INSTITUT UNIVERSITAIRE DE TECHNOLOGIE
UNIVERSITÉ D'ANGERS

PROJET : Cuve connecté

BUT GEII Semestre 3

Rapport

IUT ANGERS 2024/2025

AL NEJEM Adrien
HASSANI Houbayida

SOMMAIRE

Introduction

Contexte:

Dans la ville d'Angers, les précipitations sont fréquentes. En 2024, la ville a enregistré une augmentation de 15 % par rapport à la normale, selon Météo-France.
Dans une démarche écologique, des particuliers utilisent des cuves afin de récupérer ces eaux de pluie.
Ces eaux sont principalement utilisées pour des activités telles que l'arrosage des jardins, le lavage de véhicules ou d'autres usages domestiques.
La plupart de ces cuves de récupération sont enterrées sous terre pour des raisons esthétiques et pratiques.
Cependant, cette configuration rend difficile la surveillance de ces cuves.
C'est pour répondre à ce besoin que nous avons choisi ce projet.
Notre but est d'apporter une solution technique permettant la surveillance du niveau d'eau, de l'humidité et de la température de ces cuves à distance.

Cahier des charges:

Pour réaliser ce projet, nous avions un délai de 4 semaines avec de nombreuses tâches à effectuer qui sont listées ci-dessous :

- Mesure de la température et de l'humidité de l'eau
- Mesure du niveau d'eau et de la tension
- Affichage des mesures sur un site web
- Communication par Wi-Fi
- Utilisation du Firebettle ESP32
- Étude et assemblage de la carte électronique
- Création de librairies
- Autonomie du système sur une année

Schéma fonctionnel:

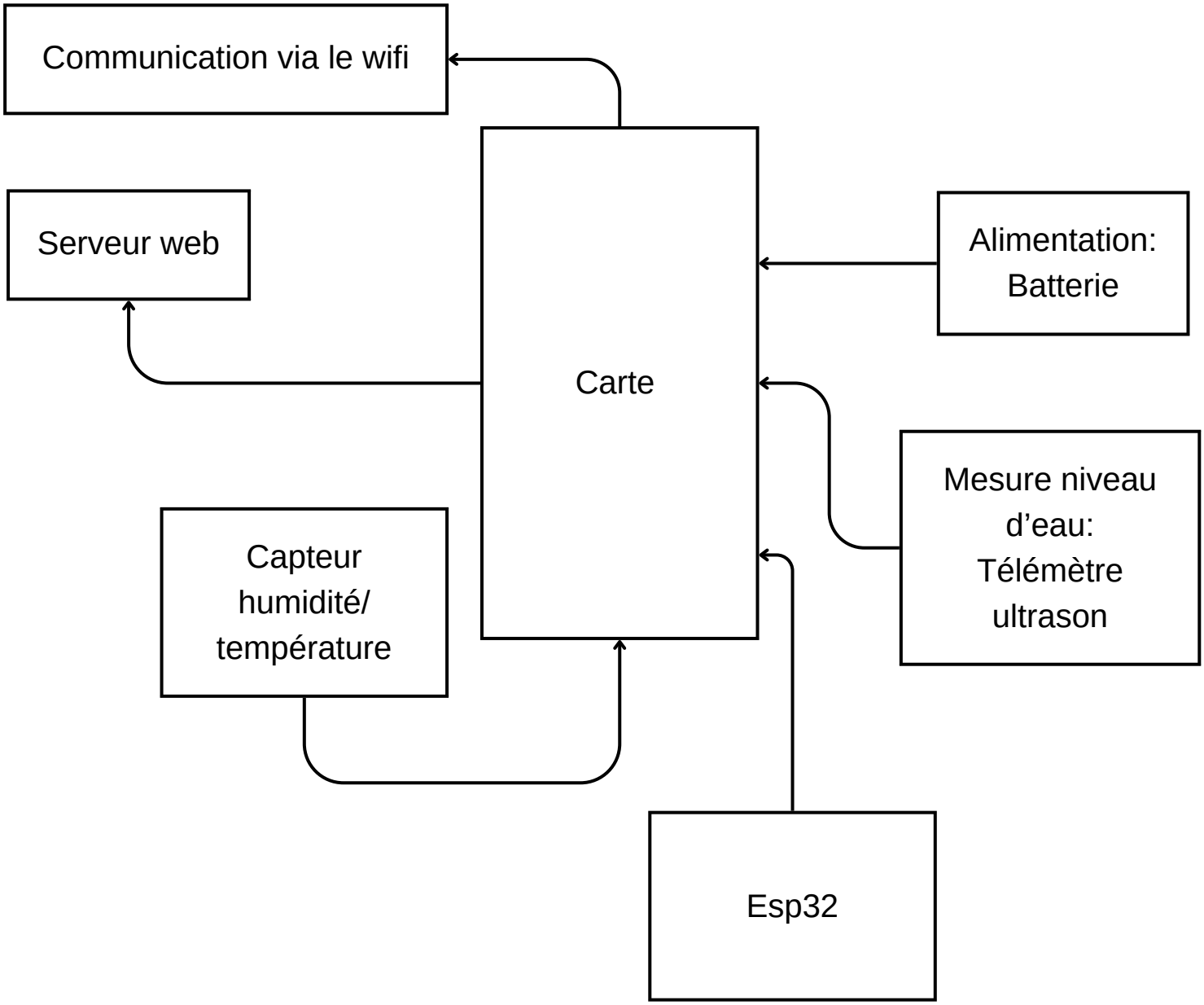
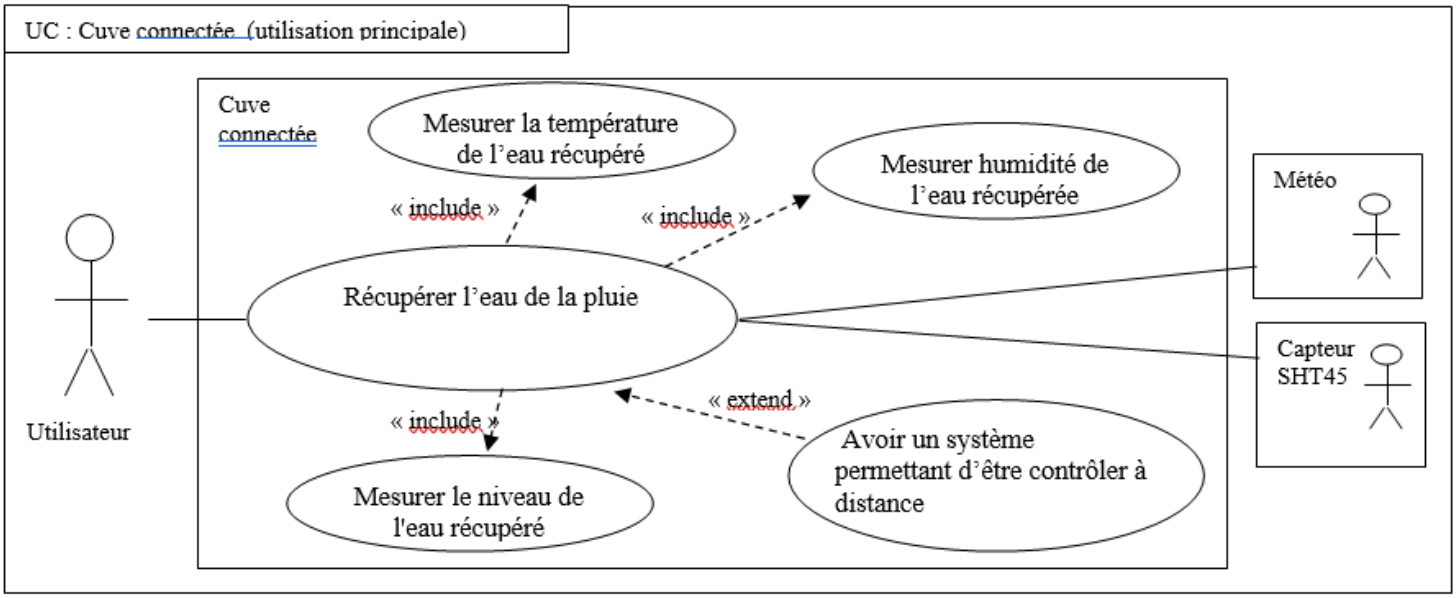


Diagramme des cas d’utilisation

Ce diagramme décrit les services rendus par le système vis-à-vis d'un acteur particulier.
Il montre les interactions du système avec les acteurs extérieurs : rôles tenus par des humains ou autres systèmes.



Gestion de projet

Ce tableau présente approximativement le temps qui a été pris pour réaliser les tâches énumérées ci-dessus.

Étude des composants et mesure des différentes valeurs	**2 semaines**
Interface web	**1 semaine**
Création de librairies	**2 heures**
Assemblage et tests de la carte	**3 heures**

Choix technologiques

Rôle de la carte

La carte a pour rôle principal de collecter les données des capteurs (humidité, distance, température, et pourcentage de la batterie) via l'ESP32. Ce dernier centralise et traite ces informations tout en les transmettant grâce à ses fonctionnalités IoT. La carte assure également l'alimentation des différents composants. Enfin, le bouton poussoir permet, grâce à un signal NL0, de sortir le système du mode veille.

Nomenclature

Type	Name	Value	Library	Device	Gate	Footprint	Sheet	Module
Part	FRAME1		0IUT CARTOUCHES	--	G$1	--	1/1	
Part	U2	SHT41-AD1B	0IUT xPROJETS	--	G$1	SHT41-AD1B	1/1	
Part	R1	10k	0IUT RLCQ	-CMS1206	G$1	RCMS1206	1/1	
Part	R2	10k	0IUT RLCQ	-CMS1206	G$1	RCMS1206	1/1	
Part	C1	100nF	0IUT RLCQ	-CMS1206	G$1	CCMS1206	1/1	
Part	GND2	GND	0IUT ALIMENTATIONS	--	VR1	--	1/1	
Part	J1		0IUT CONNECTEURS	-EMBXAD	G$1	XA_4D	1/1	
Part	GND4	GND	0IUT ALIMENTATIONS	--	VR1	--	1/1	
Part	C2	100nF	0IUT RLCQ	-CMS1206	G$1	CCMS1206	1/1	
Part	C3	4.7µF	0IUT RLCQ	-SMC_A	G$1	SMC_A	1/1	
Part	GND5	GND	0IUT ALIMENTATIONS	--	VR1	--	1/1	
Part	C4	10µF	0IUT RLCQ	-SMC_C	G$1	SMC_C	1/1	
Part	C5	100nF	0IUT RLCQ	-CMS1206	G$1	CCMS1206	1/1	
Part	H1	HOLE3.2	0IUT HOLES	3.2	G$1	3,2	1/1	
Part	H2	HOLE3.2	0IUT HOLES	3.2	G$1	3,2	1/1	
Part	H3	HOLE3.2	0IUT HOLES	3.2	G$1	3,2	1/1	
Part	H4	HOLE3.2	0IUT HOLES	3.2	G$1	3,2	1/1	
Part	BP1		0IUT xPROJETS	-TC	G$1	11TC05L	1/1	
Part	R3	10k	0IUT RLCQ	-CMS1206	G$1	RCMS1206	1/1	
Part	C6	100nF	0IUT RLCQ	-CMS1206	G$1	CCMS1206	1/1	
Part	GND6	GND	0IUT ALIMENTATIONS	--	VR1	--	1/1	
Part	ESP32-FB-0654	ESP32-EV1.0	0IUT xPROJETS	--	G$1	ESP32-EV1.0	1/1	
Part	GND3	GND	0IUT ALIMENTATIONS	--	VR1	--	1/1	

Composant	Rôle
ESP32 (FireBeetle 2)	Microcontrôleur principal : collecte, centralise et traite les données, connecte au Wi-Fi.
SHT45	Capteur de température et d'humidité : mesure les conditions environnementales.
A02YYUW	Capteur à ultrasons : mesure la distance.
BP1 (Bouton poussoir)	Permet de sortir la carte du mode veille via un signal NL0.
C1, C2, C5, C6	Condensateurs de découplage : stabilisent les tensions en fournissant les appels de courant.
C3, C4	Condensateurs de lissage : stabilisent la tension en éliminant les fluctuations.
R1, R2, R3	Résistances : limitent le courant et ajustent les niveaux logiques.
J1 (Connecteur A02YYUW)	Connecteur pour le capteur à ultrasons : facilite le branchement et l'alimentation.
J2 (Connecteur SHT45)	Connecteur pour le capteur de température et d'humidité : assure la connexion avec l'ESP32.

