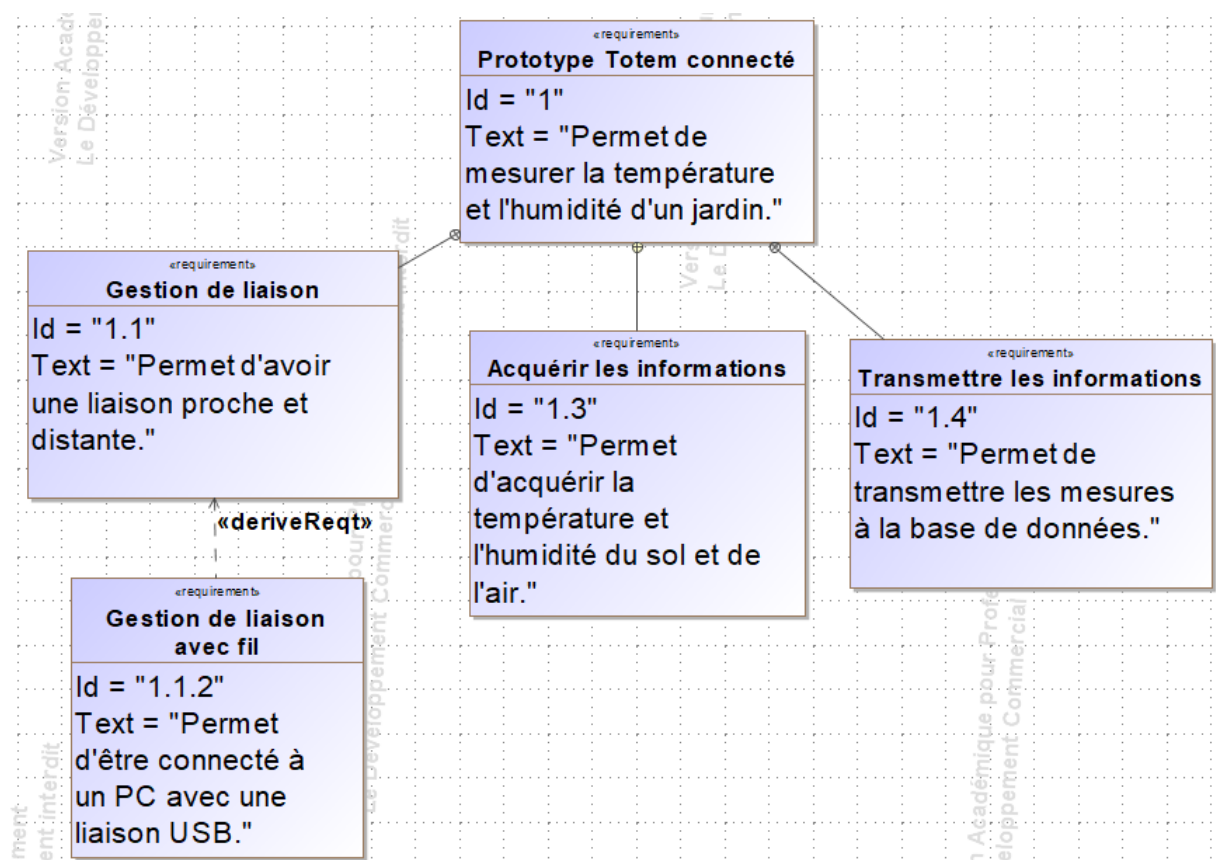


## 2 DUGELAY Enzo – Création d'une IHM PC en liaison série (RS232)

### 2.1 Présentation

Ma partie consiste à la création d'une IHM (Interface Homme/Machine) PC en liaison série (RS232), le but de cette partie est de mesurer la température et l'humidité du sol d'une plante afin d'envoyer ces valeurs dans une base de données et de pouvoir les utiliser afin de donner des indications à l'enfant si la température et l'humidité sont trop hautes ou trop basses.

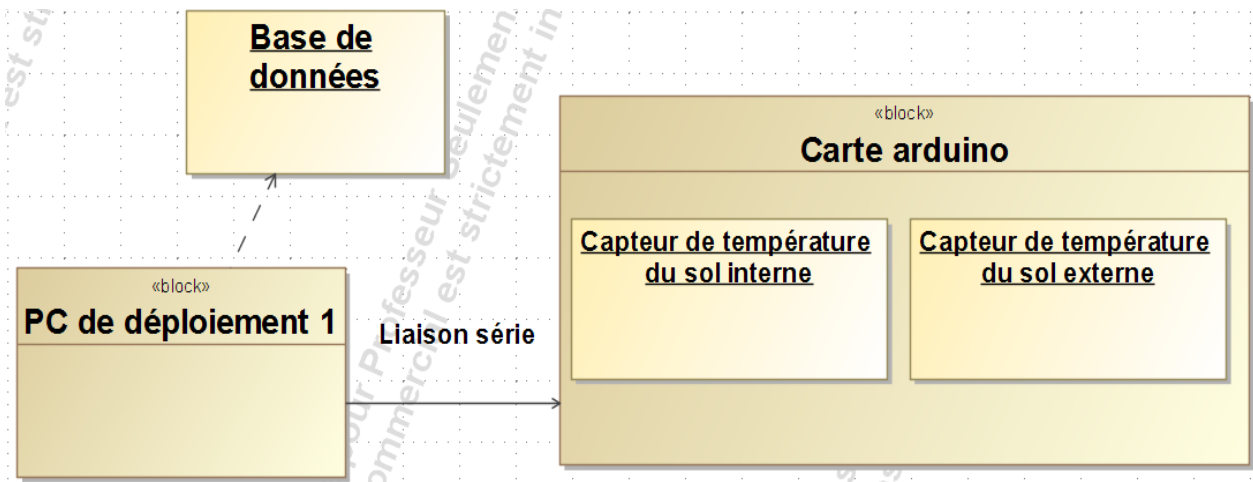
Mon objectif est de concevoir un système répondant à des exigences clairement définies. Pour cela, j'ai utilisé un diagramme d'exigences qui présente toutes les fonctionnalités et les caractéristiques requises pour le système.



Le diagramme d'exigences décrit les fonctionnalités et les exigences du système. Le système doit être branché avec une liaison USB à un PC et doit être capable de mesurer la température et l'hygrométrie du sol. Les mesures doivent être acquises à l'aide des capteurs et transmises à une base de données afin de les stockées.

## 2.2 Analyse préliminaire

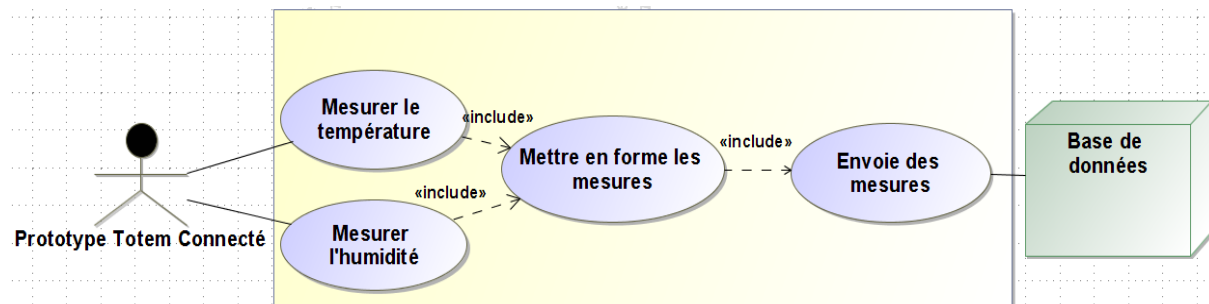
Pour concevoir le système, j'ai réalisé un diagramme de bloc interne qui m'a permis de visualiser les différents blocs fonctionnels et leurs interactions d'un point de vue matériel.



