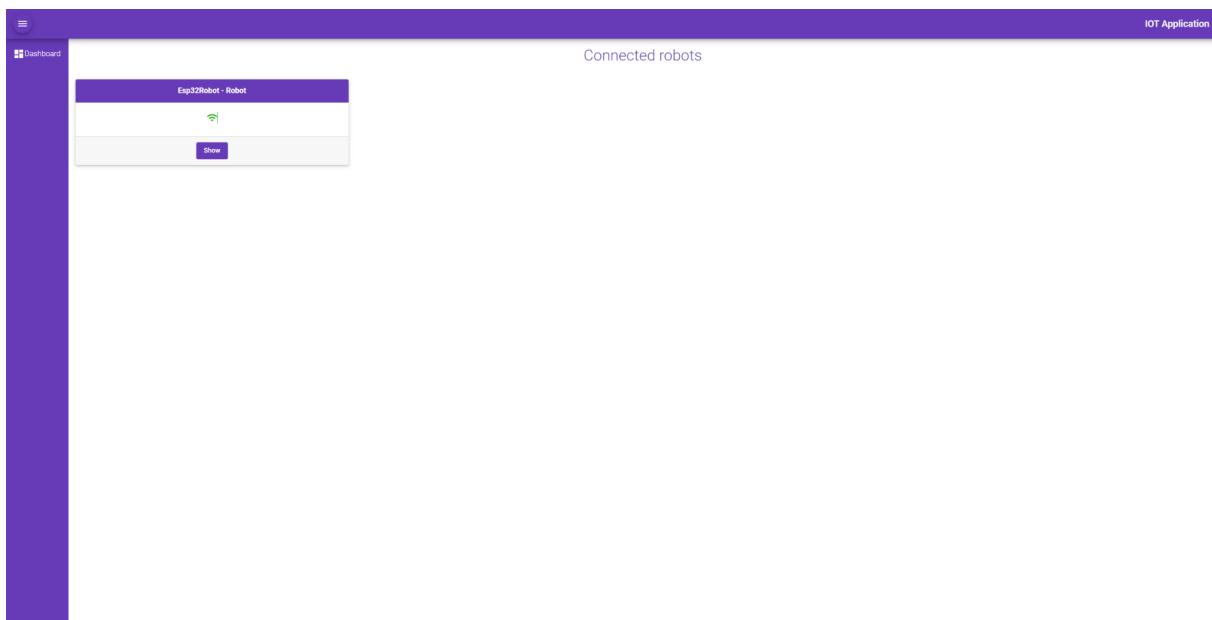
Nous voyons alors que nos scripts permettent quatre types d'exécution :

- local : permettant de lancer l'environnement dev en HTTP
- localSsl : permettant de lancer l'environnement dev en HTTPS
- dockerSsl : permettant l'exécution dans un environnement Docker avec HTTPS en dev
- dockerSslProd : permettant l'exécution dans un environnement Docker avec HTTPS en prod

Ces scripts facilitent la configuration de l'application, la rendant maniable sans toucher au code.

7.3- Dashboard principal

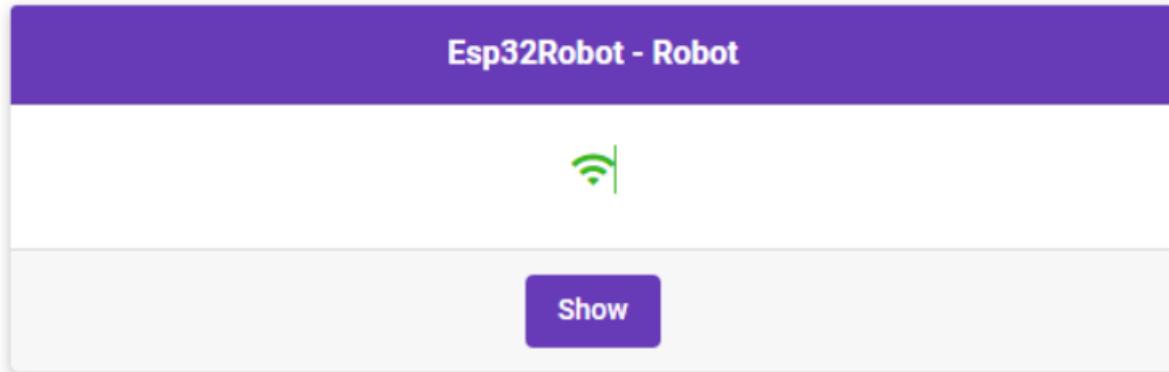
Comme mentionné précédemment, le front sert de Dashboard. Nous avons conçu celui lui de la manière suivante :



Dashboard Angular - Page d'accueil

Le dashboard sert exclusivement à deux choses : voir en temps réel les ESP connectés et accéder au détail d'un équipement (un à la fois) en appuyant sur le bouton “Show”. Le détails de l'équipement sélectionné est expliqué à la partie suivante (7.4)

Chaque équipement est représenté dans cette page comme ceci :



Affichage d'un ESP dans la page d'accueil

Du haut vers le bas nous avons : le type d'équipement en titre, la connectivité symbolisée par l'icône WiFi (si l'ESP est connecté ou non) et le bouton “Show” permettant d'accéder au détail de l'équipement.

A savoir que l'exemple ci-dessus n'est pas exhaustif, le front peut afficher plusieurs équipements.

Pour que le front puisse détecter en temps réel si des équipements sont connectés ou non nous avons utilisé le protocole WebSocket nous permettant d'être averti pour chaque connexion/déconnexion. Pour aller plus loin, tous les appels réseaux de ce front sont exclusivement sous le protocole WebSocket. En effet, le front affiche les équipements et leurs informations en temps réel, ce que nous verrons dans la partie suivante.