Schéma structurel

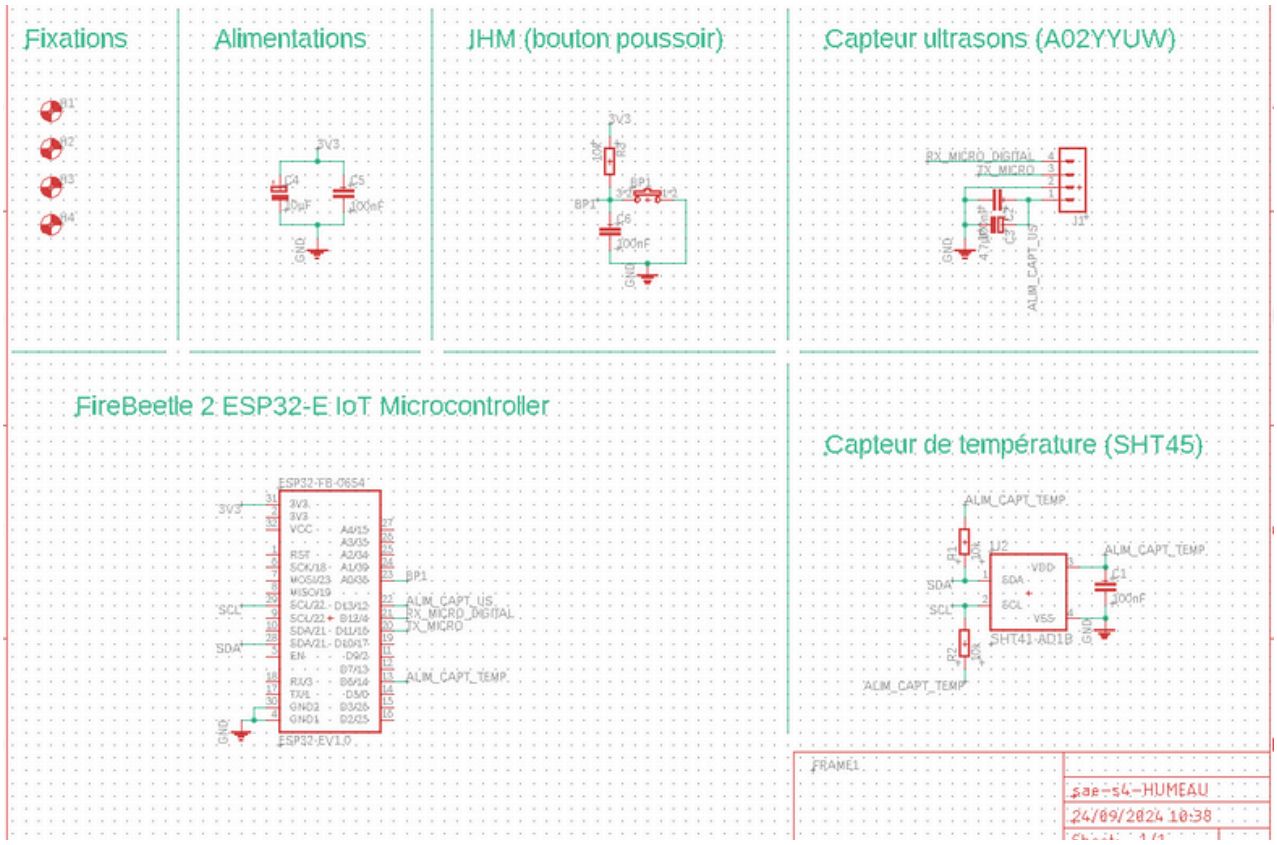
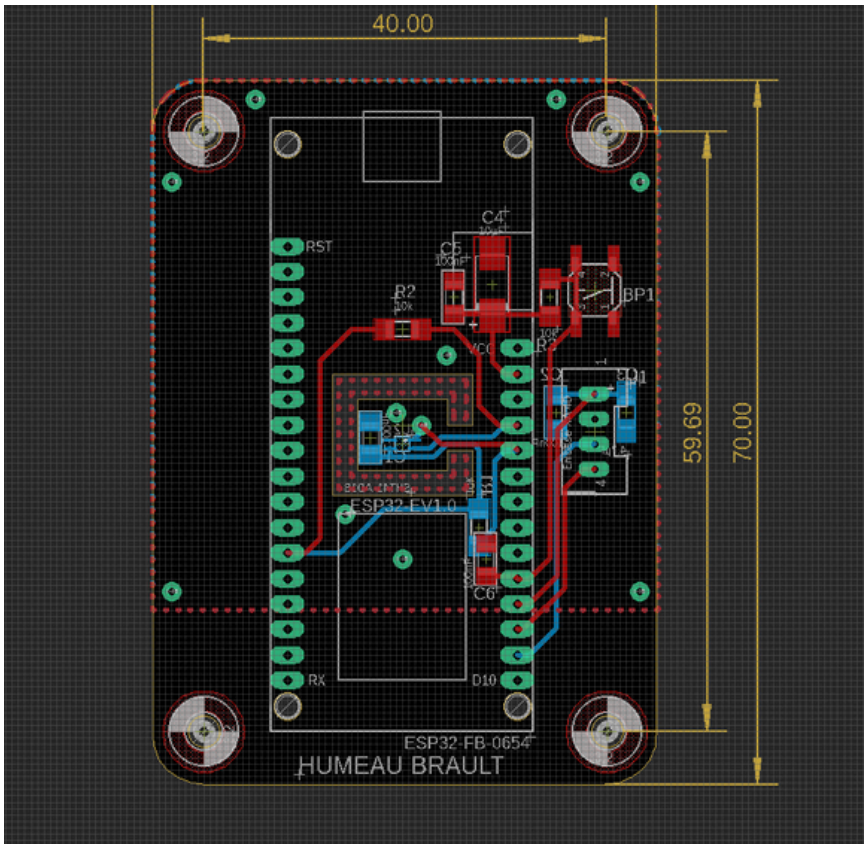
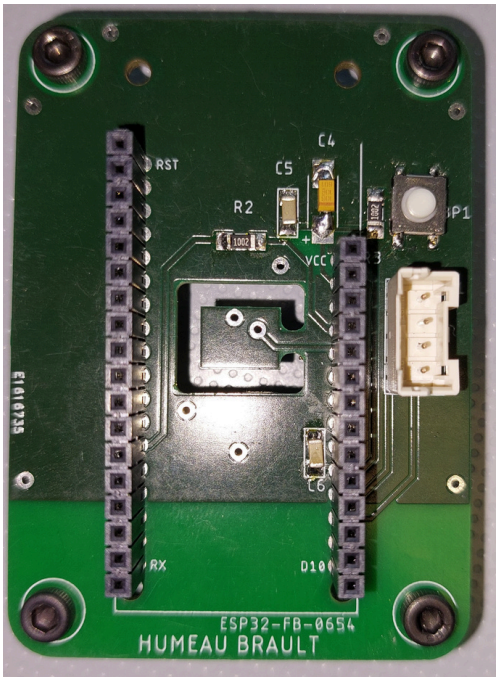
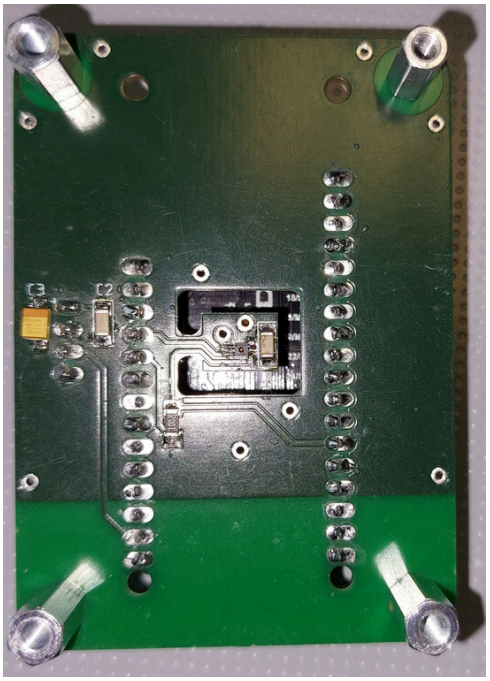


Schéma de routage



Carte assemblée

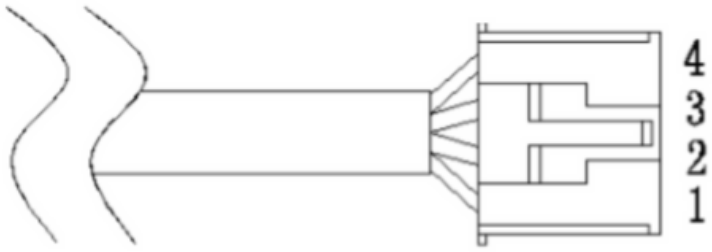




a) Capteur Ultrason

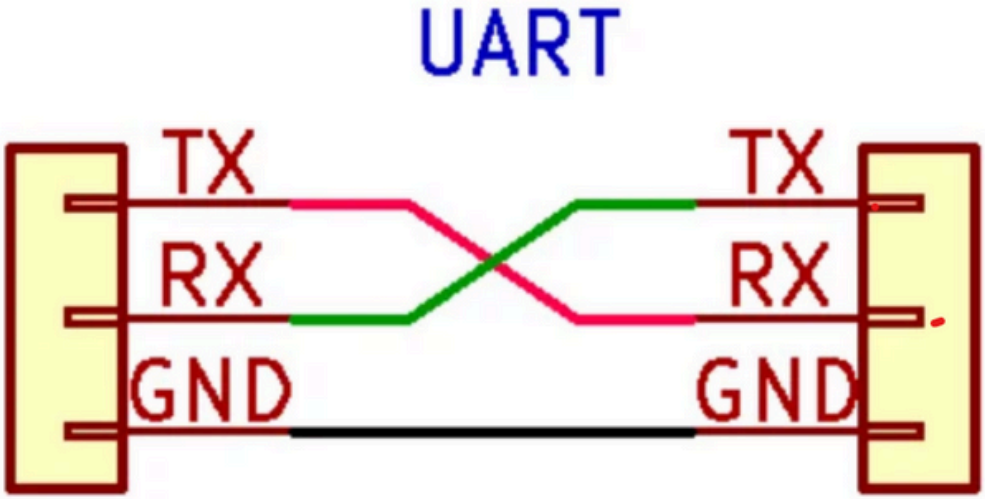
Le capteur ultrason SEN0311 de DFRobot est un capteur de mesure de distance étanche, idéal pour des applications dans des environnements difficiles ou extérieurs. Il utilise des ondes ultrasonores pour détecter des objets et mesurer des distances avec précision.

Nous avons opté pour le SEN0311 qui présente une consommation réduite (<8mA), peut être alimenté en 3,3V ou 5V, est étanche et peut mesurer des distances allant de 0,03 à 4,5m qui recouvre donc la plage de profondeur de la cuve. Ce capteur peut dialoguer avec le microcontrôleur par liaison série UART (RX/TX) et possède de librairies qui permettent de le piloter facilement et de récupérer les grandeurs mesurées avec le microcontrôleur.



Étiquette	Nom	Description
1	VCC	Entrée d'alimentation
2	GND	Sol
3	RX	Sélection de sortie de valeur traitée/valeur en temps réel
4	Émission	Sortie UART

Fonctionnement de la liaison UART:



- Le capteur ultrason SEN0311 utilise la broche TX pour envoyer automatiquement des mesures sous forme de trames UART.

- Le microcontrôleur lit ces trames via sa broche RX, extrait les octets de données, et calcule la distance en millimètres.