Ce diagramme de bloc interne nous permet de comprendre comment est représenté le système dans ma partie. Il y a donc un ordinateur qui contient mon IHM PC qui est relié en liaison USB à une carte Arduino, elle contient un capteur de température interne et un capteur d'humidité externe.

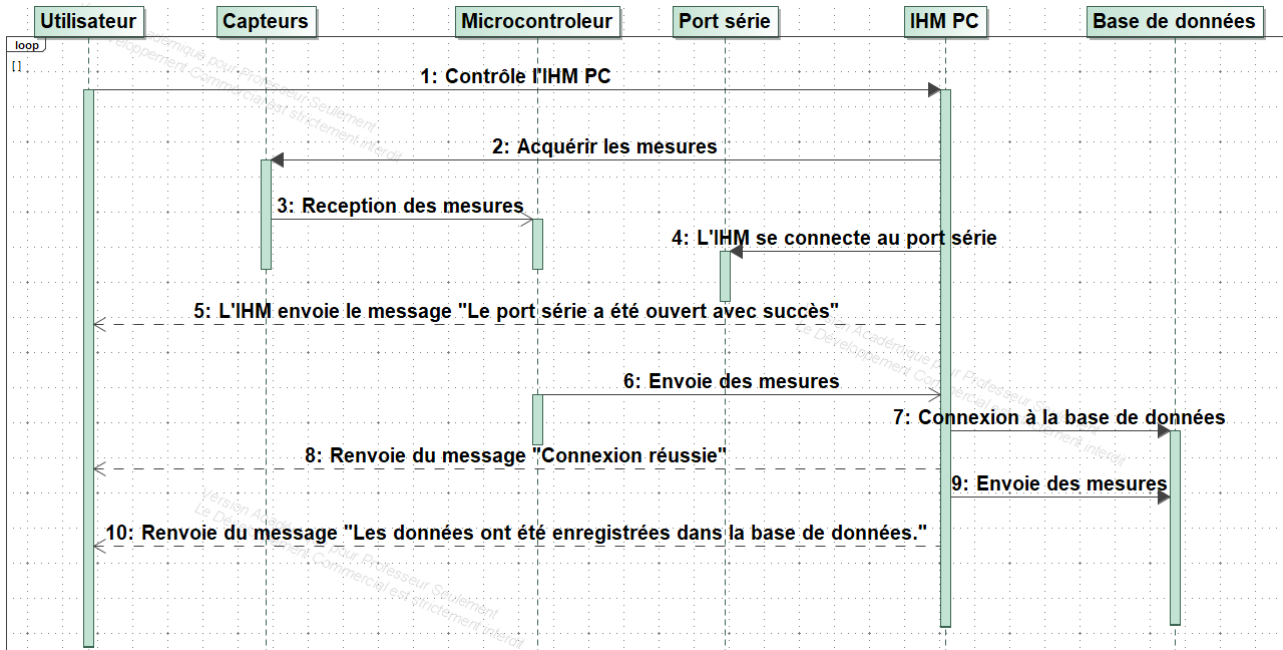
Dans ce système, une liaison physique USB standard est utilisée pour connecter le PC à la carte Arduino. Cependant, une émulation de la liaison série RS232 a été codée afin de simuler une communication série entre les deux dispositifs.

On peut ensuite retrouver les différentes fonctionnalités du système par un diagramme de cas d'utilisation afin de visualiser les interactions entre les acteurs et le système.



Ce diagramme de cas d'utilisation représente le système Prototype Totem Connecté qui est relié à l'acquisition de la température et de l'humidité. Ces valeurs sont ensuite mises en forme puis envoyées à l'acteur Base de données.

Afin de visualiser les interactions entre les différents éléments du système, on utilise un diagramme d'utilisation qui nous sera utile pour comprendre la chronologie des événements et les échanges entre les différents acteurs du système.



Ce diagramme de séquence nous permet donc de comprendre la chronologie des événements du système. On a l'utilisateur qui contrôle l'IHM PC, puis l'IHM PC qui ordonne la reception des mesures aux capteurs. Pour finir le microcontrôleur envoie les mesures à l'IHM PC qui les distribue à la base de données.

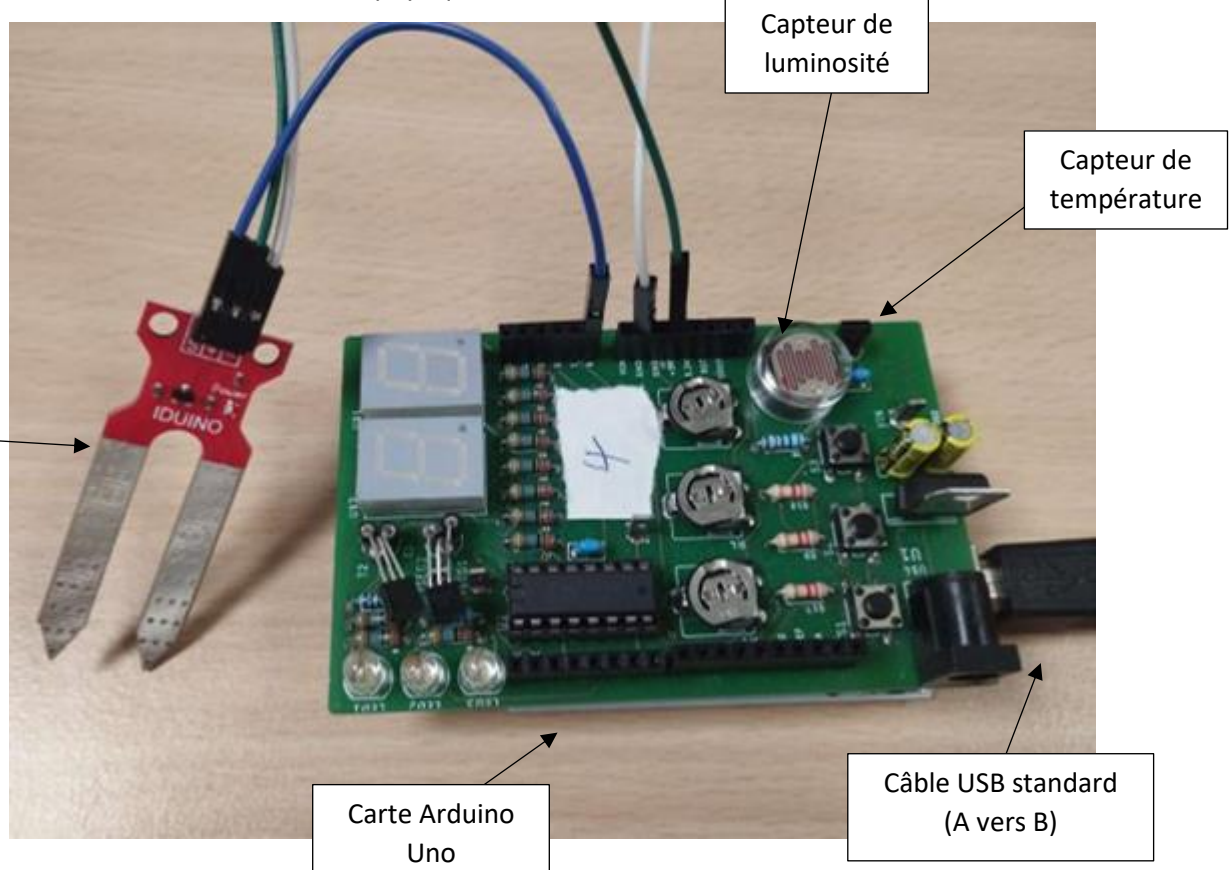
## 2.3 Ressources utilisées

Pour mener à bien ma partie de projet, j'ai utilisé différents outils matériels et physiques. Initialement, on nous avait imposé des ressources matérielles et logicielles tel qu'utiliser une carte Raspberry Pi avec une carte SenseHat, que l'on devait utiliser Windows 10 comme OS ainsi que le logiciel de développement QT Creator 5.2 qui devait être installé sur la carte Raspberry Pi.