7.4- Visualisation d'un équipement

Après l'appuie sur le bouton “Show”, cette page se présente :

The screenshot shows a dashboard titled "IOT Application" with the sub-section "Détails : Esp32Robot - Robot". The interface is divided into several panels:

- Latest Data:** A table showing current values for Temperature (25), Ligth (1149), Latitude (0), Longitude (0), and Timestamp (12/06/2021, 20:01:40).
- Services:** A table listing a single connection: MQTT with IdConnection 24:0A:C4:61:91:F4, Role Station, and AddressIp 172.19.0.1:53372.
- Geolocation:** A panel containing a Google Maps embed which failed to load, displaying an error message: "Impossible de charger Google Maps correctement sur cette page." and "Ce site Web vous appartient ? OK".
- Stream:** A video stream preview showing a circular pattern of colored bars (red, green, blue) on a black background.
- Line chart Light:** A line graph showing Light levels over time, with values fluctuating between 1000 and 1200.
- Line chart Temperature:** A line graph showing Temperature levels over time, with values fluctuating between 15 and 25.
- Bar chart Motor:** A bar graph showing Motor activity levels, with values ranging from 0 to 100.

Détails d'un équipement - Vue globale

Ceci représente l'ensemble des données pouvant être collectées par l'interface. Nous allons nous intéresser à chacune des parties de cette page, en détaillant le rôle de chacune d'elles. Il faut rappeler que **toutes ces données sont récupérées via des WebSocket**.

This screenshot shows the same "Latest Data" and "Services" sections as the previous one, but the "Geolocation" and "Stream" sections are missing or not visible.

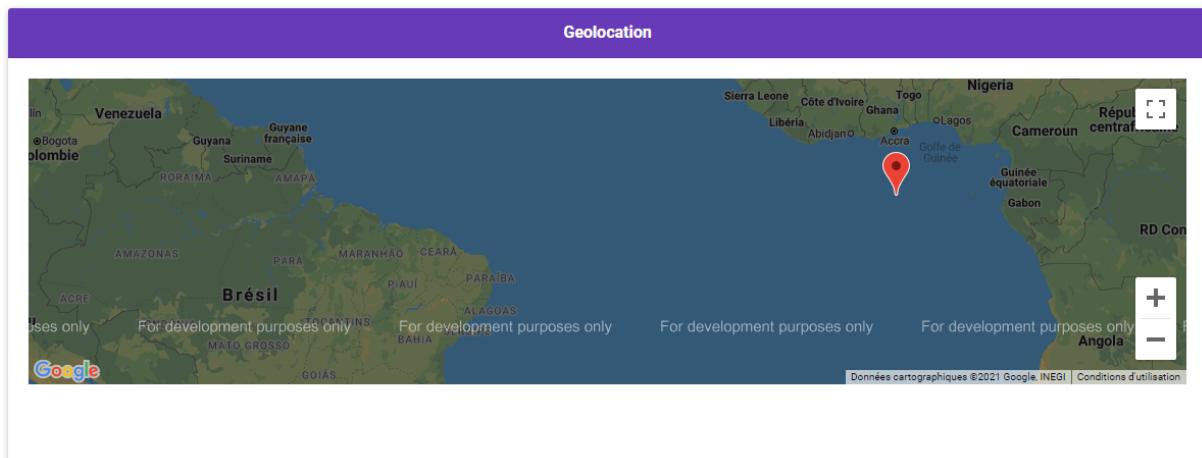
Category	Value
Temperature	25
Ligth	1145
Latitude	0
Longitude	0
Timestamp	12/06/2021, 20:02:16

Name	IdConnection	Role	AddressIp
MQTT	24:0A:C4:61:91:F4	Station	172.19.0.1:53372

Détails d'un équipement - Dernières données et services actifs

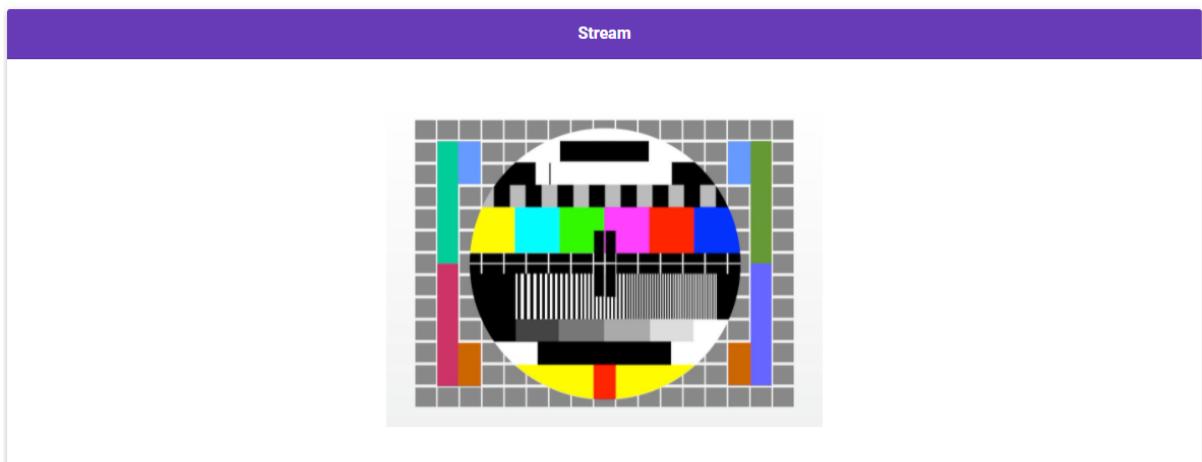
Commençons par les onglets “Latest Data” et “Services” :

- L'onglet “Latest Data” affiche un tableau des dernières données reçues toutes catégories confondues (température, luminosité, latitude, longitude, date de dernière modifications). Il permet d'avoir une vue synthétique des dernières fluctuations de l'environnement du robot.
- L'onglet “Services” affiche aussi un tableau mais représentant les différentes connexions du robot (MQTT pour quasiment l'ensemble et TCP uniquement pour la caméra).