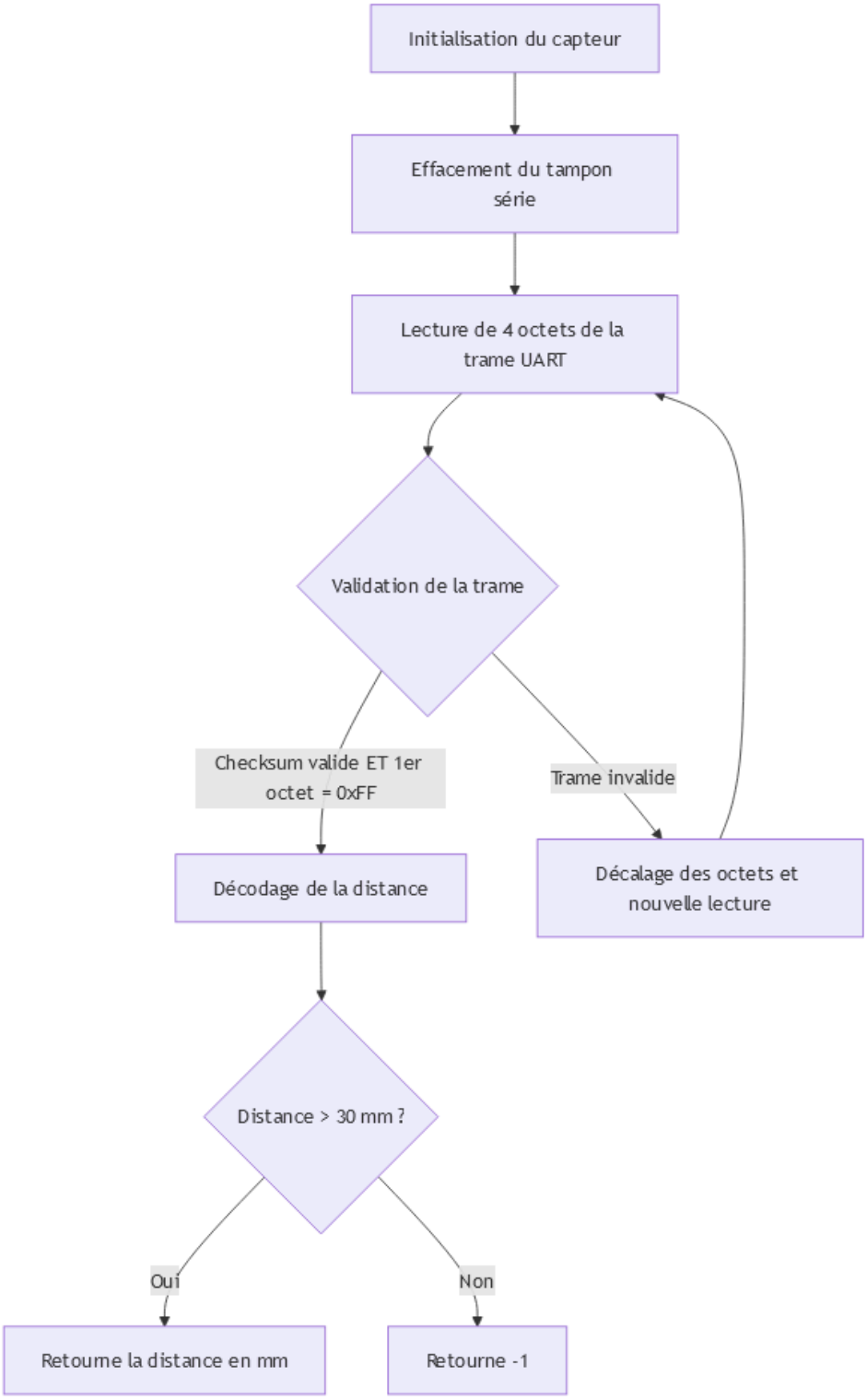
- Ce protocole permet une communication simple et fiable pour surveiller les distances avec précision.

- La vitesse de transmission est de 9600 bits par seconde (9600 bps).

- Configuration :
  - Bit de start :1
  - Bits de données : 32 bits
  - Bit de parité : Aucun
  - Bit de stop: 1

Code du capteur à ultrasons :

Ce schéma résume le fonctionnement du code permettant à l'ESP32 de mesurer des distances en utilisant le capteur ultrasonique, grâce à une communication série via les broches TX/RX. Le capteur envoie des trames de données, et le programme traite ces trames pour extraire une distance fiable.



Voici les points clés du fonctionnement :

- Initialisation du capteur :

On initialise la liaison série ainsi que les broches TX et RX utilisées pour la communication avec le capteur. Le port série est initialisé à 9600 bauds. On configure la broche 12 comme sortie pour alimenter le capteur ultrasonique

- Nettoyage du buffer (flushUs) :

Avant chaque nouvelle mesure, la méthode flushUs() efface le buffer série pour éviter d'éventuelles données parasites. Cela assure que seules les nouvelles trames du capteur seront traitées.

- Lecture de la distance (getDistance) :

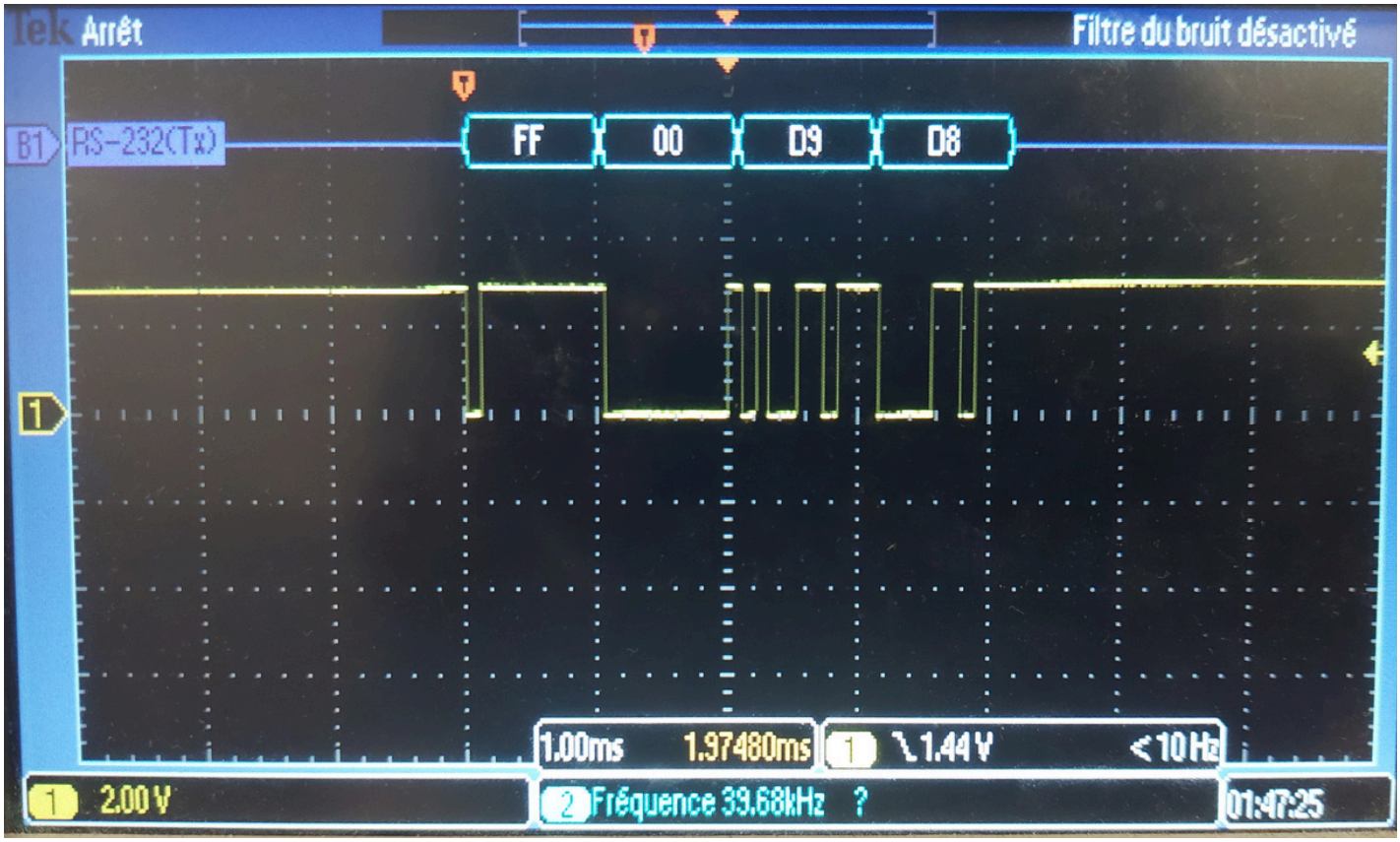
La méthode principale, getDistance(), gère la réception des données :

  - Un tableau data[4] est utilisé pour stocker les octets reçus, qui forment une trame. Cette trame contient la distance sur deux octets, un octet de synchronisation (toujours 0xFF), et un octet pour le checksum.
  - Checksum : Le programme calcule un checksum en additionnant les trois premiers octets de la trame et en prenant les 8 bits de poids faible (& 0x00FF). Si cette somme correspond au quatrième octet de la trame, la donnée est considérée comme correcte.
  - Si le checksum est incorrect ou si le premier octet n'est pas 0xFF, la trame est rejetée, et le programme tente de corriger en décalant les données reçues pour chercher une trame correcte.

- Extraction de la distance :

Si la trame est validée, la distance est extraite en combinant les deux octets centraux de la trame. La distance est calculée ainsi : (data[1] << 8) + data[2]. Ce décalage permet de reconstituer la valeur sur 16 bits. Si la distance mesurée est supérieure à 30 mm, elle est renvoyée, sinon la fonction retourne -1.

Visualisation à l’oscilloscope :



Trame de données envoyée par le capteur :

- Le capteur envoie périodiquement une trame de 4 octets contenant les informations de distance mesurée. 0xFF 0x00 0xD9 0xD8

- 0xFF : Octet de synchronisation (indique le début de la trame).

- 0x00(MSB) et 0xD9(LSB) : Octets représentant la distance mesurée en millimètres (valeur 16 bits).

- La distance est calculée comme suit :
  Distance (en mm) = 0x00 × 256 + 0xD9 = 0 + 217 = 217 mm.

- 0xD8 : Octet de somme de contrôle (checksum) pour vérifier l'intégrité des données. Le calcul du checksum est correct : 0xFF+0x00+0xD9=0x1D8. La somme des trois premiers octets donne 0xD8 après un modulo 256.

b) Batterie :

Pour pouvoir alimenter le télémètre, nous avons opté pour une batterie LiPo (polymère au lithium) 6000 mAh. D'après le site du constructeur, ce type de batterie fournit une alimentation électrique très efficace et il est protégé avec un circuit de contrôle.
Son coût est de 19,80 euros ce qui est plutôt raisonnable sachant que, d'après l'étude ci-dessous, cette batterie à une autonomie d'environ 460 jours :

En effet, l'ESP32 ayant une consommation de 40mA en mode actif et de 10µA en mode deep-sleep d'après le site de Lucidarme, le capteur ultrason ayant une consommation de 8mA et le capteur de température ayant une consommation de 0,4µA, on peut en déduire le courant consommé par heure (pour une Tactif=20s et Tsleep=30min :

Si T = 30 min, on a I = Actif x (0,04 + 0,008 + 4 x 10^^-6) + Tsleep * 10µ = 20sec/3600 x (0,04 + 0,008 + 4 x 10^^-6) + 0,5 x 10µ = 271,7µA

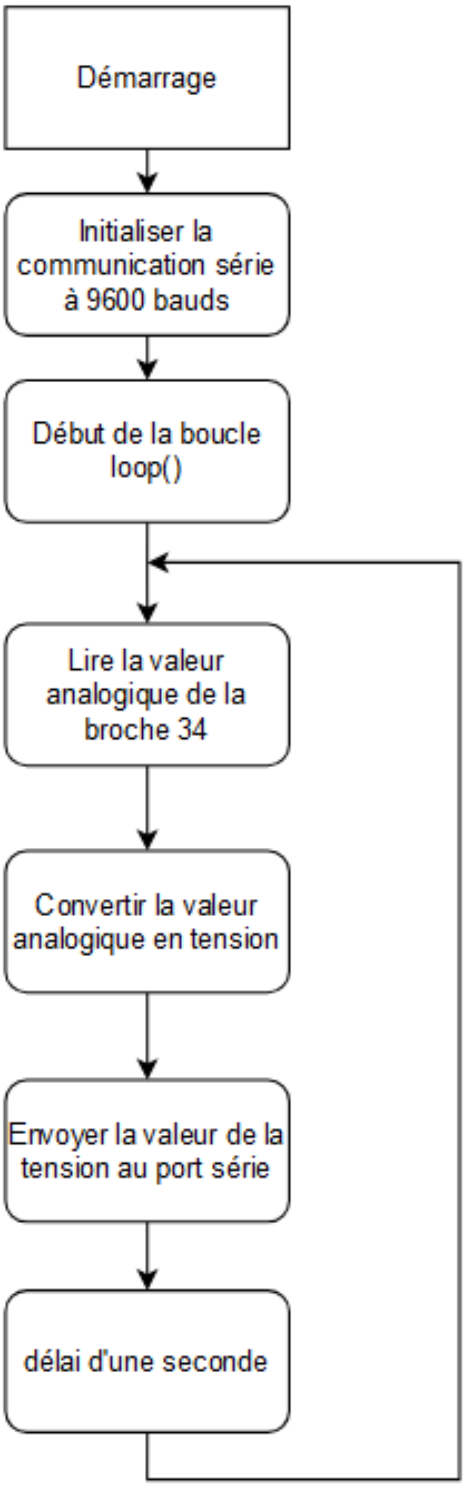
On a donc l'autonomie suivante :

Autonomie = 6Ah/543,3378µA = 11042,9 heures = 1,26 ans = 460 jours

Dimensions : 50 mm x 50 mm x 20 mm

Code de la batterie

Cet organigramme correspond au programme pour mesurer et afficher la tension de la batterie.



Voici une explication de chaque ligne de code du programme.