J'ai décidé de ne pas utiliser la carte Raspberry Pi mais de la remplacer par une carte Arduino pour diverses raisons :

- Coût : Les cartes Arduino sont généralement moins chères que les cartes Raspberry Pi (environ 20/30€ contre 35/45€)
- Faible consommation d'énergie : Les cartes Arduino ont une consommation d'énergie très faible par rapport aux cartes Raspberry Pi (environ 50mA contre 300mA à 500mA)
- Taille compacte : Les cartes Arduino ont souvent une taille plus petite que les cartes Raspberry Pi
- Facilité d'utilisation : La programmation en Arduino est plus simple que celle en Python sur Raspberry Pi

Visualisation des ressources physiques :



Le fil bleu est branché à la borne de sortie S du capteur et est relié à l'entrée de la carte.

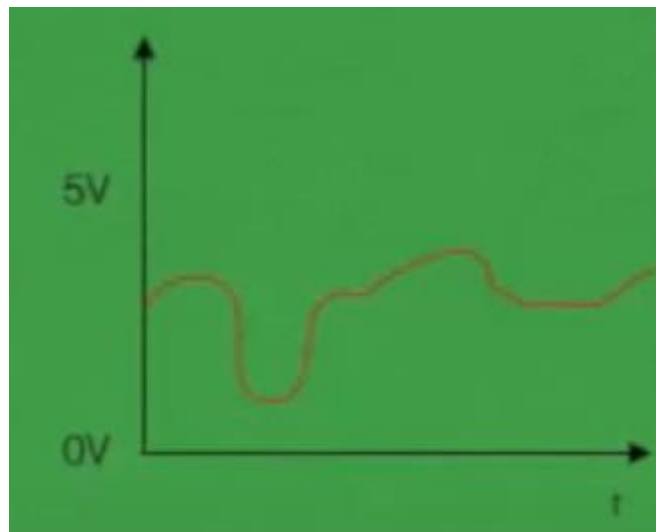
Le fil vert est branché à la borne + du capteur et est relié à la tension 5V de la carte.

Le fil blanc est branché à la borne – du capteur et est relié à GND (ground) qui représente la masse.

Le capteur d'humidité est le capteur de sol Iduino ME110 avec comme référence : me1104053199514540, qui a une tension d'entrée qui est acceptée entre 3,3 volts et 5 volts et de courant maximum 20 milliampères, le capteur coûte un peu moins de 2€.

Le capteur de température est un capteur TMP36, il coûte 2€ environ aussi, il doit être alimenté entre 2,7 volts et 5,5 volts, il consomme moins de 50 microampères et renvoie en tension de sortie entre 0,1 volt et 2 degrés.

D'un point de vue physique, le capteur renvoie au microcontrôleur d'arduino des tensions qui représentent les valeurs analogiques de la température par exemple, que l'on peut représenter sous la forme d'une courbe comme celle-ci, qui traduit la tension entre 0 et 5 volts au cours du temps. Le montage envoie donc une valeur analogique qui varie dans le temps.






Ensuite, le convertisseur analogique/numérique est intégré au microcontrôleur. Ce CAN permet de convertir les signaux analogiques en valeurs numériques que le microcontrôleur pourra traiter. Il fonctionne en échantillonnant le signal analogique à des intervalles de temps régulières et en quantifiant ces échantillons pour les présenter ensuite sous forme numérique. La résolution du CAN détermine le nombre de niveaux discrets dans lesquels le signal analogique peut être quantifié. Ici, nous avons un CAN de 10 bits qui nous permet d'obtenir  $2^{10}$  niveaux de quantification/valeur décimales, ce qui correspond à 1024 valeurs, allant de 0 à 1023.

On peut aussi calculer le quantum qui va nous servir à connaître la précision de notre capteur. On divise 100 par le nombre de bits possible  $2^{10}$  donc 1024 bits, ce qui nous donne  $100/1024 = 0,098$ . On suppose donc que le capteur d'humidité a une précision à moins de 0,1% près, ce qui nous donne des valeurs très précises pour un capteur qui coûte 2€.

Donc, le convertisseur analogique/numérique va traduire la valeur du capteur entre 0 et 5 volts puis traduit cette valeur sous forme de 10 bits. Ensuite, la partie pour convertir ces 10 bits se trouve dans le code que l'on va analyser plus tard.

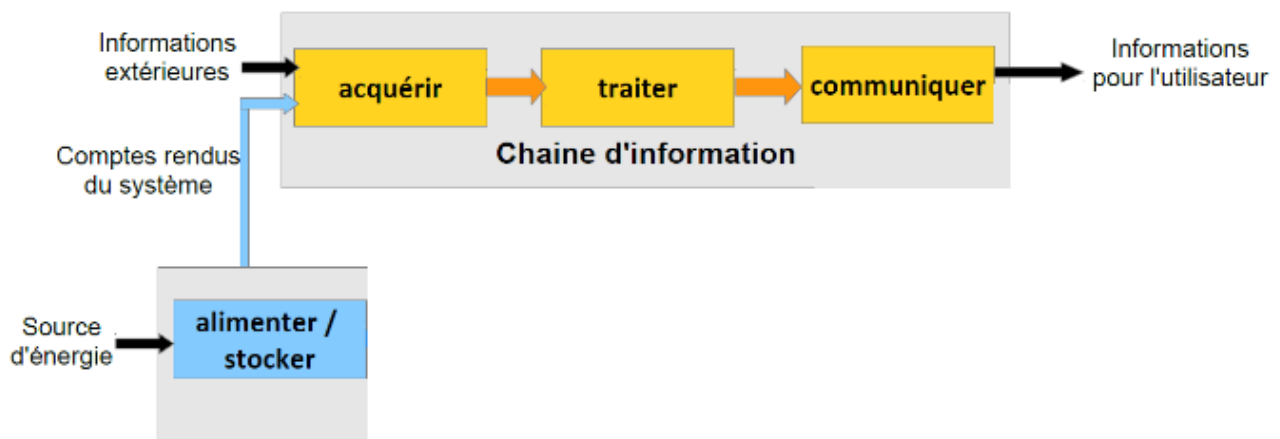
Les outils logiciels que j'ai utilisé lors de ma partie sont donc :

Logiciel	Utilisation
 <b>QT Creator</b>	Logiciel de développement en C++.
 <b>Arduino</b>	Logiciel de développement en C et en C++.
 <b>Magicdraw</b>	Outil de modélisation pour les diagrammes.

Pour réaliser ce travail, voici les tâches à réaliser :

- Installer et câbler les différents capteurs à la carte Arduino
- Mesurer l'hygrométrie et la température du sol
- Mettre en forme les mesures
- Récupérer les mesures avec QT Creator
- Transmettre les mesures à la base de données

La partie que j'ai en charge concerne la chaîne d'information qui est composée des fonctions alimenter, acquérir, traiter et communiquer.



Dans la chaîne d'information, la fonction « alimenter » est représentée par le câble USB qui va alimenter la carte Arduino à partir de l'ordinateur. La fonction « acquérir » consiste à mesurer l'hygrométrie et la température du sol à l'aide des capteurs. Ensuite, la fonction « traiter » consiste à mettre en forme les mesures acquises afin de faciliter leur traitement. Pour finir, la fonction « communiquer » permet la transmission de la trame que l'on va récupérer et analyser avec QT Creator.