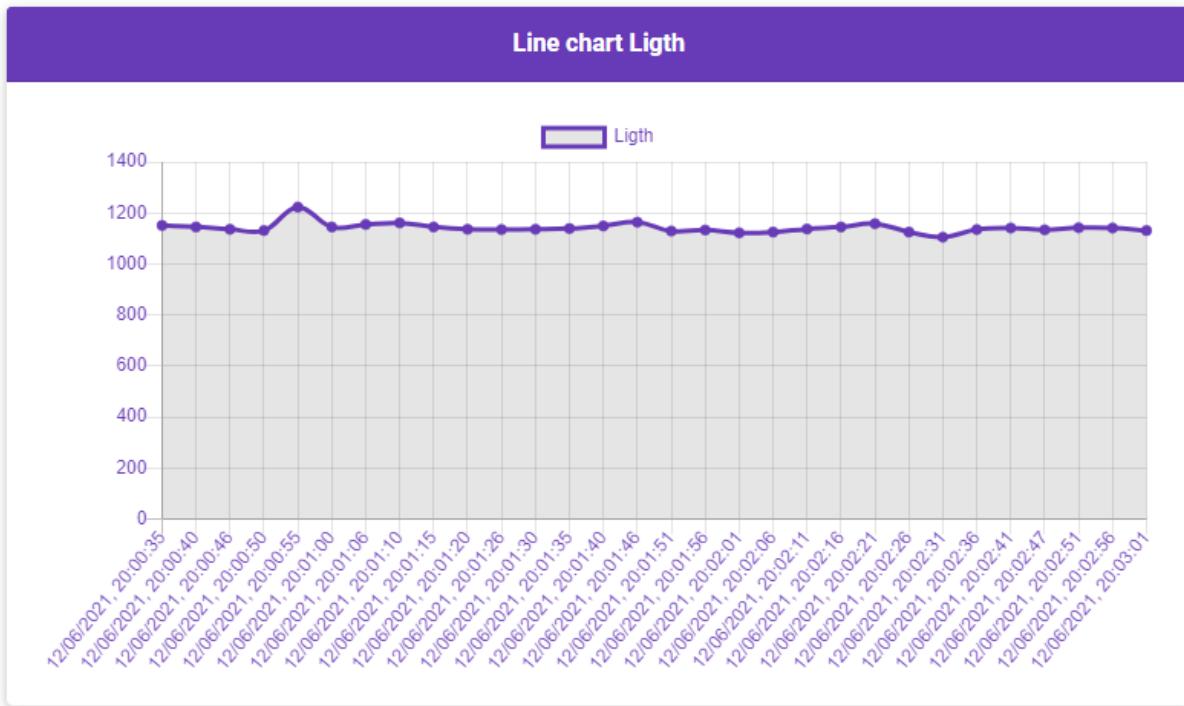
Détails d'un équipement - Géolocalisation

Ici, nous avons implémenté une carte indiquant où se trouve l'équipement en utilisant la latitude et la longitude de celui-ci. A noter qu'ici sur l'écran, le capteur n'était pas en place sur l'ESP, voilà pourquoi le point est en pleine mer.