Dans la partie setup, on a juste inséré une ligne de code pour envoyer la variable "tension" sur le port série avec une vitesse de transmission de 9600 bauds.

```
1   void setup() {
2   Serial.begin(9600);
3   }
```

Cette ligne de code permet de récupérer la valeur analogique tension sur la broche 34.

```
6   int valeurAnalog = analogRead(34);
```

Ce calcul permet de convertir la valeur analogique en décimal pour avoir la valeur de la tension.

```
7   float tension = (float) 2*valeurAnalog/1000;
```

Ici, on envoie sur le port série le texte "Tension sur 34 : " et la valeur de la tension.

```
8   Serial.print("Tension sur 34 : ");
9   Serial.println(tension);
```

À la fin, on ajoute un délai d'une seconde afin d'éviter que l'Arduino spamME le port série.

```
11   delay (1000);
```

c) Capteur de températures et d'humidités

Concernant le capteur de température et d'humidité, nous avons fait un tableau pour comparer les différents capteurs que l'on aurait pu utiliser :

Caractéristique	Capteur SHT4X	Capteur SHT3X	Bosch BME280	Honeywell HIH6130
Température de fonctionnement	-40 à 125 °C	-40 à 125 °C	-40 à 85 °C	-25 à 85 °C
Plage d'humidité	0 à 100% HR	0 à 100% HR	0 à 100% HR	0 à 100% HR
Précision température	±0.2 °C (typique)	±0.3 °C	±1.0 °C	±0.5 °C
Précision humidité	±1.5% HR	±2% HR	±3% HR	±4% HR
Temps de réponse	8 secondes typiques	8 secondes typiques	1 seconde	5 secondes
Consommation d'énergie	0.9 µA au repos	1.3 µA au repos	0.1 µA en veille	0.6 mA actif
Interface	I2C	I2C	I2C, SPI	I2C

Les capteurs SHT4X et SHT3X sont souvent utilisés car ils sont très précis pour faire des mesures de température et d'humidité. Également, ils ont une petite taille ce qui permet de les intégrer facilement sur une petite carte électronique comme la nôtre. En revanche, la soudure est donc beaucoup plus dure à réaliser.

Le Bosch BME280 mesure la pression barométrique en plus de la température et de l'humidité ce qui le rend idéal pour les applications météorologiques. Également, son temps de réponse est très rapide par rapport aux autres capteurs. Pour notre cuve connectée, nous ne nous intéressons pas à la mesure de la pression barométrique et on n'impose pas spécialement un temps de réponse très rapide.

Le Honeywell HIH6130 est connu pour sa robustesse et sa fiabilité dans des applications industrielles. En revanche, sa consommation d'énergie est très importante par rapport aux autres capteurs et il est beaucoup moins précis.

Notre choix s'est donc porté sur le capteur SHT4X qui est une version plus développée du SHT3X avec des améliorations en termes de précision et de consommation énergétique. Nous avons choisi le modèle SHT45 car c'est la version la plus récente et nous avons priorisé la qualité mais nous aurions très bien pu prendre le SHT41 si on avait des contraintes budgétaires.

Code du capteur de températures & d'humidités

Ci-dessous est présenté le programme permettant de mesurer la température et l'humidité.

- Nous utilisons 2 librairies pour ce programme.

1) Wire :
Cette bibliothèque est disponible directement sur Arduino et il permet de gérer la communication en I2C qui est l'interface de notre capteur SHT45.

2) SHTSensor :
Cette bibliothèque permet de simplifier l'interfaçage avec le SHT45. Elle est disponible sur le site "GitHub" mais elle comprend des notions en C++ comme les classes ce qui n'a pas simplifié la compréhension de celui-ci.
Il y a beaucoup de lignes dans ce programme qui ne nous intéressent pas forcément comme la classe SHT3xAnalogSensor qui gère les capteurs analogiques SHT3x (inutile car nous travaillons avec un SHT45).
En revanche, nous allons nous intéresser aux fonctions readTemperature() et readHumidity() qui permettent de lire directement la valeur de la température et de l'humidité à partir du capteur ce qui rend le code beaucoup plus simple.

```
1   #include <Wire.h>
2
3   #include "SHTSensor.h"
```

- Pour communiquer avec le capteur SHT45, nous utilisions la ligne de code ci-dessous.

En effet, cette ligne crée une instance de la classe SHTSensor nommée sht (définie dans la bibliothèque SHTSensor.h).
C'est cette instance qui nous permet d'interagir avec le capteur.

```
5   SHTSensor sht;
```

- Nous allons maintenant nous intéresser à la partie setup qui est exécutée lorsque l'on lance le programme pour initialiser les configurations de base.

Concernant la configuration des ports, il faut savoir que le SHT45 est initialement relié à la masse et alimenté sous vcc (3.3V) ce qui pose problème car si on est en veille, le SHT45 continuera de consommer. Nous avons donc fait le choix de mettre la broche vcc sur une broche en entrée de l'ESP32 (ici, on a choisi la broche 14).

```
11    pinMode(14, OUTPUT);
12    digitalWrite(14, HIGH);
```

Les lignes ci-dessous permettent d'initialiser la bibliothèque Wire et la communication série avec 115200 bauds pour la vitesse de transmission. On peut ainsi voir et envoyer des données sur le moniteur série.
On ajoute un délai de 1 seconde pour laisser la console série s'installer.

```
15    Wire.begin();
16    Serial.begin(115200);
17    delay(1000); // let serial console settle
```

L'instruction if...else utilisée ci-dessous permet d'initialiser notre capteur SHT45 et d'afficher sur le moniteur série si l'initialisation s'est réalisée avec succès ou non.
La dernière ligne de la fonction void setup() configure le capteur SHT45 avec une précision moyenne.

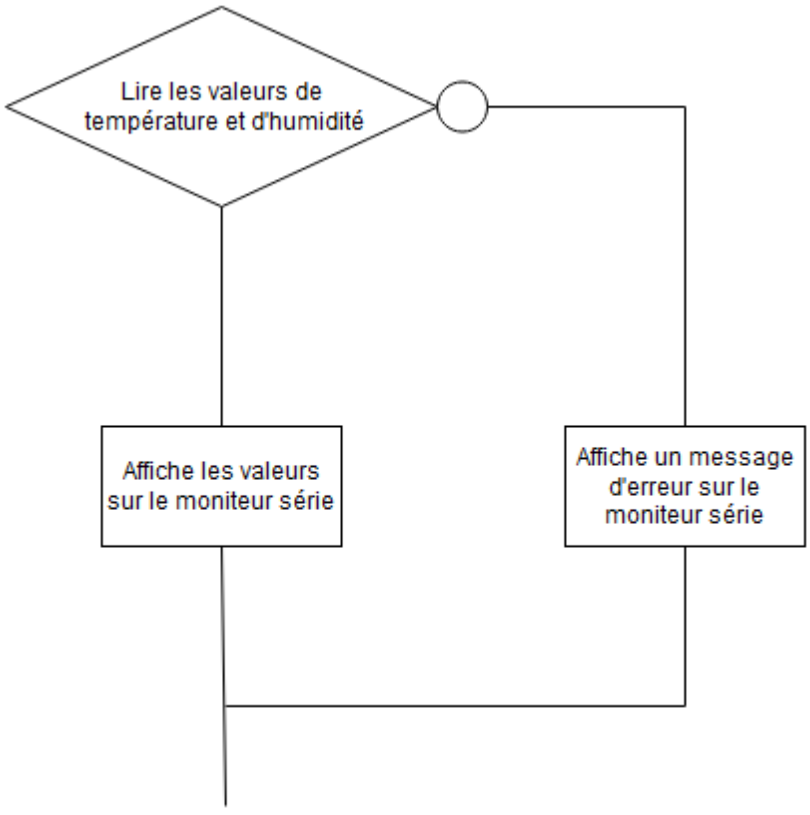
```
19    if (sht.init()) {
20        Serial.print("init(): success\n");
21    } else {
22        Serial.print("init(): failed\n");
23    }
24    sht.setAccuracy(SHTSensor::SHT_ACCURACY_MEDIUM);
```

- Nous allons maintenant nous intéresser à la partie loop qui est exécutée en boucle lorsque setup() est exécuté.

Les lignes de code ci-dessous nous permettent d'afficher la valeur de la température et de l'humidité avec 2 décimales sachant qu'il y a un message d'erreur si la lecture a échoué.
Avec le délai de 1 seconde, les valeurs seront mises à jour toutes les secondes sur le moniteur série.

```
31  if (sht.readSample()) {
32      Serial.print("SHT:\n");
33      Serial.print("  RH: ");
34      Serial.print(sht.getHumidity(), 2);
35      Serial.print("\n");
36      Serial.print("  T:  ");
37      Serial.print(sht.getTemperature(), 2);
38      Serial.print("\n");
39  } else {
40      Serial.print("Error in readSample()\n");
41  }
42
43  delay(1000);
```

Voici un organigramme représentant cette instruction :



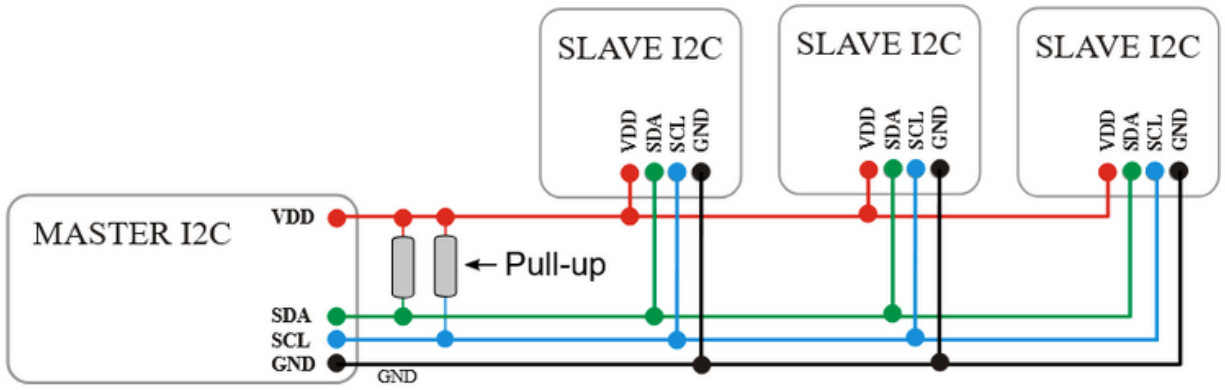
I2C

Comme vu précédemment, le SHT45 utilise l'interface I2C que nous allons présenter.
En effet, l'I2C est un bus de communication permettant à plusieurs périphériques électroniques de communiquer entre eux.
Cette communication se fait généralement entre un maître (ici, notre ESP32) et un ou plusieurs esclaves (par exemple, le SHT45).
Pour se faire, 3 fils sont nécessaire :

- un signal de donnée (SDA)
- un signal d'horloge (SCL)
- un signal de référence électrique (masse)

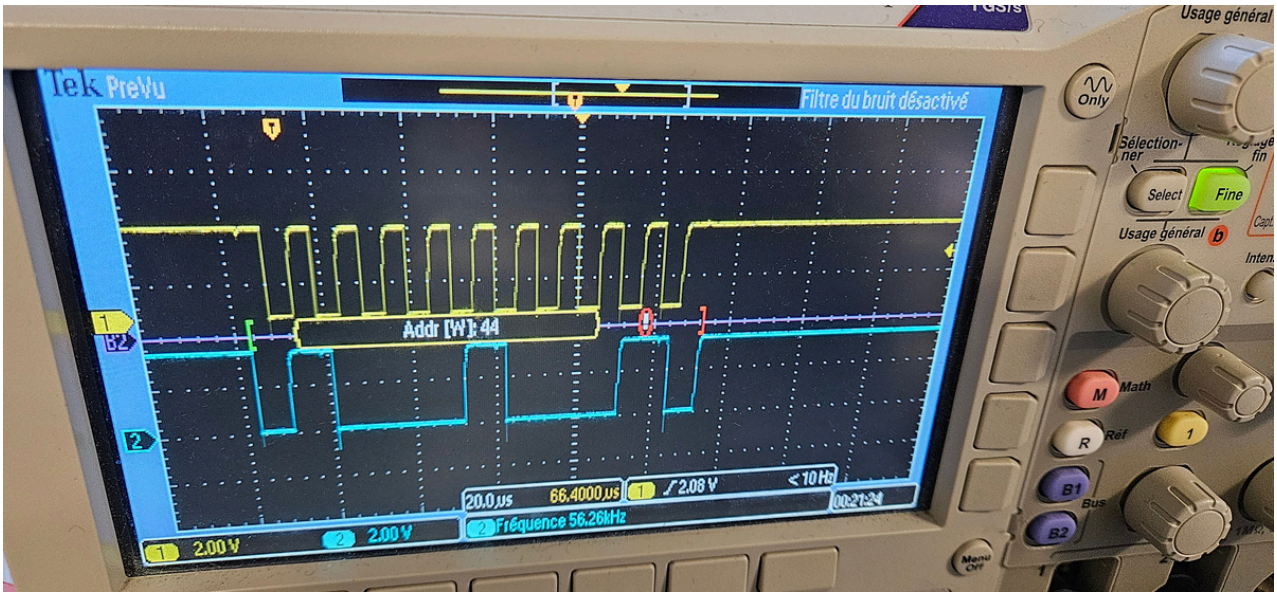
À noter que l'on ajoute une alimentation VDD et une résistance de pull-up sur chaque signal pour imposer le niveau de repos (NL1).