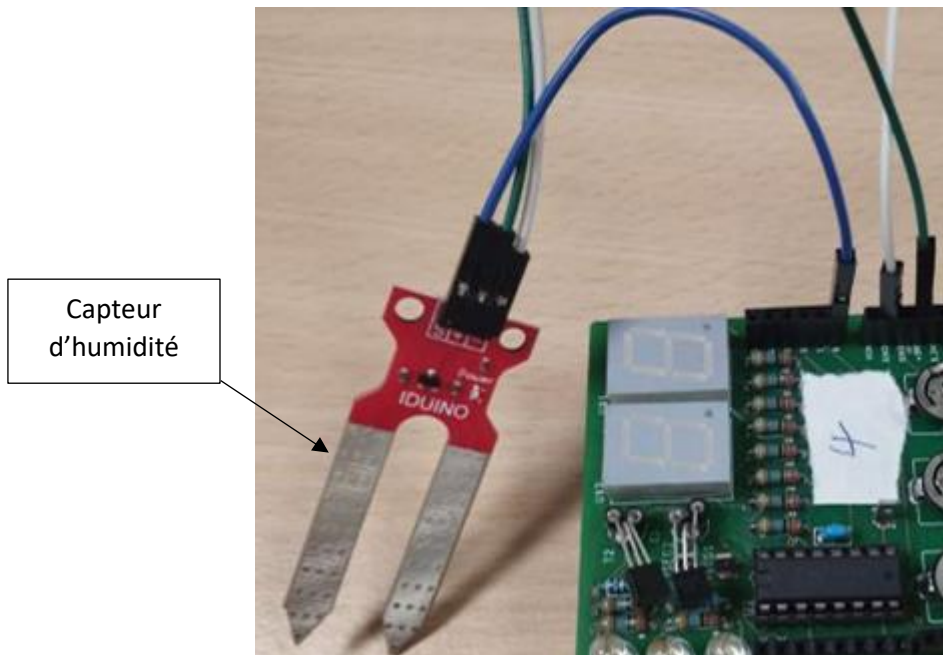
La suite de la chaîne d'information est la chaîne d'énergie (distribuer, convertir, transmettre et agir), mais ce ne sont pas des étapes que je dois prendre en compte dans ma partie.

## 2.4 Réalisation et développement de la trame sous Arduino

### 2.4.1 Réalisation de la trame

Dans un premier temps, à l'aide du logiciel de programmation Arduino il faut pouvoir récupérer la température et l'humidité à l'aide des capteurs pour ensuite configurer une trame qui sera composée de ces mêmes valeurs, afin de les récupérer avec Qt Creator.

Pour récupérer ces valeurs, il faut tout d'abord configurer les pins que je vais utiliser pour brancher le capteur d'humidité.



Le capteur d'humidité est donc alimenté et sa borne de sortie S est reliée à l'aide du câble bleu à la sortie analogique A0. Avec le code suivant, on va donc définir les sorties aux pins correspondant.

```
#define temperaturePin 4    // Définition de temperaturePin au pin 4
#define humidityPin A0      // Définition de humidityPin au pin A0
```

Le capteur de température est donc configuré sur le pin 4 et le capteur d'humidité est relié au pin A0.

Ensuite j'ai utilisé la fonction void setup() qui est la première fonction qui s'exécute dans le code et elle ne s'exécute qu'une seule fois. Elle contient généralement les instructions qui définissent à quels pins sont reliés les sorties et les entrées de la carte arduino (OUTPUT et INPUT) et aussi pour définir le débit de communication en nombre de bit par seconde (en baud) pour communiquer avec l'ordinateur. Ici, je défini donc le débit de communication à 9600 bauds.

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

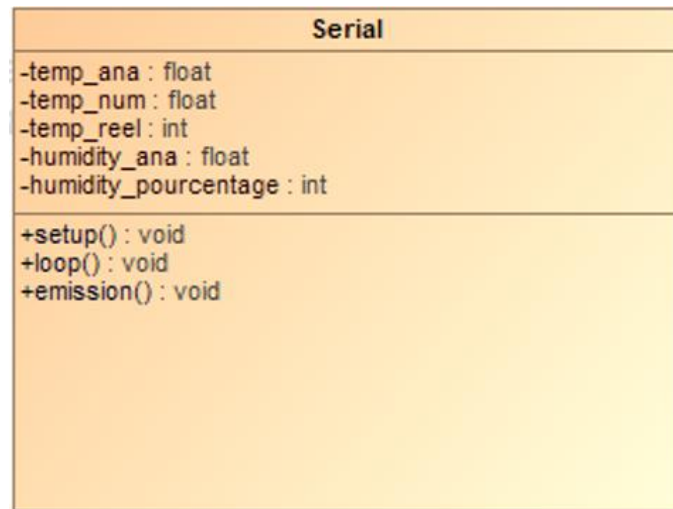
void loop()
{
    emission();    // Initialisation de la fonction emission
}
```



La fonction void loop() s'exécute après la fonction void setup(), c'est une fonction qui est censée s'exécuter à l'infini comme une boucle qui va se répéter tant que la carte est sous tension.

On l'utilise donc pour créer la fonction emission(), qui est la fonction qui va nous servir à récupérer la température, l'humidité, puis on les convertit pour ensuite créer la trame.

On peut organiser les fonctions que l'on a déclarées dans un diagramme de classe.



Dans la fonction emission(), il y a donc les variables que l'on va déclarer :

- Variables pour convertir la température :

```
// Variables pour convertir la température
float temp_ana, temp_num;
int temp_reel;
```

Les valeurs de la température en analogique et en numérique sont déclarées en float afin d'avoir les chiffres après la virgule. Mais la température réelle que je vais l'afficher dans la trame est en int, donc n'aura pas les chiffres après la virgule car on n'a pas l'utilité de connaître si la température est de 24,2 °C ou si elle est de 24,4°C.

- Variables pour convertir l'humidité

```
// Variables pour convertir d'humidité
float humidity_ana;
int humidity_pourcentage;
```

Comme pour les variables pour convertir l'humidité, j'ai besoin de déclarer l'humidité analogique afin d'avoir les chiffres après la virgule donc je déclare humidity\_ana en float. Mais je n'ai pas besoin d'avoir les chiffres après la virgule pour le pourcentage de l'humidité donc je déclare humidity\_pourcentage en int.



Ensuite, temp\_num et humidity\_ana vont me servir à récupérer la température en numérique, et à récupérer l'humidité en analogique. J'utilise la fonction analogRead() avec le nom des pins que j'ai défini précédemment.

J'ai donc le code suivant :

```
temp_num = analogRead(temperaturePin);  
  
humidity_ana = analogRead(humidityPin);
```

Il faut maintenant convertir les valeurs que l'on vient de récupérer.

Conversion des valeurs récupérées :

- Conversion de la température

```
temp_num = analogRead(temperaturePin);  
temp_ana = (5 * temp_num) / 1023;  
temp_reel = (temp_ana - 0.5) * 100;
```

Ici, je convertis la valeur numérique en valeur analogique grâce à la formule  $(5 * \text{valeur numérique}) / 1023$ , où 5 est la tension maximale 5 volts, et où 1023 correspond aux nombres de valeurs différentes sur 10 bits donc 1024, -1 donc 1023.

Ensuite, je convertis la valeur analogique en valeur réelle, avec la formule  $(\text{valeur analogique} - 0,5) * 100$ . La formule utilisée suppose que le capteur de température fournit une sortie linéaire avec une tension de 0,5 volt correspondant à 0 °C. Je soustrais donc 0,5 à la valeur analogique puis je multiplie par 100 pour obtenir enfin la température réelle en degré Celsius.

La tension de sortie de ce capteur varie entre 0,1 volt et 2 volts ce qui correspond à -40°C pour 0,1 volt et 150°C pour 2 volts.

- Conversion de l'humidité