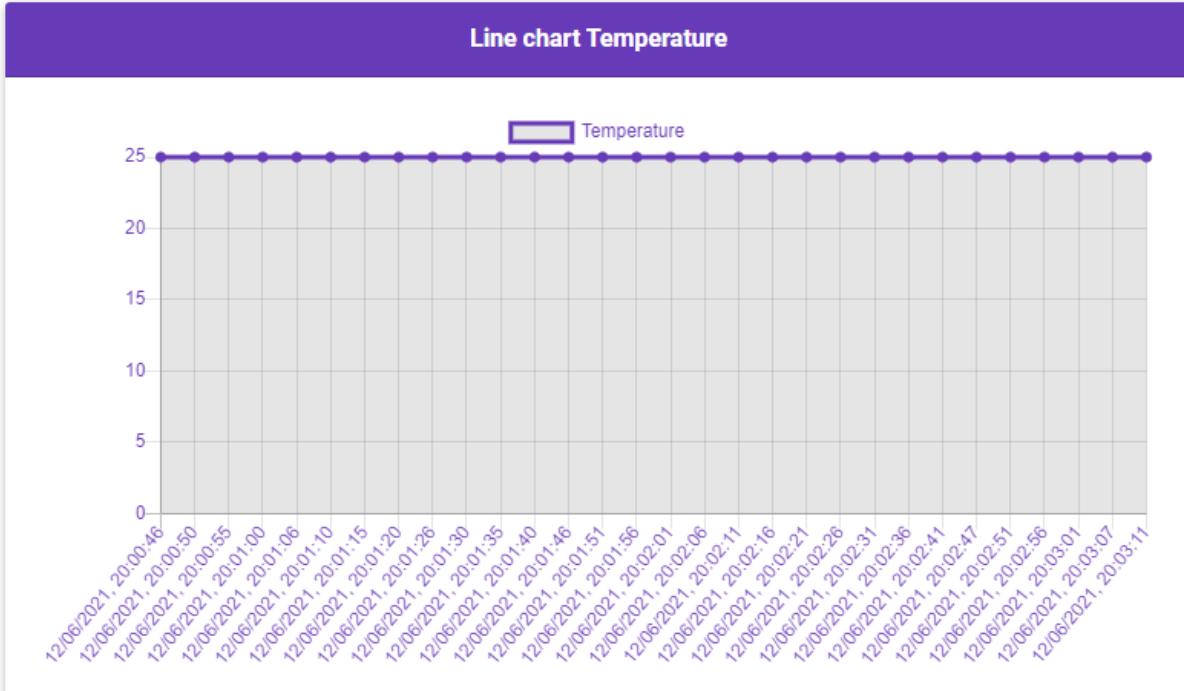
Détails d'un équipement - Flux vidéo

Cet onglet sert à afficher ce que la caméra de l'ESP capture, toujours au travers de la WebSocket. Il y a un peu de retard évident sur la caméra puisque le front est hébergé en ligne, pour des raisons de trafic donc.



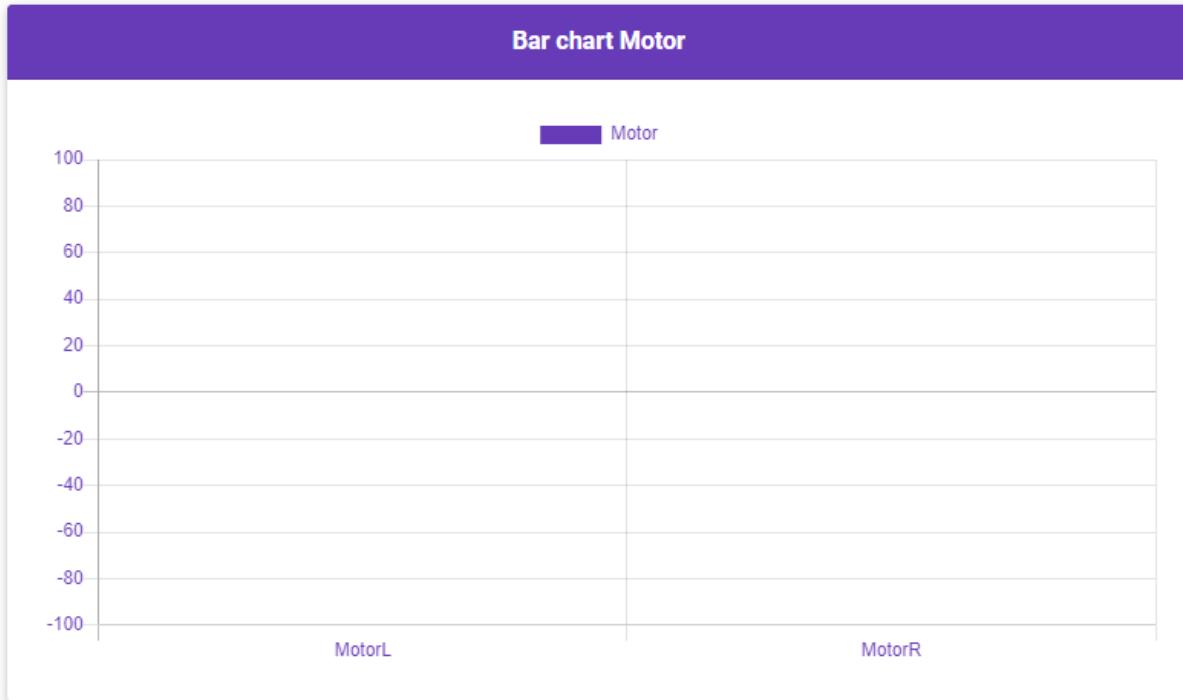
Détails d'un équipement - Graphique du taux de luminosité dans le temps

Ici nous pouvons observer le taux de luminosité toutes les 5 millisecondes, toujours en temps réel.



Détails d'un équipement - Graphique de la température ambiante dans le temps

Cet écran, similaire au précédent, affiche la température en degré Celsius toutes les 5 millisecondes.



Détails d'un équipement - Graphique de l'utilisation en temps réel des moteurs du robot

Cet onglet permet de voir en temps réel l'utilisation des moteurs (droite et gauche) de l'ESP.

8- Sécurité

Nous avons mis en place plusieurs mécanismes de sécurité pour ne pas mettre en péril nos services, mais aussi nos données transitant dans le Cloud.

Nous n'avons pas implémenter une parfaite sécurité par soucis de temps, mais il y toujours des améliorations à faire.

8.1- SSL/TLS pour HTTPS

Transport Layer Security (TLS), le successeur du désormais obsolète Secure Sockets Layer (SSL), est un protocole cryptographique conçu pour assurer la sécurité des communications sur un réseau informatique. Plusieurs versions du protocole sont largement utilisées dans des applications telles que la messagerie électronique , la messagerie instantanée et la voix sur IP , mais son utilisation en tant que couche de sécurité dans HTTPS reste la plus visible publiquement.

Le protocole TLS vise principalement à assurer la confidentialité et l'intégrité des données entre deux ou plusieurs applications informatiques communicantes. Il s'exécute dans la couche applicative d'Internet et est lui-même composé de deux couches : l'enregistrement TLS et les protocoles de prise de contact TLS.