Voici la structure d'un module I2C :



Durant notre projet, on a rencontré un problème après l'assemblage de la carte.
En effet, lors des tests de la carte, on n'arrivait pas à faire fonctionner notre programme. On a donc visualisé à l'oscilloscope, avec l'aide de M. Leduc, les signaux SCL et SDA pour vérifier que l'on retrouvait bien 0x44 concernant les bits d'adresses comme mentionné dans la datasheet du SHT45.

Signaux SDA et SCL obtenus à l'oscilloscope :



Légende :

- Signal SCL
- Signal SDA

L'oscilloscope nous affiche bien 44 pour les bits d'adresse et on remarque également sur le signal d'horloge que le start passe de 1 à 0 et le stop passe de 0 à 1.

Extrait de la datasheet du SHT45

Device Overview

Product	Details
SHT40-xD1B	base RH&T accur., possible I2C addr.: 0x44, 0x45, 0x46
SHT40-AD1F	SHT40-AD1B with PTFE membrane
SHT40-AD1P	SHT40-AD1B with protective cover
SHT41-AD1B	intermed. RH&T accur., 0x44 I2C addr.
SHT43-ADCB	ISO17025 3-point calibration certificate
SHT45-AD1B	±1.0 %RH, ±0.1 °C accur., 0x44 I2C addr.

→ On retrouve bien 0x44 concernant les bits d'adresse.

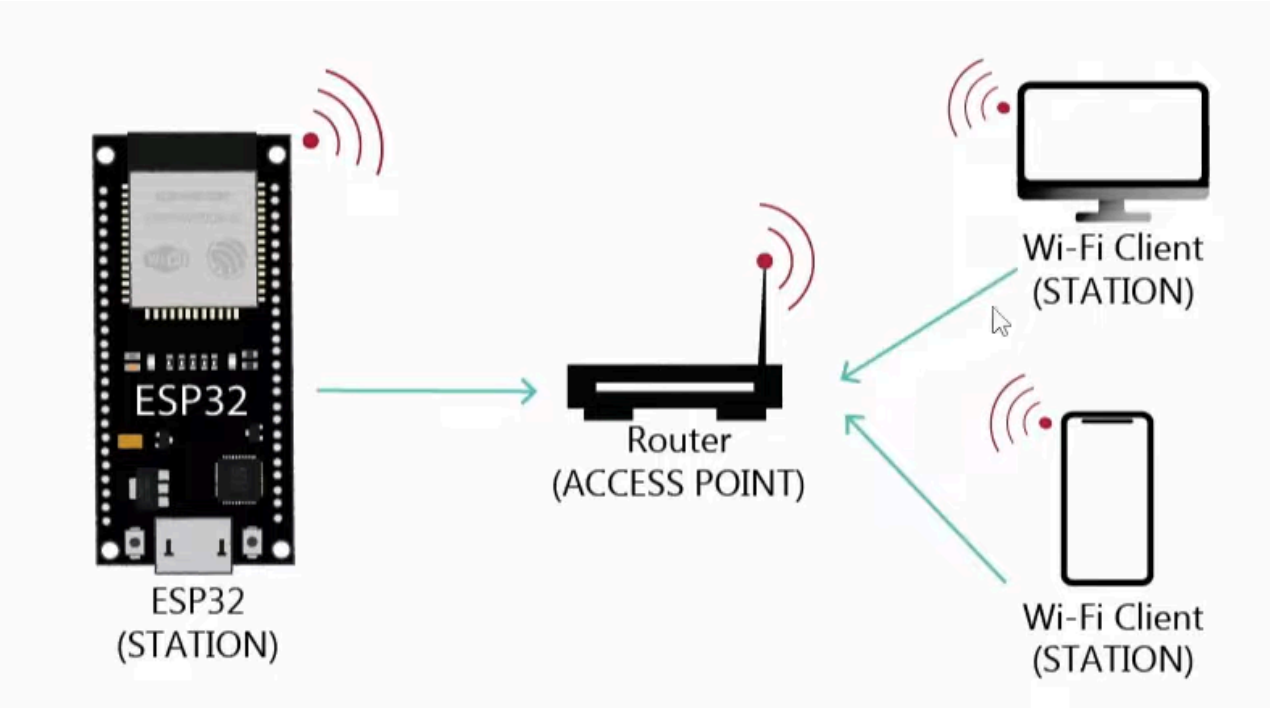
d) Esp32

présentation

Wifi

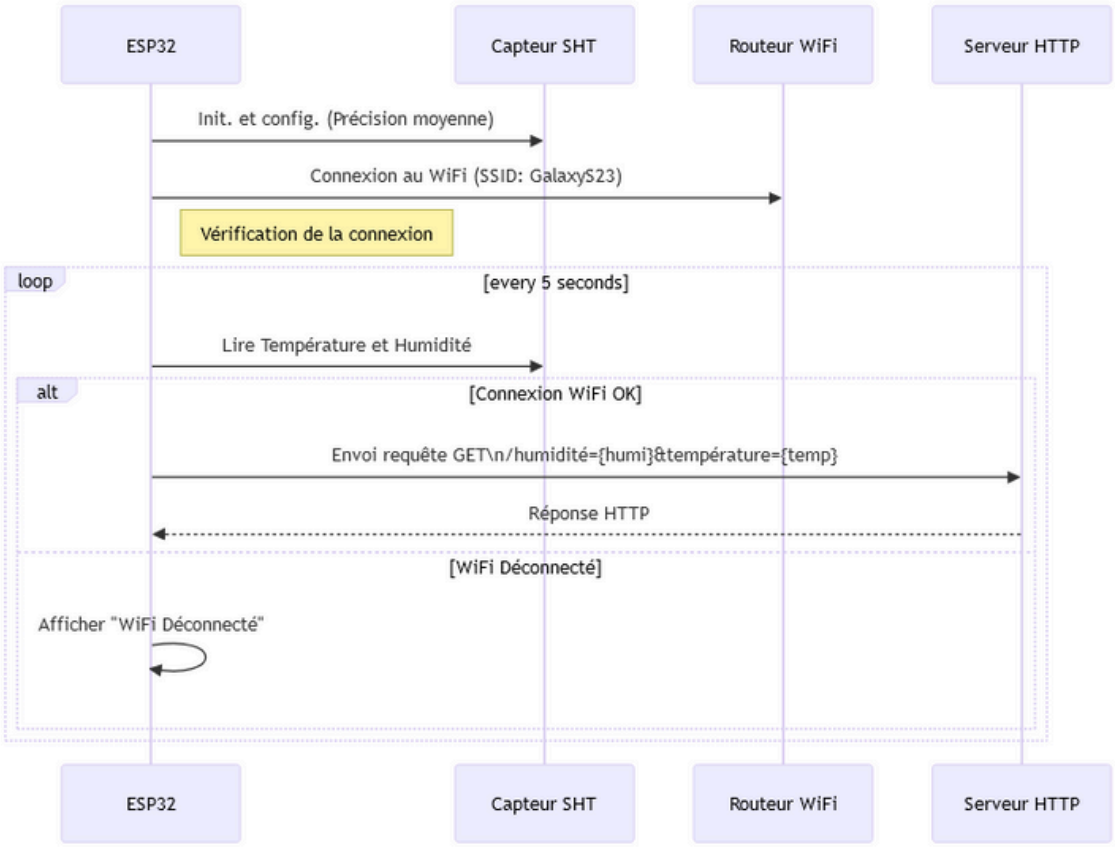
L'ESP32 possède 2 modes Wi-Fi mais nous allons uniquement nous intéresser au mode STATION.
Le principe est le suivant : L'ESP se connecte en Wi-Fi au routeur puis il peut envoyer et recevoir des requêtes d'autres appareils.
Pour le connecter à un réseau Wi-Fi, on a fait un partage de connexion avec notre téléphone. On a ensuite connecté l'ESP32 sur un PC qui est connecté au réseau Wi-Fi de notre partage de connexion.

Voici un schéma illustrant la connexion Wi-Fi avec l'ESP32 :



CODE PARTIE WI-FI

Cet organigramme correspond au programme pour afficher la température et l'humidité de l'eau dans la cuve sur une page web :



Le détail du programme est expliqué dans les pages suivantes.

CODE PARTIE WI-FI

Nous utilisons 4 librairies dans ce programme : Wifi.h, HTTPClient.h, SHTSensor.h et Wire.h. Dans cette partie, nous allons uniquement détailler les deux premières librairies énumérées car les deux autres ont déjà été détaillées précédemment.

Lorsqu'on installe la carte ESP dans le logiciel de développement Arduino, le programme gérant la carte intègre d'office la bibliothèque Wifi.
Il n'est donc pas nécessaire de l'installer. Il suffit juste d'appeler la bibliothèque en début de code.
Concernant la bibliothèque HHTPClient, elle est disponible sur Arduino pour l'installer.
Ces deux bibliothèques permettent de gérer la connexion Wi-Fi et les requêtes HTTP.

```
1   #include <WiFi.h>
2   #include <HTTPClient.h>
```

Dans ces 2 lignes, on renseigne le nom du réseau (ici, GalaxyS23), et le mot de passe du réseau (ici, adrienouais).
C'est sur ce réseau défini que l'ESP32 va se connecter.

```
6   const char* ssid = "GalaxyS23";
7   const char* password = "adrienouais";
```

Ici, on spécifie l'adresse URL du serveur auquel le microcontrôleur va envoyer des données. Il faut renseigner l'adresse IP du réseau auquel l'ESP32 est connecté.

```
10   String serverName = "http://192.168.107.97:5000/api";
```

Avant de passer à la partie setup, on initialise la variable “lastTime” qui est utilisée pour chronométrer les envois de données et on règle le timer sur 5000 ms (5 secondes) ce qui signifie qu’il y aura un délai de 5 secondes entre chaque envoie de données. Il faut aussi noter qu’on a utilisé des unsigned long car nous sommes dans l’ordre de grandeur des millisecondes donc un int n’aurait pas été suffisant pour stocker les données.
Également, on ajoute la ligne 14 pour pouvoir interagir avec notre capteur comme expliqué précédemment.

```
12  unsigned long lastTime = 0;
13  unsigned long timerDelay = 5000;
14  SHTSensor sht;
```

Nous allons maintenant passer à la partie setup.

Tout comme le code permettant de mesurer la valeur de la température et de l’humidité, on met la broche vcc sur la broche 14 puis on initialise la bibliothèque Wire et la communication série avec 115200 bauds pour la vitesse de transmission sans oublier le délai de 1 seconde.

```
23  pinMode(14, OUTPUT);
24  digitalWrite(14, HIGH);
25
26  Wire.begin();
27  Serial.begin(115200);
28  delay(1000); // let serial console settle
```

Cette ligne de code initialise notre capteur SHT45 pour qu’il puisse mesurer la valeur de la température et de l’humidité.

```
29  sht.init();
```

Ensuite, on insère ces 2 lignes de code avant la boucle pour établir la connexion de notre réseau sur l’Arduino. Ainsi, il est affiché sur le moniteur série “Connecting” pour dire qu’on tente de se connecter au réseau.

```
31  WiFi.begin(ssid, password);
32  Serial.println("Connecting");
```

Cette boucle while attend que la connexion au réseau soit établie et affiche "." sur le moniteur série pour dire que le processeur est en cours d'exécution. Généralement, lorsqu'il y a un problème au niveau de la carte électronique ou autre, l'Arduino affiche des "." en boucle sur le moniteur série. Il est également possible que cela marche et qu'il y ait quand même une série de points sur le moniteur série. Cela signifie juste que la connexion met un peu de temps à s'établir.

```
33  while(WiFi.status() != WL_CONNECTED) {
34    delay(500);
35    Serial.print(".");
36  }
```

Pour que le moniteur série soit plus facile à lire, on affiche une ligne vide puis un message pour dire que la connexion est établie avec l'adresse IP de l'ESP32

```
37  Serial.println("");
38  Serial.print("Connected to WiFi network with IP Address: ");
39  Serial.println(WiFi.localIP());
```

Enfin, on affiche sur le moniteur série que le timer est réglé sur 5 secondes et c'est après ce temps-là que l'Arduino va prendre ses mesures.

```
41  Serial.println("Timer set to 5 seconds (timerDelay variable), it will take 5 seconds before publishing the first reading.");
42  sht.setAccuracy(SHTSensor::SHT_ACCURACY_MEDIUM); // configurer la précision du capteur SHT (température et humidité)
```

Nous allons maintenant passer à la partie loop() :

Pour que le code s'exécute, on vérifie avec la ligne ci-dessous qu'il y a bien eu 5 secondes depuis le dernier envoi.

```
49  if ((millis() - lastTime) > timerDelay) {
```

Avant d'avancer dans le code, il faut savoir que les instructions seront exécutées uniquement si la connexion au réseau a été faite.
Dans le cas contraire, un message d'erreur s'affichera sur le moniteur série.

```
51    if(WiFi.status()== WL_CONNECTED){
```

```
80    }
81    else {
82      Serial.println("WiFi Disconnected");
83    }
```

Ici, on récupère tout simplement les valeurs mesurées (la température et l'humidité).

```
HTTPClient http;
sht.readSample();
float temp = sht.getTemperature();
float humi = sht.getHumidity();
```

On construit ensuite une URL pour envoyer les données au serveur.
Pour ce faire, on utilise le nom du serveur et on ajoute les valeurs d'humidité et de température en tant que paramètres de requête GET. Cela configure une requête HTTP GET pour envoyer les données au serveur.