```
humidity_ana = analogRead(humidityPin);  
humidity_pourcentage = map(humidity_ana, 0, 1023, 0, 100);
```

Pour l'humidité, j'utilise la fonction map, qui est utilisée de cette forme :

```
map(nom, valeur minimale, valeur maximale, 0, 100) ;
```

Cette fonction va me permettre de créer un pourcentage de 0 à 100, selon les valeurs que l'on obtient, donc qui varient entre 0 et 1023, si on obtient 512 en valeur analogique, on aura donc 50% d'humidité.

Ensuite, pour la partie, affichage de la trame. Si on part du principe que l'on a 24°C et une humidité de 30%, on va alors l'afficher sous cette forme :

```
Température : 24 °C --- Humidité : 30 %
```

Je configure la trame à l'aide de ce code dans la fonction `emission()` :

```
// Affichage de la trame
delay(1000);
Serial.print("Température : ");
Serial.print(temp_reel);
Serial.print(" °C");
Serial.print(" --- Humidité : ");
Serial.print(humidity_pourcentage);
Serial.println(" %");
```

J'utilise ici un timer d'une seconde avec la fonction `delay`, donc 1000 millisecondes pour 1 seconde.

Ensuite la fonction `Serial.print()` sert à afficher dans le moniteur série le texte que l'on veut, donc je configure la trame de sorte à ce qu'elle soit sous cette forme :

Température : temp\_reel °C --- Humidité : humidity\_pourcentage %

On retourne donc les valeurs `temp_reel` et `humidity_pourcentage`.

On a donc d'un point de vue global ce code :

```
#define temperaturePin 4    // Définition de temperaturePin au pin
#define humidityPin A0      // Définition de humidityPin au pin A0

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    emission();    // Initialisation de la fonction emission
}
```

```

void emission()
{
    // Variables pour convertir la température
    float temp_ana, temp_num;
    int temp_reel;

    temp_num = analogRead(temperaturePin);    // récupère la température numérique
    temp_ana = (5 * temp_num) / 1023;        // Conversion numérique/analogique
    temp_reel = (temp_ana - 0.5) * 100;       // Conversion de la valeur analogique en °C

    // Variables pour convertir d'humidité
    float humidity_ana;
    int humidity_pourcentage;

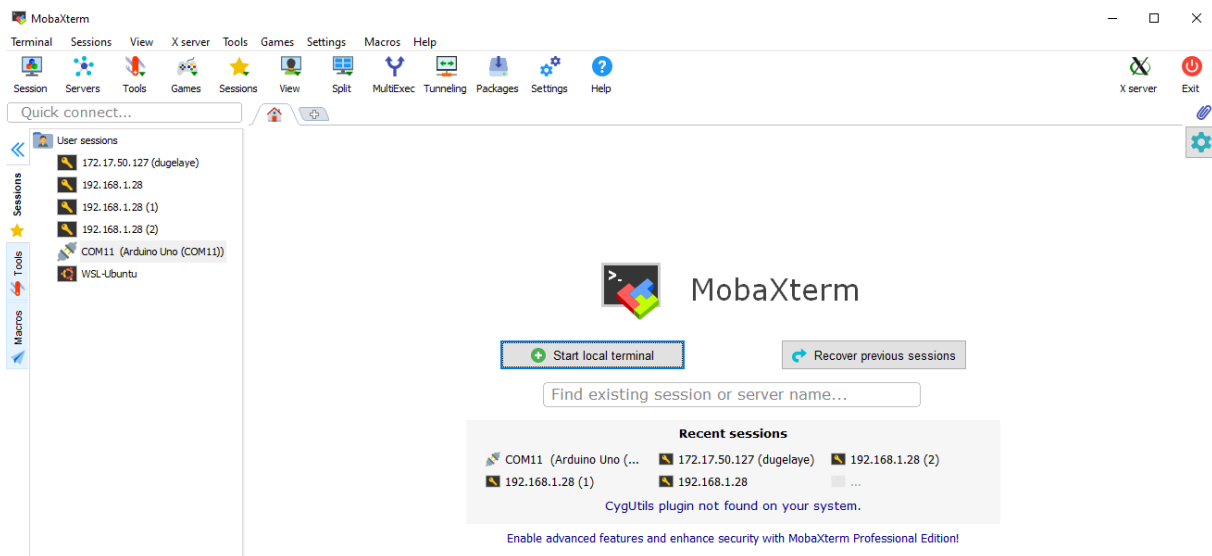
    humidity_ana = analogRead(humidityPin);   // Récupère l'humidité en valeur analogique
    // Conversion de l'humidité analogique en pourcentage
    humidity_pourcentage = map(humidity_ana, 0, 1023, 0, 100);

    // Affichage de la trame
    delay(1000);    // Délai de 1 seconde entre chaque affichage de la trame
    Serial.print("Température : ");
    Serial.print(temp_reel);
    Serial.print(" °C");
    Serial.print(" --- Humidité : ");
    Serial.print(humidity_pourcentage);
    Serial.println(" %");
}