Ce qui semble simple en théorie est plus compliqué dans la réalité. Le problème de base est que le serveur doit indiquer la clé au client et cela avant que la communication avec le TLS soit sécurisée. Toute personne qui envoie des pièces jointes chiffrées connaît bien ce problème : vous chiffrerez un fichier et devez alors communiquer au destinataire le mot de passe secret, par exemple par téléphone.

Le protocole TLS utilise la procédure suivante pour résoudre ce problème :

1. Lorsque le client, par exemple un navigateur Internet, contacte le serveur Web, celui-ci envoie d'abord son certificat au client. Ce certificat SSL prouve que le serveur est authentique et qu'il ne dissimule pas une fausse identité.
2. Le client vérifie la validité du certificat et envoie un numéro aléatoire au serveur, chiffré avec la clé publique (Public Key) du serveur.
3. Le serveur utilise ce numéro aléatoire pour générer la clé de session (Session Key), qui est utilisée pour chiffrer la communication. Comme le numéro aléatoire provient du client, celui-ci peut être sûr que la clé de session émane effectivement du serveur adressé.
4. Le serveur envoie la clé de session au client sous forme chiffrée. Ce chiffrement est effectué au moyen de l'échange de clés Diffie-Hellmann.
5. Les deux parties peuvent maintenant envoyer leurs données de manière sécurisée avec la clé de session.

La raison pour laquelle le chiffrement asymétrique n'est utilisé que pour la transmission de la clé de session (mais pas pour le chiffrement des flux de données eux-mêmes) est la question de la vitesse ; le chiffrement asymétrique est relativement lent et retarderait donc significativement la communication des données.

Nous avons donc généré nos propres certificats en tant que notre propre autorité de certification et nos propres clés et certificats serveurs signé par notre autorité de certification. Malgré le fait que dans la réalité il faudrait bien évidemment utiliser une vraie autorité de certification pour certifier que nous sommes bien ceux que nous prétendons être.

Pour cela nous avons utilisé OpenSSL. OpenSSL est une bibliothèque logicielle pour les applications qui sécurisent les communications sur les réseaux informatiques contre les écoutes clandestines ou doivent identifier la partie à l'autre extrémité. Il est largement utilisé par les serveurs Internet , y compris la majorité des sites Web HTTPS .

OpenSSL contient une implémentation open source des protocoles SSL et TLS . La bibliothèque principale , écrite en langage de programmation C , implémente des fonctions cryptographiques de base et fournit diverses fonctions utilitaires. Des wrappers permettant l'utilisation de la bibliothèque OpenSSL dans une variété de langages informatiques sont disponibles.

Pour commencer nous générerons la clé privée de notre CA :

```
openssl genrsa -des3 -out rootCA.key 2048
```

Puis nous générerons le certificat du CA avec notre clé privée :

```
openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.crt
```

Nous générerons la clé privée pour notre serveurs :

```
openssl genrsa -out privateKey.key 2048
```

Et nous générerons le CSR (Corporate Social Responsibility) avec un fichier de configuration :

```
openssl req -new -sha256 -key privateKey.key -config openssl.cnf -out certificate.csr
```

Nous pouvons alors ensuite générer le certificat serveur signé par notre CA :

```
openssl x509 -req -in certificate.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -out certificat.crt -days 3650 -sha256
```

Pour finir nous générerons un fichier PFX contenant tous nos objets concernant le serveur :

```
openssl pkcs12 -export -out certificate.pfx -inkey privateKey.key -in certificat.crt
```

Nous obtenons donc à la fin un certificat SSL/TLS pour nos serveurs approuvé par notre autorité de certification. Il sera utilisé dans nos applications pour permettre l'utilisation du protocole HTTPS pour l'API web, ainsi que la communication temps réel avec SignalR. Nous l'utilisons aussi pour notre service MQTT.

Cependant, nous n'avons pas eu le temps d'implémenter TLS pour notre serveur TCP.

Enfin nous n'avons pas eu le temps de mettre en place l'utilisation de HTTPS pour les requêtes effectuées par l'ESP pour l'authentification.

8.2- Identification, Authentification et autorisation

En sécurité des systèmes d'information, la Gestion des Identités et des Accès (GIA) (en anglais Identity and Access Management : IAM) est l'ensemble des processus mis en œuvre par une entité pour la gestion des habilitations de ses utilisateurs à son système d'information ou à ses applications. Il s'agit donc de gérer qui a accès à quelle information à travers le temps. Cela implique ainsi d'administrer la création, la modification, et les droits d'accès de chaque identité numérique interagissant avec les ressources de l'entité.

Les composantes de la GIA peuvent être divisées en 4 catégories distinctes, qui fonctionnent conjointement.

- L'identification, initialisant l'identité de l'utilisateur
- L'authentification, de l'utilisateur au système
- L'autorisation, de l'utilisateur à l'accès de la ressource
- La gestion de l'utilisateur

Toutes les parties identifications, authentification, et autorisation ont été décrites fonctionnellement dans les parties précédentes nous faisons alors qu'un bref rappel des notions.

Identification

Certaines applications informatiques ou certains services en ligne n'étant pas publics, mais réservés à des utilisateurs bien particuliers, il est nécessaire pour eux de passer par une procédure d'identification, au cours de laquelle ils fourniront un identifiant, représentation de leur identité.

Cette étape initie la relation entre l'utilisateur et les ressources informationnelles du système. Elle permet d'attribuer à l'utilisateur un ensemble de paramètres qui le caractérisent de façon unique, rassemblé sur une fiche identité, propre à chaque utilisateur comportant également les droits d'accès qui sont associés à la fois à l'utilisateur et au contexte de son utilisation des ressources de l'entité.