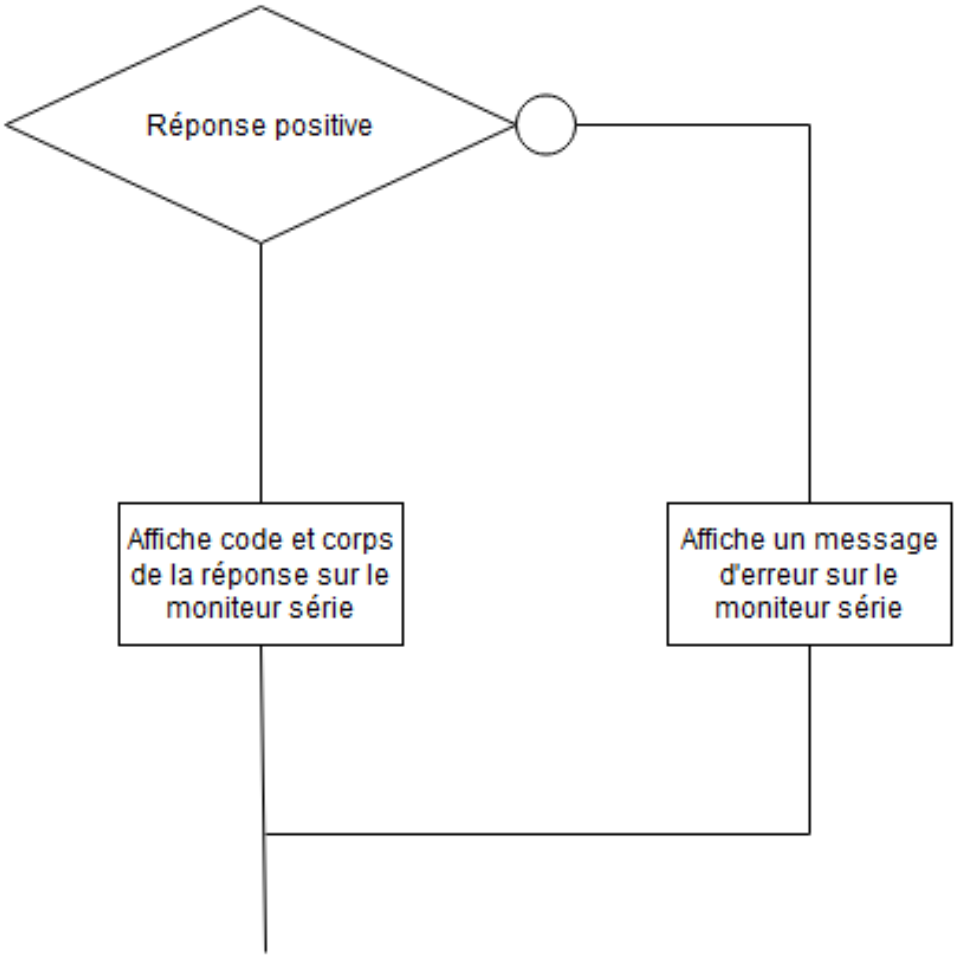
Après la création de l'URL, on initialise la requête en utilisant l'URL et on envoie la requête GET sachant que le résultat est stocké dans httpResponseCode.

```
57  String serverPath = serverName + "?humidity=" + humi + "&temperature=" + temp;
58
59  // Your Domain name with URL path or IP address with path
60  http.begin(serverPath.c_str());
61
62  int httpResponseCode = http.GET();
```

Après l'envoi de la requête, il reçoit et traite le code de la réponse HTTP.
Si la requête est réussie, il imprime le code de réponse et le contenu de la réponse sur le moniteur série (pour nous, c'était 200).
Si la requête n'est pas réussie, il imprime un code d'erreur (pour nous, c'était -1).

```
64      if (httpResponseCode>0) {
65        Serial.print("HTTP Response code: ");
66        Serial.println(httpResponseCode);
67        String payload = http.getString();
68        Serial.println(payload);
69      }
70      else {
71        Serial.print("Error code: ");
72        Serial.println(httpResponseCode);
73      }
74      // Free resources
75      http.end();
76    }
```

Voici un organigramme pour traduire l'instruction if...else utilisée :



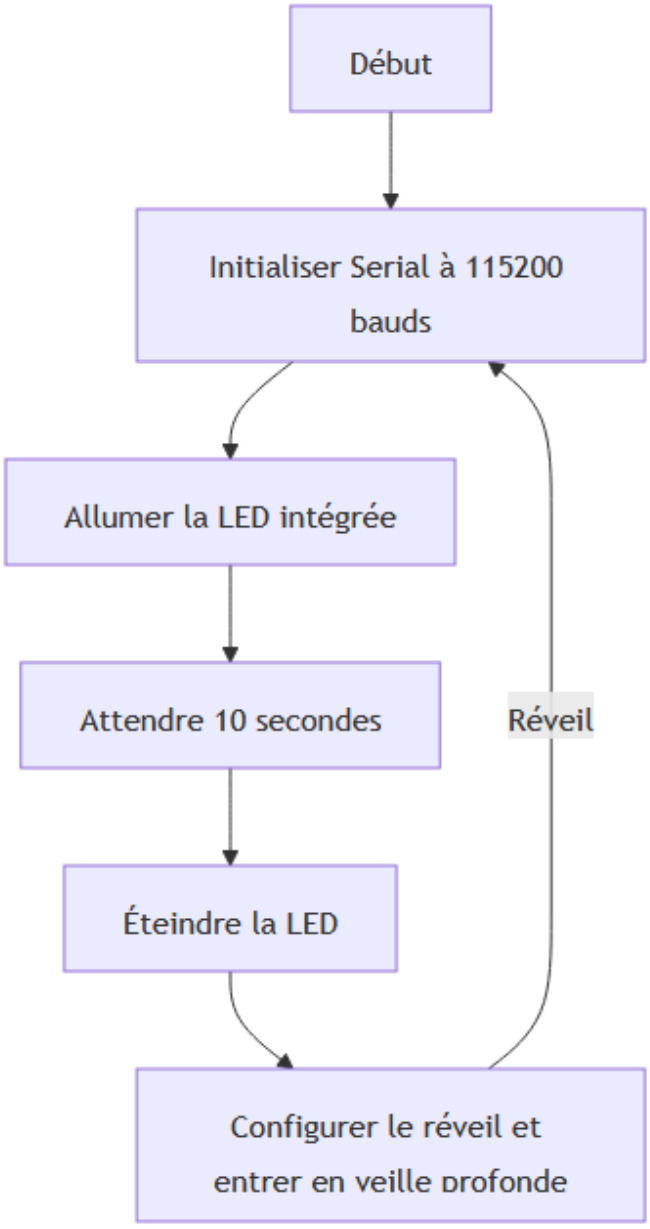
On utilise cette dernière ligne de code pour l'enregistrement de l'heure actuelle pour la synchronisation des cycles futurs :

```
80        lastTime = millis();
```

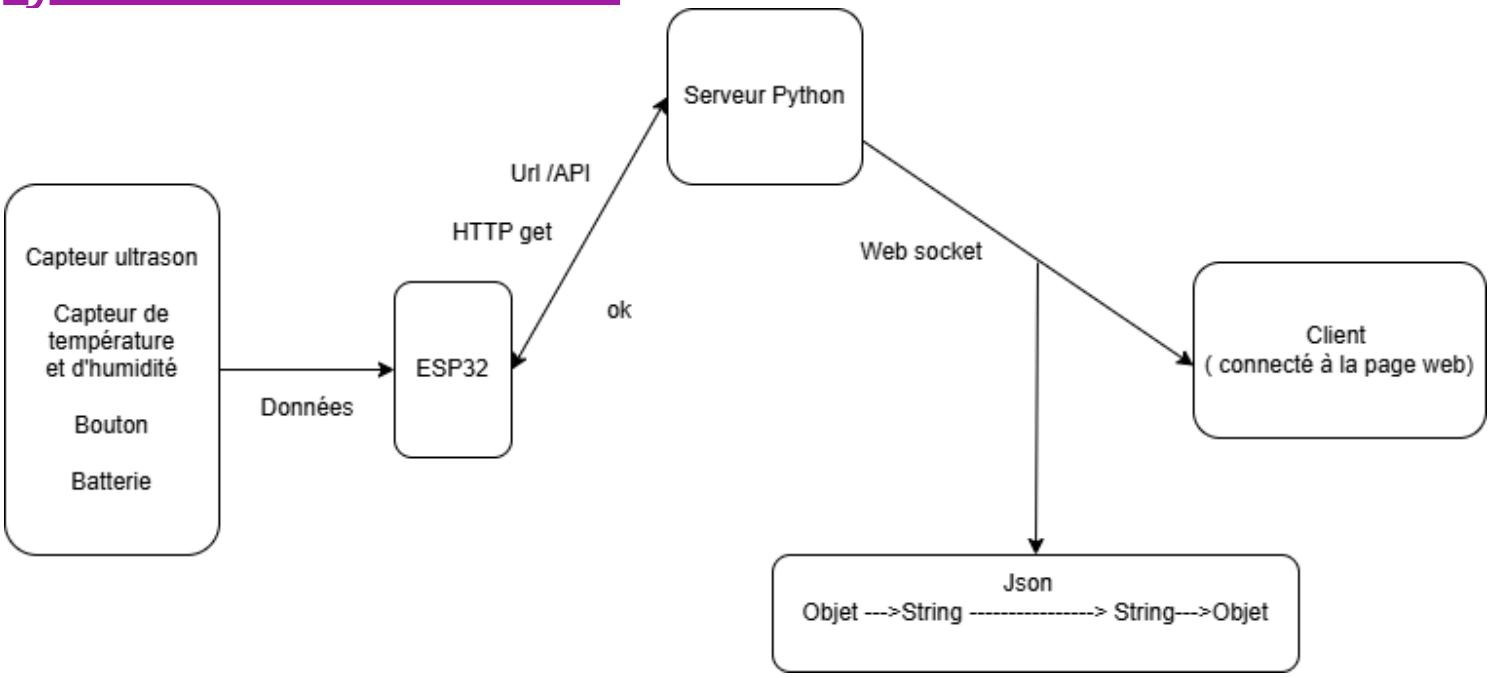
MODE DEEP SLEEP

Lors de l'étude du choix de la batterie, on a mentionné rapidement le mode deep sleep sans l'expliquer.
En effet, ce mode intégré dans l'ESP32 permet de consommer moins d'énergie.
Ainsi, l'ESP32 reste 10 secondes en fonctionnement normal avec la LED allumée, puis bascule en mode deep sleep pendant 10 secondes avec la LED éteinte avant de recommencer le cycle.

Organigramme du programme pour le mode deep sleep :



Communication entre capteurs, esp32, serveur python et Client Web



Ce schéma résume les interactions entre les différents composants du système de surveillance de la cuve. L'Esp32 reçoit les données comme expliqué précédemment et les envoie au serveur python via une requête HTTP GET.

Une requête HTTP GET est une méthode utilisée pour envoyer les données collectées par l'Esp32 au serveur python via une URL spécifique.

Le serveur Python reçoit et enregistre les données envoyées par l'ESP32. En guise de confirmation, il renvoie OK pour indiquer que la transmission a été bien reçue.