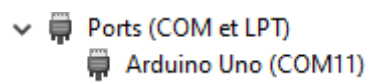
```

#### 2.4.1 Visualisation de la trame sous MobaXterm

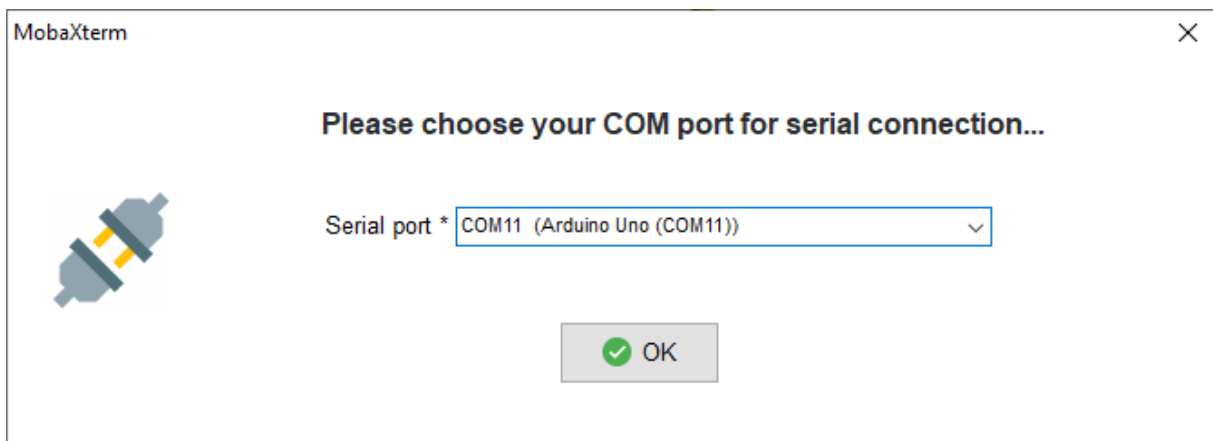
Ensuite, mon moniteur série sur Arduino ne fonctionnait pas malgré la réinstallation du logiciel arduino donc je me suis aidé du logiciel MobaXterm qui est un logiciel qui va nous servir à afficher la trame selon le port que l'on choisi.



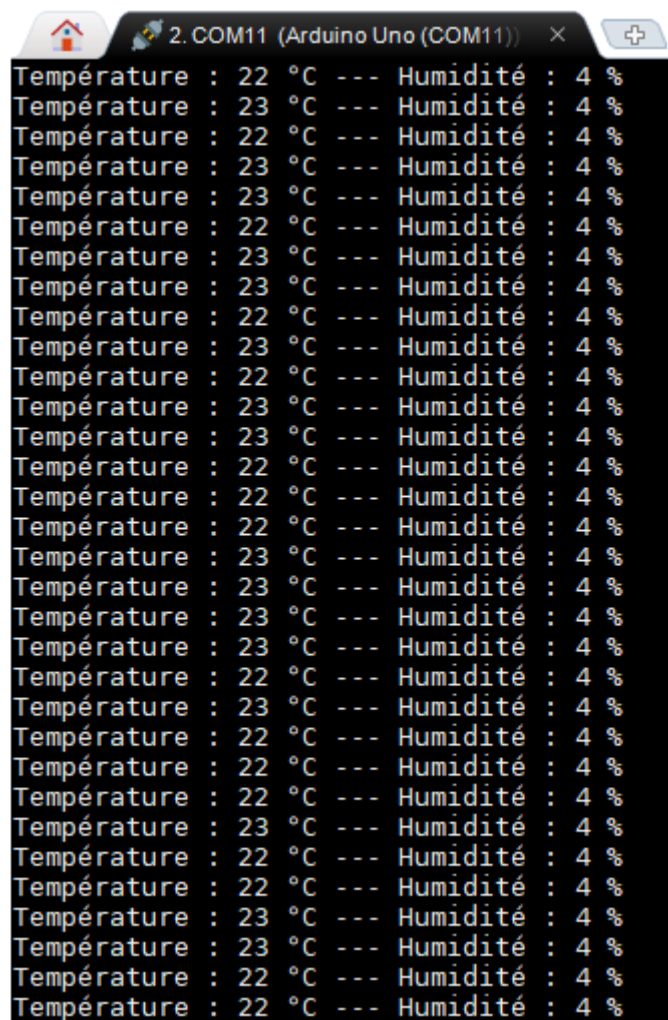
Ici la carte Arduino est connectée sur le port COM11 comme on le trouve dans le gestionnaire de périphériques.



Sur MobaXterm, je me connecte donc au port COM11



Puis on relève donc la trame sous cette forme :



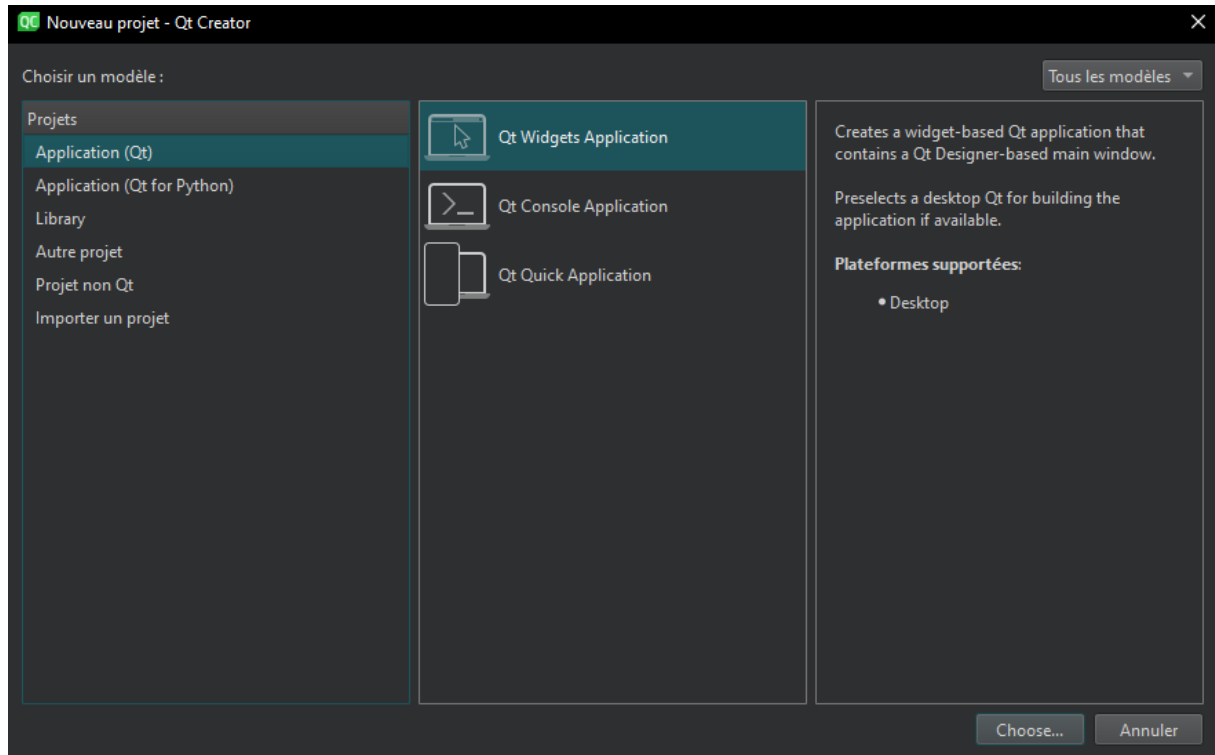
La température ici est de 22 degrés Celsius et l'humidité est de 4%. L'humidité de 4% paraît bas mais c'est logique puisque le capteur utilisé est un capteur de sol et non de l'air, il est donc normal d'avoir une valeur qui n'est pas précise et adaptée à l'air.

## 2.5 Analyse de la trame sous QT Creator

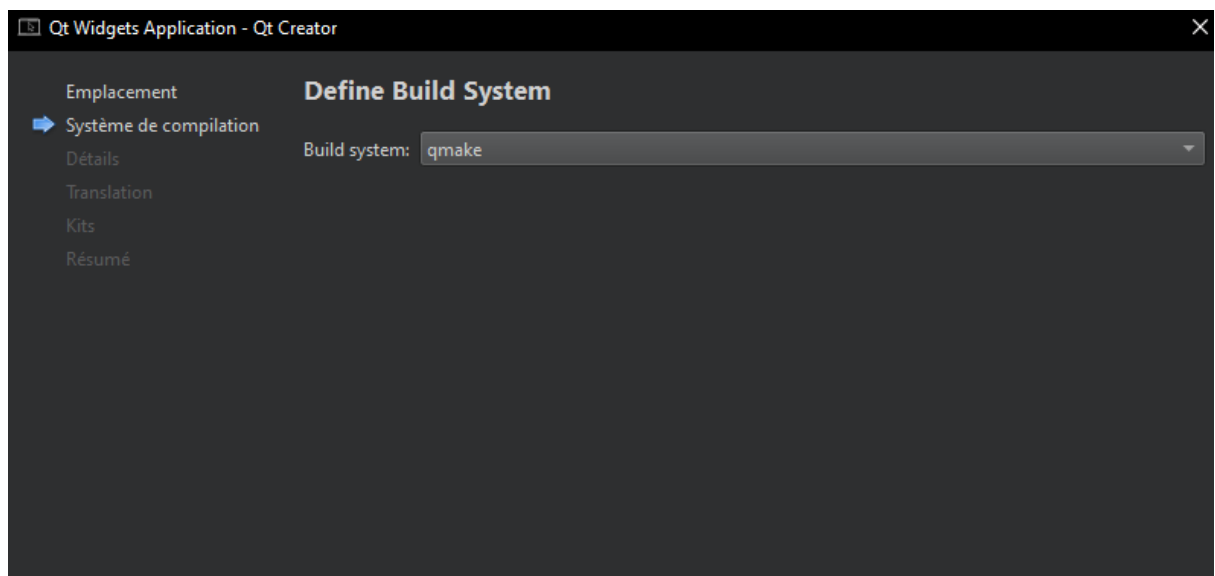
### 2.5.1 Création du projet

Maintenant que notre trame a été configurée, il faut pouvoir la récupérer depuis le logiciel QT Creator afin de récupérer seulement les valeurs qu'il nous intéresse pour les envoyer à la base de données.

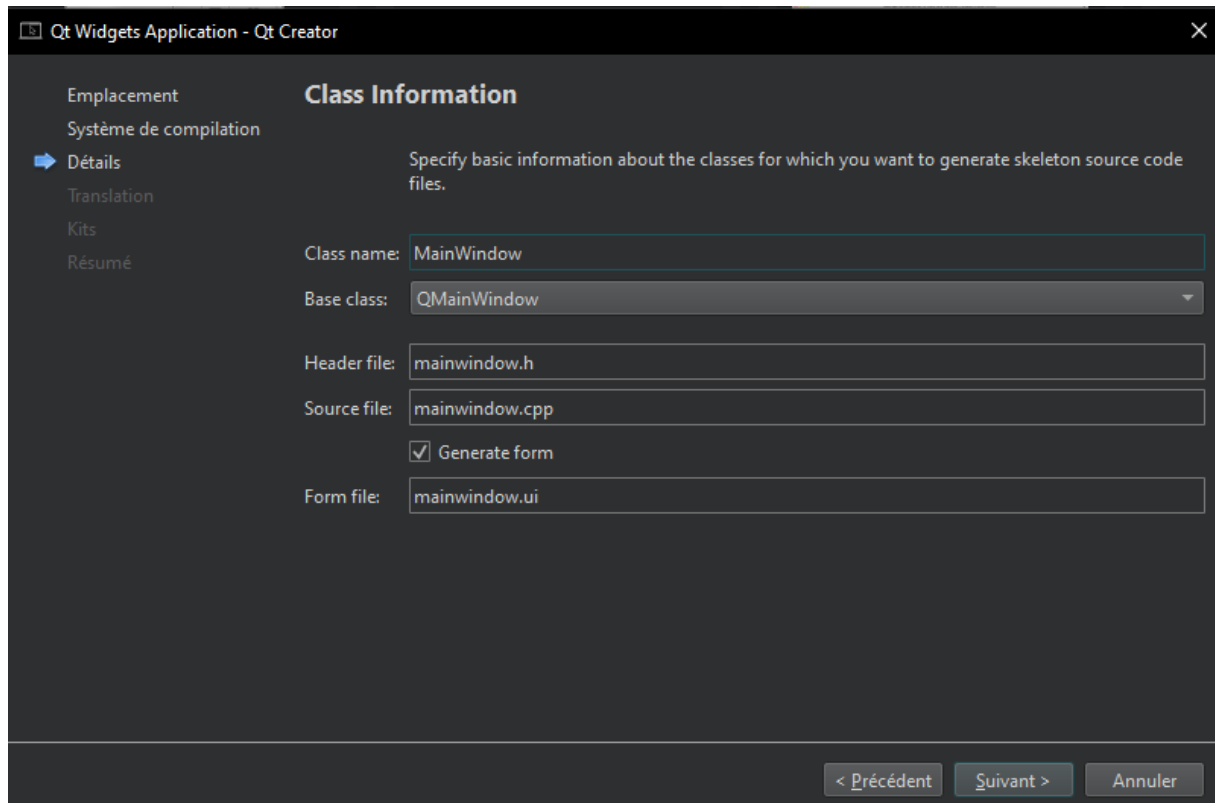
Tout d'abord, il faut créer le projet sous Qt Creator. Je crée donc une nouvelle application Qt sous Widgets.



Je choisis ensuite le gestionnaire de type qmake, qui est un outil de gestion de projet. Il est utilisé pour générer des fichiers de construction pour les projets Qt, principalement le fichier .pro qui contient les instructions pour la construction du projet.



Pour le nom des fichiers, j'ai choisi en classe de base QMainWindow et j'ajoute la classe MainWindow à mon projet, les fichiers prendront donc le nom de mainwindow.

The image shows the 'Class Information' dialog in Qt Creator. On the left is a sidebar with tabs: 'Emplacement', 'Système de compilation', 'Détails' (selected), 'Translation', 'Kits', and 'Résumé'. The main area is titled 'Class Information' and contains the text 'Specify basic information about the classes for which you want to generate skeleton source code files.' Below this are several input fields: 'Class name:' with 'MainWindow', 'Base class:' with a dropdown menu showing 'QMainWindow', 'Header file:' with 'mainwindow.h', 'Source file:' with 'mainwindow.cpp', and 'Form file:' with 'mainwindow.ui'. There is a checked checkbox labeled 'Generate form'. At the bottom right are three buttons: '< Précédent', 'Suivant >', and 'Annuler'.

Maintenant que le projet est créé, il faut d'abord supprimer le code par défaut qui relie le fichier source et le fichier d'en-tête au fichier .ui où il y a la partie graphique. Je le supprime car créer la partie graphique en code plutôt que dans le fichier .ui directement permet un meilleur contrôle sur la partie graphique et aussi que l'on crée des pointeurs en code ce qui permet une allocation de la mémoire.

Dans le fichier mainwindow.h, je supprime cette ligne qui fait référence à un pointeur vers l'interface graphique.