Authentification

Une fois qu'une identité a été associée à l'utilisateur, il est nécessaire de la vérifier afin de confirmer son individualité. L'authentification prend généralement la forme d'un couple identifiant/mot de passe et permet de créer une session entre l'utilisateur et le système afin de permettre à celui-ci d'interagir avec la ressource jusqu'à la fin de la session (déconnexion, expiration).

L'authentification des utilisateurs prend de plus en plus d'importance en Gestion des identités et des accès afin de limiter les fraudes et s'assurer de l'identité d'une personne tentant d'accéder à un système. L'identification et l'authentification permettent d'empêcher les intrusions potentiellement source de corruption des données informatiques de l'entité.

Autorisation

Après l'authentification et avant l'accès de l'utilisateur à la ressource visée, il est nécessaire de vérifier son droit à y accéder. La requête de l'utilisateur est alors mise en lien avec ses habilitations et détermine si la ressource lui est accessible ou non. La nature du droit d'accès est alors déterminée et associée à l'identité de l'utilisateur. En général, les types d'accès sont la consultation, la modification, l'exécution d'un programme, ou son administration.

Gestion utilisateur

Cette fonction rassemble la gestion et la modification des utilisateurs, des groupes et des rôles. Le cycle de vie entier de l'utilisateur, de la création à la suppression de son identité au sein du système, est alors contrôlé.

9- Déploiement de l'application

Pour que l'application soit disponible dans le Cloud, nous avons fait le choix d'utiliser un serveur dans le Cloud.

Ce serveur est de type VPS (serveur privé virtuel). C'est une machine virtuelle qui exécute sa propre copie d'un système d'exploitation, et nous avons accès au super utilisateur du système via SSH. Nous avons choisi d'utiliser le système d'exploitation Ubuntu 20.1.

La force motrice de la virtualisation des serveurs est similaire à celle qui a conduit au développement du temps partagé et de la multiprogrammation dans le passé. Bien que les ressources soient toujours partagées, comme dans le modèle de partage du temps, la virtualisation offre un niveau de sécurité plus élevé, en fonction du type de virtualisation utilisé, car les serveurs virtuels individuels sont pour la plupart isolés les uns des autres et peuvent exécuter leur propre système d'exploitation qui peut être redémarré indépendamment en tant qu'instance virtuelle.

Le partitionnement d'un seul serveur en plusieurs serveurs est de plus en plus courant sur les micro-ordinateurs depuis le lancement de VMware ESX Server en 2001. Le serveur physique exécute généralement un hyperviseur chargé de créer, libérer et gérer les ressources des systèmes d'exploitation « invités », ou des machines virtuelles. Ces systèmes d'exploitation invités se voient allouer une part des ressources du serveur physique, généralement de manière à ce que l'invité n'ait connaissance d'aucune autre ressource physique, à l'exception de celles qui lui sont allouées par l'hyperviseur.

Comme un VPS exécute sa propre copie de son système d'exploitation, les clients ont un superutilisateur-accès de niveau à cette instance de système d'exploitation, et peut installer presque tous les logiciels qui s'exécutent sur le système d'exploitation ;

cependant, en raison du nombre de clients de virtualisation s'exécutant généralement sur une seule machine, un VPS dispose généralement d'un temps processeur, d'une RAM et d'un espace disque limités

Le type d'hébergement est autogéré, nous permettant d'administrer nous-même notre instance du serveur. Bien évidemment, il est nécessaire d'avoir des connaissances dans l'administration de serveurs Linux dans notre cas, mais cela nous permet un meilleur contrôle sur notre processus de déploiement.

9.1- Docker

Docker est un ensemble de produits de plate - forme en tant que service (PaaS) qui utilisent la virtualisation au niveau du système d'exploitation pour fournir des logiciels dans des packages appelés conteneurs. Les conteneurs sont isolés les uns des autres et regroupent leurs propres logiciels, bibliothèques et fichiers de configuration ; ils peuvent communiquer entre eux par des canaux bien définis. Étant donné que tous les conteneurs partagent les services

d'un seul noyau de système d'exploitation , ils utilisent moins de ressources que les machines virtuelles .

Docker peut empaqueter une application et ses dépendances dans un conteneur virtuel pouvant s'exécuter sur n'importe quel ordinateur Linux, Windows ou macOS. Cela permet à l'application de s'exécuter dans divers emplacements, tels que sur site , dans un cloud public et/ou dans un cloud privé. Lors de l'exécution sur Linux, Docker utilise les fonctionnalités d'isolation des ressources du noyau Linux (telles que les groupes de contrôle et les espaces de noms du noyau) et un système de fichiers compatible avec l'union (tel que OverlayFS) pour permettre aux conteneurs de s'exécuter dans un seul Linux exemple, en évitant les frais généraux de démarrage et de maintenance machines virtuelles. Docker sur macOS utilise une machine virtuelle Linux pour exécuter les conteneurs.

Les conteneurs Docker étant légers, un seul serveur ou une seule machine virtuelle peut exécuter plusieurs conteneurs simultanément. Une analyse de 2018 a révélé qu'un cas d'utilisation typique de Docker implique l'exécution de huit conteneurs par hôte et qu'un quart des organisations analysées en exécutent 18 ou plus par hôte.

La prise en charge des espaces de noms par le noyau Linux isole principalement la vue d'une application sur l'environnement d'exploitation, y compris les arborescences de processus, le réseau, les ID utilisateur et les systèmes de fichiers montés, tandis que les groupes de contrôle du noyau fournissent une limitation des ressources pour la mémoire et le processeur.