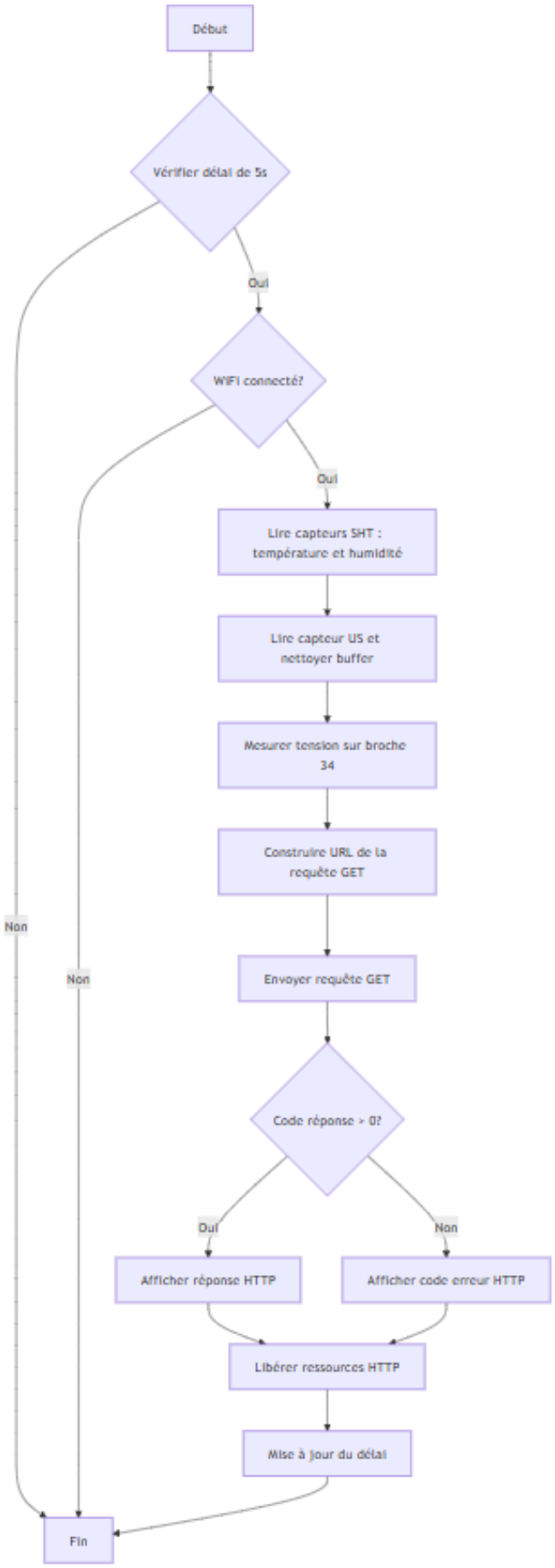
Une fois les données reçues, le serveur Python les envoie en temps réel au client web (page web ouverte sur le navigateur). Cette transmission se fait via une connexion WebSocket.

WebSocket est un protocole de communication bidirectionnelle qui permet de maintenir une connexion ouverte entre le client (page web ouverte sur le navigateur) et le serveur. L'avantage est que nous n'avons pas à rafraichir la page web pour actualiser les valeurs envoyés par l'esp32. Les valeurs sont mises à jour en temps réel.

Le serveur reçoit des données de capteurs (tension, distance, température, humidité) sous forme d'objets. Avant d'envoyer les données aux clients Web, le serveur convertit cet objet en une chaîne JSON. Le client convertit cette chaîne JSON en un objet JavaScript puis les affiche.

Code requête HTTP GET:

Ce schéma résume le fonctionnement du code permettant à l'ESP32 de faire des requêtes HTTP GET au serveur python.



Le programme démarre une fonction qui calcule si suffisamment de temps (au moins 5 secondes) s'est écoulé depuis la dernière requête.

Si oui, le processus continue pour collecter les données et envoyer une nouvelle requête.

Sinon, le programme se termine et attend que le délai soit atteint avant de réessayer.

Puis le système vérifie si l'esp32 est connectée au réseau Wi-Fi.

Si oui, le programme passe à l'étape suivante pour collecter les données.

Sinon, il se termine car il est nécessaire d'avoir une connexion Wi-Fi pour envoyer les données.

Le capteurs SHT45 (température et humidité) est activé pour effectuer des mesures. Les valeurs de température et d'humidité sont extraites et stockées.

Le capteur ultrasonique mesure la distance, et son buffer est nettoyé pour éviter que d'anciennes données créent des erreurs.

La fonction dédiée à la mesure de la tension est appelé. La valeur mesurée est utilisée pour le rapport final.

Une URL est générée pour envoyer les données au serveur.

Cette URL inclut les valeurs collectées des capteurs (température, humidité, distance, tension) sous forme de paramètres de la requête GET.

La requête GET est envoyée au serveur.

- Si le code de réponse est supérieur à 0 (par exemple, 200), la réponse du serveur est affichée dans la console.
- Sinon, un message d'erreur avec le code correspondant est affiché.

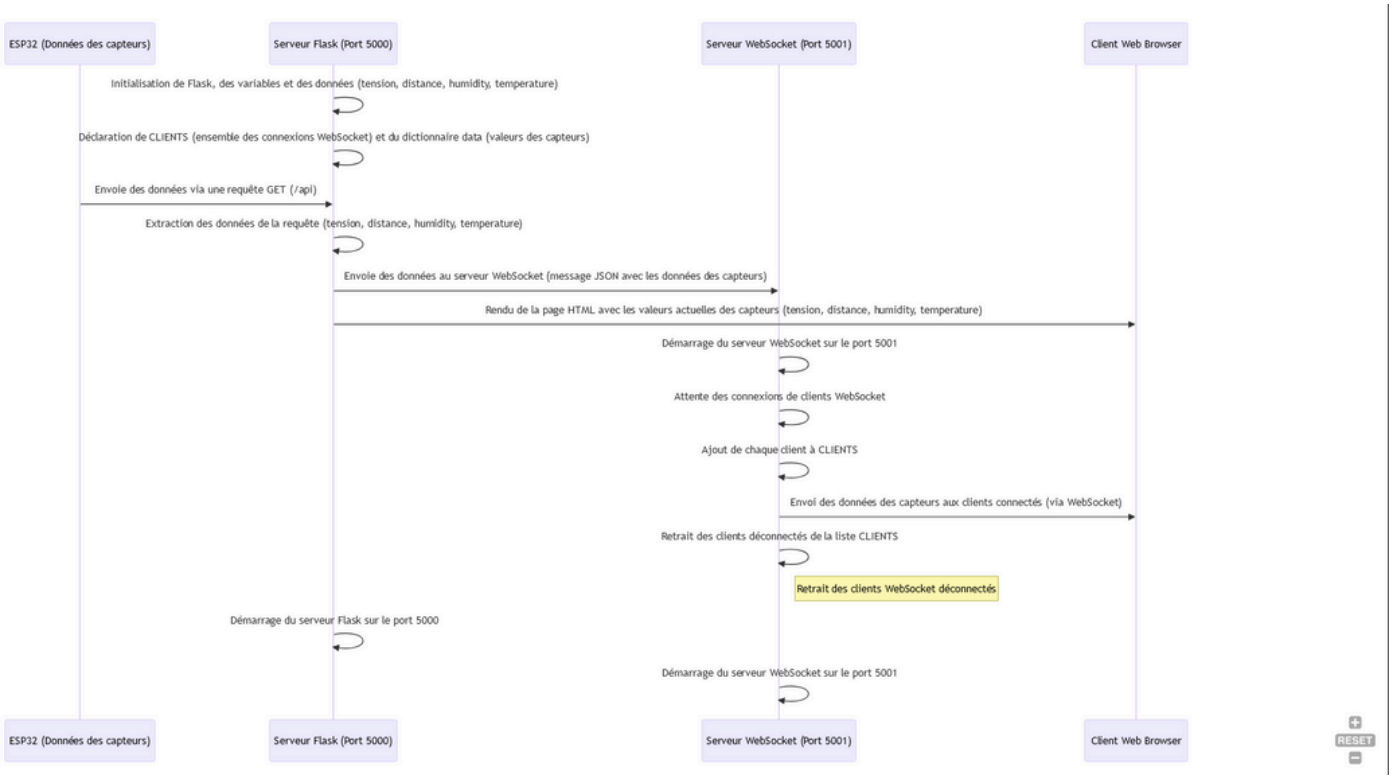
Les ressources associées à la requête HTTP (mémoire et connexions) sont libérées pour éviter tout risque de fuite mémoire.

Le délai de la dernière requête est mis à jour pour permettre le calcul du prochain intervalle de 5 secondes avant la prochaine requête.

Une fois toutes les étapes terminées, le processus se termine. Le programme attend le prochain cycle pour redémarrer la boucle.

Code serveurs python

Ce schéma résume le fonctionnement du code des serveurs python.



On démarre le serveur Flask sur le port 5000. Flask est un microframework càd un ensemble d'outils permettent de faciliter le développement du serveur en python

On initialise des variables pour stocker les données des capteurs (tension, distance, humidity, temperature).

- Création d'un dictionnaire data : Ce dictionnaire est utilisé pour conserver et formater les données reçues.
- Déclaration de la liste des clients WebSocket : Un ensemble (CLIENTS) est utilisé pour enregistré les clients connecté à la page web.

On démarre le serveur WebSocket sur le port 5001 en parallèle.

Le serveur Flask extrait les valeurs des capteurs transmises par l'esp32 dans la requête GET.
Les nouvelles données sont enregistrées dans le dictionnaire pour être accessibles aux autres parties du système.

Le serveur Flask transmet les données au serveur WebSocket sous forme d'un message JSON.

- Cela garantit que toutes les mises à jour sont disponibles pour les navigateurs web connectés.

Lorsque des clients (navigateurs web) se connectent au serveur WebSocket, ils sont ajoutés à la liste CLIENTS.

Envoi des données mises à jour : Le serveur WebSocket envoie le message JSON contenant les valeurs des capteurs à tous les clients connectés.

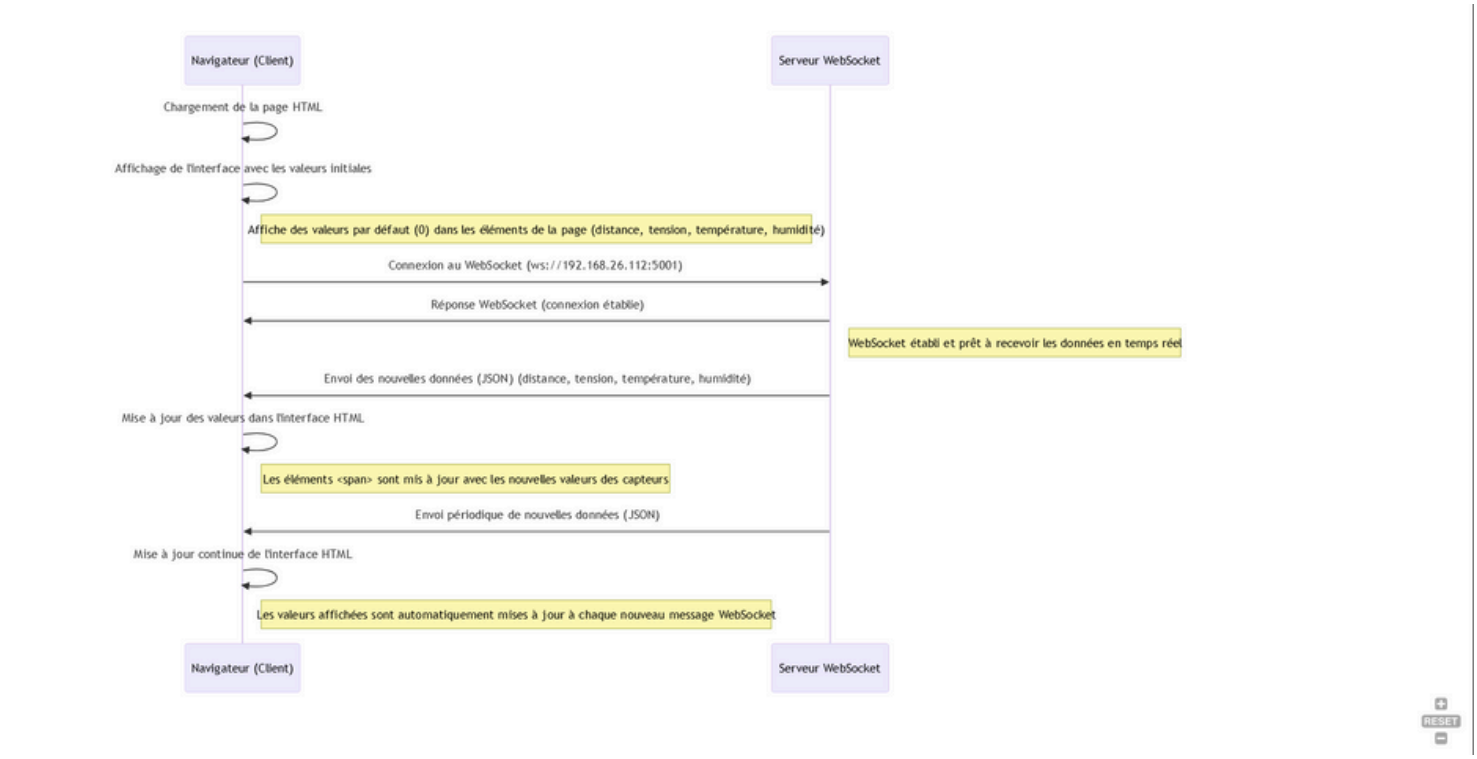
Les navigateurs web reçoivent ces messages pour mettre à jour leur interface utilisateur en temps réel.