```
private:
    Ui::MainWindow *ui;
```

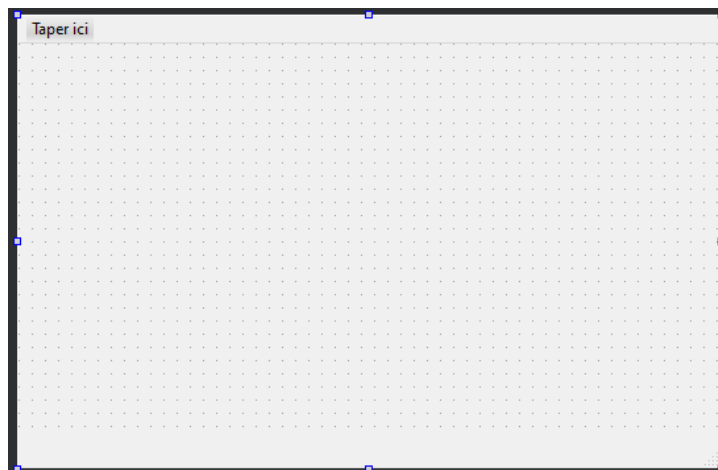
Je peux aussi supprimer dans le constructeur les parties où ui est utilisé.

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}
```

Pour ne garder que :

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
}
```

Je ne vais donc pas utiliser le fichier .ui qui va rester sous cette forme :



### 2.5.2 Réalisation et développement du code

Pour les #includes, ils sont déclarés dans le fichier/mainwindow.h où se trouve la classe MainWindow et aussi dans le fichier/mainwindow.cpp :

```
< > [icon]/mainwindow.h [icon] [icon] # :: Ui [icon] Windows (CRLF) [icon] [icon] Ligne 18, colo
5  #include <QMainWindow>           // Fenêtre principale de l'application
6  #include <QMessageBox>          // Pour afficher des fenêtres de dialogue
7  #include <QtSql>                 // Permet d'utiliser les classes SQL
8  #include <QtSql/QtSqlDatabase>   // Permet de se connecter à la base de données
9  #include <QtSql/QtSqlQuery>      // Permet les requêtes SQL
10 #include <QtSql/QtSqlError>      // Permet de signaler une erreur de la base de données
11 #include <QtSerialPort/QtSerialPort> // Pour se connecter au port série
12 #include <QtSerialPort/QtSerialPortInfo> // Obtenir des infos sur les ports série
13 #include <QtWidgets>             // Widgets et fonctionnalités graphiques
```