9.1.1- Docker Serveur applicatif

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["Projects/APIRobot/APIRobot.csproj", "Projects/APIRobot/"]
COPY ["Libraries/MongoDBAccess/MongoDBAccess.csproj", "Libraries/MongoDBAccess/"]
COPY ["Libraries/ConfigPolicy/ConfigPolicy.csproj", "Libraries/ConfigPolicy/"]
COPY ["Projects/SharedModels/SharedModels.csproj", "Projects/SharedModels/"]
RUN dotnet restore "Projects/APIRobot/APIRobot.csproj"
COPY .
WORKDIR "/src/Projects/APIRobot"
RUN dotnet build "APIRobot.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "APIRobot.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "APIRobot.dll"]
```

Nous avons réalisé un Dockerfile multi-étapes. Cela évite :

- Création manuelle d'images intermédiaires
- Réduire la complexité
- Copie des artefacts d'une étape à une autre
- Minimise la taille finale de l'image

Un Dockerfile à plusieurs étapes combine les instructions de développement et de production dans un seul Dockerfile.

L'étape 1 configure l'image qui sera utilisée pour la production (alias "base"). L'étape 2 utilise une image SDK (alias "build"), copie notre code de projet dans un répertoire de travail, restaure les packages NuGet, crée le code et le publie dans un répertoire nommé publish . L'étape 3 copie le répertoire de publication dans le répertoire de travail de l'image de production et définit la commande dotnet à exécuter une fois le conteneur en cours d'exécution.

Ce qui était au début deux images différentes sont maintenant combinées en une seule à l'aide du Dockerfile à plusieurs étapes. Le résultat final est une image de production qui peut être utilisée pour exécuter le conteneur sur votre machine, sur un serveur ou dans le cloud.

[9.1.2- Docker Angular](#)

```
FROM node:12.16.1-alpine

RUN npm install -g @angular/cli@11.2.12

WORKDIR /app

COPY ./package.json ./package-lock.json ../
RUN npm install

COPY ..

ENTRYPOINT ["npm", "run", "dockerSslProd"]
```

C'est un Dockerfile à une seule étape. Premièrement nous installons le CLI de angular afin de pouvoir utiliser les commandes angular. Nous créons ensuite un répertoire qui contiendra l'application. Nous copions le fichier packages.json contenant toutes les dépendances de l'application, puis nous les installons.

Enfin nous copions tous les fichiers angular de la solution. Nous définissons à la fin la commande à exécuter pour le démarrage de l'application angular lors de l'exécution de conteneur docker.

9.2- Docker Compose

Docker Compose est un outil permettant de définir et d'exécuter des applications Docker multi-conteneurs. Il utilise des fichiers YAML pour configurer les services de l'application et effectue le processus de création et de démarrage de tous les conteneurs avec une seule commande. L' docker-compose utilitaire CLI permet aux utilisateurs d'exécuter des commandes sur plusieurs conteneurs à la fois, par exemple, la création d'images, la mise à l'échelle de conteneurs, l'exécution de conteneurs arrêtés, etc. Les commandes liées à la manipulation d'images ou aux options interactives avec l'utilisateur ne sont pas pertinentes dans Docker Compose car elles s'adressent à un seul conteneur. Le docker-compose.yml est utilisé pour définir les services d'une application et comprend diverses options de configuration.

DockerCompose

Nous définissons dans le Docker compose les quatres conteneurs :

- Conteneur pour la base Mongo
- Pour le seeding de la base
- Pour le serveur applicatif
- Pour l'interface utilisateur

Pour chacun nous définissons l'image à utiliser ainsi que la location de leur Dockerfile.

Nous lions ensuite plusieurs conteneur entre eux pour qu'il puisse communiquer ensemble :

- le front avec le serveur
- le serveur avec la base
- le seeding avec la base

```
services:  
  mongoApp:  
    container_name: mongoApp  
    image: mongo:latest  
    restart: always  
    volumes:  
      - ./mongodb/db:/data/db  
    ports:  
      - "27017:27017"  
  
  mongoseeding:  
    image: ${DOCKER_REGISTRY}-mongoseeding  
    build:  
      context: ../../SolutionIOT  
      dockerfile: Projects/MongoSeeding/Dockerfile  
    links:  
      - mongoApp  
  
  apirobot:  
    image: ${DOCKER_REGISTRY}-apirobot  
    restart: always  
    build:  
      context: ../../SolutionIOT  
      dockerfile: Projects/APIRobot/Dockerfile  
    links:  
      - mongoApp  
  
  frontrobot:  
    container_name: FrontRobot  
    image: ${DOCKER_REGISTRY}-frontrobot  
    restart: always  
    build:  
      context: ../../Angular  
      dockerfile: Dockerfile  
    links:  
      - apirobot
```

DockerCompose override

Ce fichier permet d'override le fichier précédent en spécifiant d'autres configuration pour les conteneurs.

Pour le serveur applicatif nous spécifiant toutes les variables d'environnement dont à besoin le serveur ASPNET. C'est-à-dire les urls et ports pour les protocoles HTTP et HTTPS. L'emplacement des certificats pour HTTPS, mais aussi la redirection des ports de l'extérieur vers les ports internet du conteneur. Enfin nous mappons aussi grâce à l'instruction "volumes" des dossiers entre le système et le conteneur docker.