- Si un client se déconnecte, le serveur WebSocket détecte la déconnexion et retire le client de la liste CLIENTS.
- Cela évite d'envoyer des données inutiles à des clients inactifs.

Le navigateur reçoit les données via WebSocket et les convertit en objet Javascript.
Les nouvelles valeurs des capteurs sont affichées dynamiquement sans avoir besoin de recharger la page.

Code page html

Ce schéma résume le fonctionnement du code des serveurs python.



La page HTML est chargé par le navigateur web du client.

- Structure de l'interface : La page HTML inclut des éléments pour afficher des informations telles que la distance, la tension, la température, et l'humidité. Ces éléments apparaissent avec des valeurs par défaut.

Un fichier CSS est utilisé pour styliser l'interface.

Connexion WebSocket initiée par le navigateur :

- Lorsqu'un utilisateur ouvre la page, le navigateur tente de se connecter au serveur WebSocket via l'adresse spécifiée.
- Une fois la connexion établie, le serveur WebSocket est prêt à envoyer des mises à jour en temps réel au client.

Messages envoyés par le serveur WebSocket :

- À chaque mise à jour des données (distance, tension, température, humidité), le serveur WebSocket envoie un message JSON au navigateur.
- Ce message contient les nouvelles valeurs des capteurs.

Traitement des données reçues :

- Les messages reçus sont convertis en objet Javascript.
- Chaque valeur est extraite et insérée dans les éléments correspondants de la page (par exemple, "Distance", "Tension").

Modification en temps réel :

- Sans recharger la page, les nouvelles valeurs sont directement affichées dans l'interface utilisateur.
- Les éléments HTML (par exemple, les paragraphes indiquant "Distance", "Tension") sont modifiés pour refléter les dernières données reçues.

Expérience utilisateur améliorée :

- Le système garantit une visualisation en temps réel des données des capteurs, offrant une expérience interactive et fluide.

CRÉATION DE LIBRAIRIES

Dans le but de simplifier le programme final, on a créé plusieurs librairies permettant ainsi de réduire considérablement le programme. Pour cela, il faut créer un fichier .cpp dans lequel on écrit la fonction et un fichier .h dans lequel on intègre l'implémentation de la fonction.

1) Librairie GetTension :

- **GetTension.cpp :**

```cpp
#include "GetTension.h"

// Fonction pour lire et afficher la tension de la broche 34
void GetTension() {
  int valeurAnalog = analogRead(34);              //récupérer la tension
  float tension = (float)2 * valeurAnalog / 1000;  //convertir la valeur de la tension en décimal
  Serial.print("Tension sur 34 : ");              //envoyer sur le port série le texte "Tension sur 34 : "
  Serial.println(tension);                        //envoyer la valeur de la tension
}
```

- **GetTension.h :**

```cpp
#ifndef GetTension_h
#define GetTension_h

#include <Arduino.h>

// Déclaration de la fonction
void GetTension();

#endif
```

- **Code batterie avec librairie :**

```cpp
#include "GetTension.h"

void setup() {
  Serial.begin(9600);  //envoie de la variable "tension" sur le port série
}

void loop() {
  GetTension();
  delay(1000);  //ajout d'un délai de 1 seconde afin d'éviter que l'arduino spam le port série
}
```

On a juste à faire l'appel de la fonction dans le loop() et on voit que le code n'est pas du tout consistant.

2) Librairie Deepsleep :

- **Deepsleep.cpp :**

```cpp
#include "deepsleep.h"
// Convertir des microsecondes en secondes
#define uS_TO_S_FACTOR 1000000ULL
// Durée de chaque cycle (deep sleep et réveil)
#define TIME_TO_SLEEP  10

void deepsleep() {
  // Délai de TIME_TO_SLEEP secondes
  delay(TIME_TO_SLEEP*1000);

  // Afficher un message avant le passage en mode deep sleep
  Serial.println("Go to deep sleep");

  // Définir la durée du mode deep sleep
  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);

  // Passer en mode deep sleep
  esp_deep_sleep_start();
}
```

- **Deepsleep.h :**

```cpp
#ifndef deepsleep_h
#define deepsleep_h

#include <Arduino.h>

void deepsleep();

#endif
```

- **Code deep sleep avec librairie**

```cpp
#include "deepsleep.h"

// Setup() est appelé au démarrage et au réveil du mode deep sleep
void setup() {

  // Afficher un message dans la console
  Serial.begin(115200);
  Serial.println("Wake up");
  deepsleep();
}

void loop() {}
```

3) Librairie bouton LED

- **BoutonLed.cpp :**

```cpp
#include "BoutonLed.h"

void BoutonLed() {
int valeurAnalog = analogRead(36);
float tension = (float) valeurAnalog;
Serial.print("Tension sur 36 : ");
Serial.println(tension);
if(tension == 0){
 digitalWrite(LED_BUILTIN, HIGH);
}
else{
 digitalWrite(LED_BUILTIN, LOW);
}

}
```

- **BoutonLed.h :**

```cpp
#ifndef BoutonLed_H
#define BoutonLed_H
#include <Arduino.h>

void BoutonLed(void);

#endif
```

- **Code bouton LED avec librairie**

```cpp
#include "BoutonLed.h"

void setup() {
Serial.begin(9600);
pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
BoutonLed();
}
```

SUIVI DU PROJET AVEC LES LIVRABLES

M. Lucidarme nous a confié 6 livrables à réaliser durant le projet et ils ont tous été réalisés avec succès.

Livrable 1	**Assemblage de la carte**	
Livrable 2	**Librairie capteur US**	
Livrable 3	**Librairie LED / Batterie**	
Livrable 4	**Capteur Température**	
Livrable 5	**WiFi**	
Livrable 6	**Serveur Web**	

CONCLUSION

En conclusion, à l'issu de ce projet, nous avons accumulé pas mal de connaissances. On a découvert le monde de l'interface web qui était une notion nouvelle pour nous. Nos compétences avant ce projet étaient plutôt orienté au niveau du hardware mais on a su s'adapter et ainsi renforcé notre compétence en software, un aspect essentiel de notre parcours en GEII. Rédiger un cahier des charges et bien gérer notre temps ont été super importants.

Avec Arduino, le logiciel de programmation, on a élargi notre palette de compétences, ouvrant de nouvelles perspectives dans la programmation embarquée. Bien gérer les tâches en binôme a joué un rôle clé dans notre réussite commune. Refaire cette expérience avec une approche plus réfléchie dès le début aurait permis une meilleure répartition des tâches, libérant plus de temps pour la programmation.

Annexe 1 : code librairie wifi_requet.cpp :

```cpp
#include "wifi_requet.h"  // Inclusion de la bibliothèque personnalisée pour les requêtes WiFi
const char* ssid = "Houbix";  // Nom du réseau WiFi
const char* password = "12345678";  // Mot de passe du réseau WiFi

// Nom de domaine ou adresse IP du serveur avec le chemin d'accès API
String serverName = "http://192.168.228.112:5000/api";

// Les variables suivantes sont des unsigned longs car le temps, mesuré en millisecondes,
// deviendra rapidement un nombre trop grand pour être stocké dans un int.
unsigned long lastTime = 0;  // Variable pour stocker le temps de la dernière requête
// Temporisation réglée à 10 minutes (600000 millisecondes)
// unsigned long timerDelay = 600000;
// Réglage du temporisateur à 5 secondes (5000 millisecondes)
unsigned long timerDelay = 5000; // Temporisation de 5 secondes entre chaque requête

// Configuration des capteurs
sen0311 us(2, 4, 16);  // Initialisation du capteur ultrasonique avec les broches appropriées
SHTSensor sht;  // Initialisation du capteur SHT pour la température et l'humidité

// Fonction pour configurer le WiFi et initialiser le matériel
void wifi() {
  pinMode(14, OUTPUT);  // Configuration de la broche 14 comme sortie
  digitalWrite(14, HIGH);  // Mise en état haut de la broche 14

  Wire.begin();  // Initialisation du bus I2C
  // Configuration de la broche 12 en mode sortie
  pinMode(12, OUTPUT);

  delay(500);  // Attente de 500 millisecondes
  sht.init();  // Initialisation du capteur SHT
  // Activation du capteur ultrasonique en envoyant un signal sur la broche 12
  digitalWrite(12, HIGH);

  WiFi.begin(ssid, password);  // Connexion au réseau WiFi
  Serial.println("Wake up");  // Message de réveil
  Serial.println("Connecting");  // Message de tentative de connexion
  while (WiFi.status() != WL_CONNECTED) {  // Boucle jusqu'à ce que le WiFi soit connecté
    delay(500);
    Serial.print(".");
  }
  Serial.println("");  // Nouvelle ligne pour la clarté
  Serial.print("Connected to WiFi network with IP Address: ");  // Message une fois connecté
  Serial.println(WiFi.localIP());  // Affichage de l'adresse IP locale

  Serial.println("Timer set to 5 seconds (timerDelay variable), it will take 5 seconds before publishing the first reading.");
  sht.setAccuracy(SHTSensor::SHT_ACCURACY_MEDIUM);  // Configuration de la précision du capteur SHT
}

```

```
void requet() {
  // Envoi d'une requête HTTP GET toutes les 5 secondes
  if ((millis() - lastTime) > timerDelay) {  // Vérification si le delay est écoulé
    // Vérifie si le WiFi est toujours connecté
    if (WiFi.status() == WL_CONNECTED) {
      HTTPClient http;  // Création de l'objet HTTPClient

      sht.readSample();  // Lecture de l'échantillon de données depuis le capteur SHT
      float temp = sht.getTemperature();  // Récupération de la température
      float humi = sht.getHumidity();  // Récupération de l'humidité
      int dist;  // Déclaration de la variable pour la distance

      // Nettoyage des données du capteur ultrasonique
      us.flushUs();

      // Obtention de la distance mesurée par le capteur ultrasonique
      dist = us.getDistance();

      float tension = GetTension();  // Lecture de la tension (fonction hypothétique non définie ici)

      // Construction de l'URL avec les paramètres de la requête
      String serverPath = serverName + "?tension=" + tension + "&distance=" + dist + "&humidity=" + humi + "&temperature=" + temp;

      // Initialisation de la requête HTTP GET avec l'URL complète
      http.begin(serverPath.c_str());

      // Envoi de la requête HTTP GET
      int httpResponseCode = http.GET();

        // Vérification du code de réponse HTTP
        if (httpResponseCode > 0) {
          Serial.print("HTTP Response code: ");  // Affichage du code de réponse HTTP
          Serial.println(httpResponseCode);
          String payload = http.getString();  // Récupération du corps de la réponse
          Serial.println(payload);  // Affichage du contenu de la réponse
        } else {
          Serial.print("Error code: ");  // Message en cas d'erreur
          Serial.println(httpResponseCode);
        }

        http.end();  // Libération des ressources de la requête HTTP
      } else {
        Serial.println("WiFi Disconnected");  // Message en cas de déconnexion du WiFi
      }
      lastTime = millis();  // Mise à jour du temps de la dernière requête
      deepsleep();  // Mise en veille profonde (fonction non définie dans ce code)
    }
  }
```

Annexe 2 : code librairie wifi_requet.h :

```
1   #ifndef wifi_requet_h
2   #define wifi_requet_h
3   #include <Arduino.h>
4   #include <WiFi.h>
5   #include <HTTPClient.h>
6   #include "sen0311.h"
7   #include "SHTSensor.h"
8   #include "GetTension.h"
9   #include "deepsleep.h"
10
11  void wifi();
12  void requet();
13
14  #endif
```

Annexe 2 : code final :

```
#include "wifi_requet.h"  // Inclusion de la bibliothèque wifi_requet

void setup() {
  Serial.begin(115200);  // Initialisation de la communication série à une vitesse de 115200 bauds pour l'affichage des messages
  wifi();  // Appel de la fonction wifi()
}

void loop() {
  requet();  // Appel de la fonction requet()
}
```