```
services:
  apirobot:
    environment:
      - ASPNETCORE_URLS=https://+:8001;http://+:8000
      - ASPNETCORE_HTTPS_PORT=5001
      - ASPNETCORE_Kestrel_Certificates_Default_Password=0099669
      - ASPNETCORE_Kestrel_Certificates_Default_Path=/https/certificat.pfx
    ports:
      - "8000:8000"
      - "8001:8001"
      - "1883:1883"
      - "11000:11000"
      - "8884:8884"
    volumes:
      - ../Certificates/self/https:/https/
  frontrobot:
    ports:
      - "443:443"
    volumes:
      - ../Certificates/self:/https/
```

DockerCompose debug et release

Pour pouvoir passer le serveur et ainsi charger une configuration différente comme nous l'avons expliqué plus tôt, nous devons spécifier via une variable d'environnement si le serveur est dans un environnement de production ou de développement lors de l'exécution du conteneur. Pour cela nous avons deux fichier à spécifier avec les commandes docker-compose :

```
services:
  apirobot:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
```

```
services:
  apirobot:
    environment:
      - ASPNETCORE_ENVIRONMENT=Production
```

9.3- Installation du VPS

Nos applications sont hébergées dans des conteneurs Docker et orchestrées par Docker-Compose. Les conteneurs peuvent être exécutés sur des machines virtuelles. De plus, nous avons besoin de Git pour pouvoir avoir accès au code source de nos applications.

A la première utilisation du VPS, il est alors nécessaire d'installer toutes les dépendances.

La première étape est de mettre à jour le gestionnaire de package Linux :

```
# Update and upgrade
sudo apt-get update && apt-get upgrade
```

Puis nous installons Git:

```
# Install Git
sudo apt install git-all
sudo git --version
```

Puis nous installons Docker:

```
# Install Docker

sudo apt-get remove docker docker-engine docker.io containerd runc

#apt-transport-https : permet au gestionnaire de packages de transférer des fichiers et des données via https
#certificats ca : permet au système (et au navigateur Web) de vérifier les certificats de sécurité
#curl : il s'agit d'un outil de transfert de données
#software-properties-common : ajoute des scripts pour gérer les logiciels
sudo apt-get install apt-transport-https ca-certificates curl gnupg lsb-release

#Pour vous assurer que le logiciel que vous installez est authentique
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

#Pour installer le référentiel Docker
#La commande « $(lsb_release -cs) » analyse et renvoie le nom de code de votre installation Ubuntu.
#De plus, le dernier mot de la commande - stable - est le type de version de Docker.
echo \
  "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

#Mettez à jour les référentiels après ajout
sudo apt-get update

#Pour installer la dernière version de docker
sudo apt-get install docker-ce docker-ce-cli containerd.io

sudo docker --version
```

Puis nous installons Docker-Compose :

```
# Install docker compose

sudo curl -L "https://github.com/docker/compose/releases/download/1.29.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
sudo docker-compose --version

mkdir github_repo
cd github_repo
```

Nous avons maintenant tous les outils nécessaires pour le déploiement de nos conteneurs d'applications.

9.4- Déploiement Docker-compose

Pour pouvoir déployer nos conteneur nous devons build les images. Pour cela nous utilisons la commande docker-compose suivi de tous les fichiers yml que nous voulons prendre en compte.

Une fois le build fini, nous avons simplement à effectuer docker-compose up pour créer les conteneurs et les exécuter.

Toutes les commandes sont disponibles dans le fichier script de notre projet.

Bilan

1- Résultats

Nous avons donc :

- Un robot composé d'un ESP pour contrôler les moteurs, récolte les données des capteurs, de l'ESP pour gérer la caméra embarqué, les deux pouvant interagir avec le Cloud.
- Un serveur applicatif récolte les données des robots, et les sauvegarde. Sert de proxy entre l'application de contrôle et le robot, et entre la caméra embarquée et le site web.
- Le serveur applicatif met en œuvre l'identification, l'authentification, les autorisations et le chiffrement.
- L'application de contrôle permet le contrôle des robots via une manette ou kinect.
- Nous avons une application de visualisation du robot pour les utilisateurs.

Notre robot envoie alors les images de la caméra ainsi que ses données au serveur, qui les sauvegarde et les redirige vers l'application de visualisation. Il est contrôlable grâce à notre application de contrôle via le Cloud et localisable.

2- Difficultés rencontrées

- Pas accès au matériel tous en même temps à cause de la distance (télétravail etc..)
- Difficile de trouver des créneaux pour se réunir et bosser en physique sur le robot
- Matériel manquant

3- Evolutions

D'autres idées peuvent être développées par la suite telles que :

Projeter les données du robot sur un mur comme le font les réveils, cela permettra aussi d'afficher l'heure où d'autres informations.

Le robot pourrait suivre une personne grâce à la reconnaissance d'image ou via des capteurs (mettre en place un apprentissage grâce à l'IA).

Utilisation de la reconnaissance vocale pour mettre de la musique par exemple (si on peut avoir un module fait pour supporter ce genre de fonctionnalités comme l'assistant google).

Le robot serait aussi capable de suivre un parcours matérialisé par une ligne.

Le robot serait capable grâce au micro inclus dans le téléphone, de répéter des phrases que l'on lui a dite à voix haute ou que l'on a écrit dans l'application

Meilleure sécurité de l'ESP et pour les passwords (man in the middle), cela à était commencé et tout est implémenté sur le serveur (HTTPS).

Dash board de gestion utilisateur et des droits pour les ressources serveurs, améliorer le système de droits et d'authentification, par exemple générer plutôt des clés d'API au lieu de toujours générer des tokens d'authentification.

Conclusion

TODO

Bibliographie

<https://randomnerdtutorials.com/esp32-dc-motor-l298n-motor-driver-control-speed-direction/>

Annexes