```
> [icon] [icon]/mainwindow.cpp* [icon] [icon] # [icon] MainWindow::M... [icon] Windows (CRLF)
1  #include "mainwindow.h"
2  #include <QtWidgets>           // Widgets et fonctionnalités graphiques
3  #include <QApplication>        // Fournit une classe d'application Qt
```

Ils sont tous utiles pour diverses raisons tel que l'utilisation du SQL, du port série ou encore de la base de données.

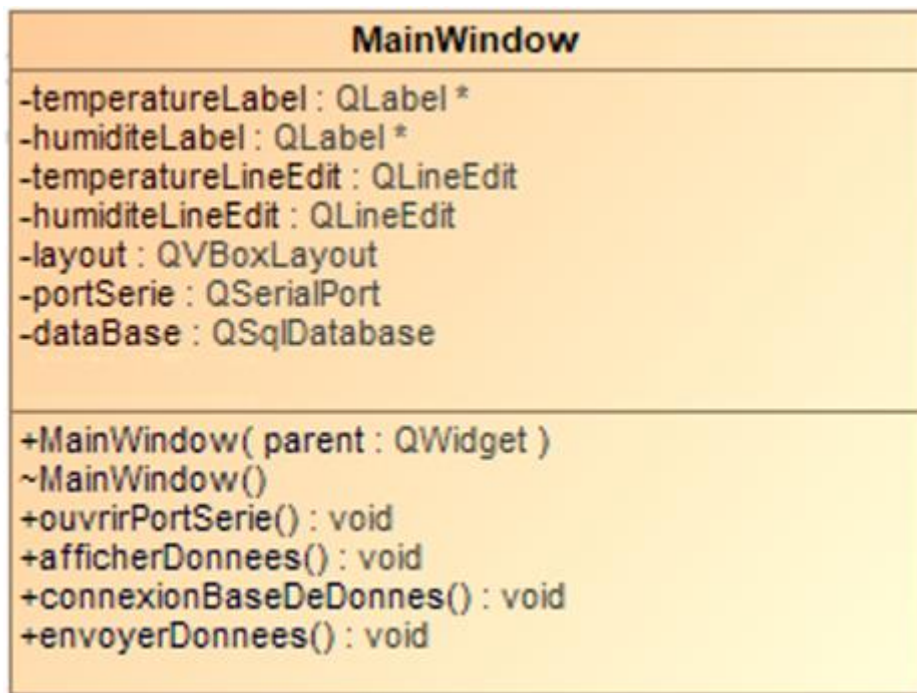
De plus, j'aurais besoin de ces lignes dans le fichier .pro.

```
> [icon] [icon]botaki.pro*
1  QT += core gui
2  QT += serialport
3  QT += sql
```

- Core gui : Ajoute les modules de base pour créer l'IHM
- Serialport : Ajoute le module Serialport pour communiquer
- SQL : Ajoute le module SQL pour utiliser les commandes SQL



La classe MainWindow peut être interprétée dans un diagramme de classe :



Où l'on retrouve les différentes fonctions que l'on va utiliser ainsi que les éléments qui vont nous servir dans notre interface graphique.

Pour la classe MainWindow, on retrouve ce code :

```
23 // Déclaration de la classe MainWindow qui hérite de QMainWindow
24 class MainWindow : public QMainWindow
25 {
26     Q_OBJECT
27
28     public:
29         MainWindow(QWidget *parent = nullptr); // Destructeur
30         ~MainWindow(); // Destructeur
31
32     private:
33         QLabel *temperatureLabel;
34         QLabel *humiditeLabel;
35         QLineEdit *temperatureLineEdit;
36         QLineEdit *humiditeLineEdit;
37
38         QVBoxLayout *layout;
39         QSerialPort portSerie;
40         QSqlDatabase dataBase;
41         QTimer *timer;
42
43     private slots:
44         void ouvrirPortSerie();
45         void afficherDonnees();
46         void connexionBaseDeDonnees();
47         void envoyerDonnees();
48 };
49
50 #endif // MAINWINDOW_H
```

La classe MainWindow peut donc être vue de deux façons différentes.

Ensuite, l'interface graphique a donc été faite en code, qui a été ajouté dans le constructeur. Ici je crée les widgets que l'on va nommer et afficher dans la fenêtre puis on leur fixe une hauteur.

```
// Création des Label et des LineEdit à afficher dans la fenêtre
temperatureLabel = new QLabel("Température :");
humiditeLabel = new QLabel("Humidité :");
temperatureLineEdit = new QLineEdit;
humiditeLineEdit = new QLineEdit;
```

Ensuite, je crée les boutons dans l'interface graphique, qui seront nommés et qui renverront vers une fonction selon le bouton. Par exemple le bouton boutonAfficher renvoie à la fonction afficherDonnees, le bouton boutonConnexion renvoie à la fonction connexionBaseDeDonnees et le bouton boutonEnvoyer renvoie à la fonction envoyerDonnees.

```
// Création des boutons pour afficher les mesures, se connecter à la base de données et enregistrer les mesures
QPushButton *boutonAfficher = new QPushButton("Afficher les mesures", this);
connect(boutonAfficher, &QPushButton::clicked, this, &MainWindow::afficherDonnees);

QPushButton *boutonConnexion = new QPushButton("Connexion base de données", this);
connect(boutonConnexion, &QPushButton::clicked, this, &MainWindow::connexionBaseDeDonnees);

QPushButton *boutonEnvoyer = new QPushButton("Enregistrer les mesures", this);
connect(boutonEnvoyer, &QPushButton::clicked, this, &MainWindow::envoyerDonnees);
```

Pour continuer, je dois créer un agencement vertical avec un layout qui servira de structure pour les éléments de l'IHM.

```
QVBoxLayout *layout = new QVBoxLayout;
layout->addWidget(temperatureLabel);
layout->addWidget(temperatureLineEdit);
layout->addWidget(humiditeLabel);
layout->addWidget(humiditeLineEdit);
layout->addWidget(boutonAfficher);
layout->addWidget(boutonConnexion);
layout->addWidget(boutonEnvoyer);

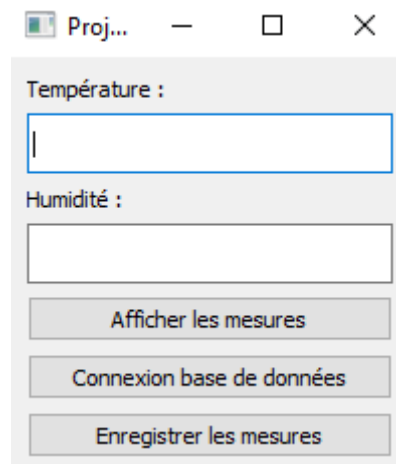
QWidget *centralWidget = new QWidget;
centralWidget->setLayout(layout);

setCentralWidget(centralWidget);

ouvrirPortSerie();
```

- QVBoxLayout \*layout : Crée un nouvel objet QVBoxLayout qui est l'agencement vertical des widgets
- layout->add : ajoutent les widgets créés plus haut
- centralWidget->setLayout(layout) : définit le layout vertical créé précédemment pour afficher dans le widget central les widgets ajoutés précédemment au layout
- setCentralWidget(centralWidget) : définit le widget central comme widget principal, pour afficher le contenu du widget principal dans la fenêtre
- ouvrirPortSerie() : exécute la fonction pour ouvrir le port série automatiquement

Ces éléments ajoutés à l'interface graphique nous donnent donc ce résultat :



Le but étant d'obtenir une interface graphique simple, compréhensible et efficace, il n'y a donc pas besoin d'avoir trop d'éléments dans l'IHM afin de bien la comprendre, il suffit d'y avoir les éléments importants.

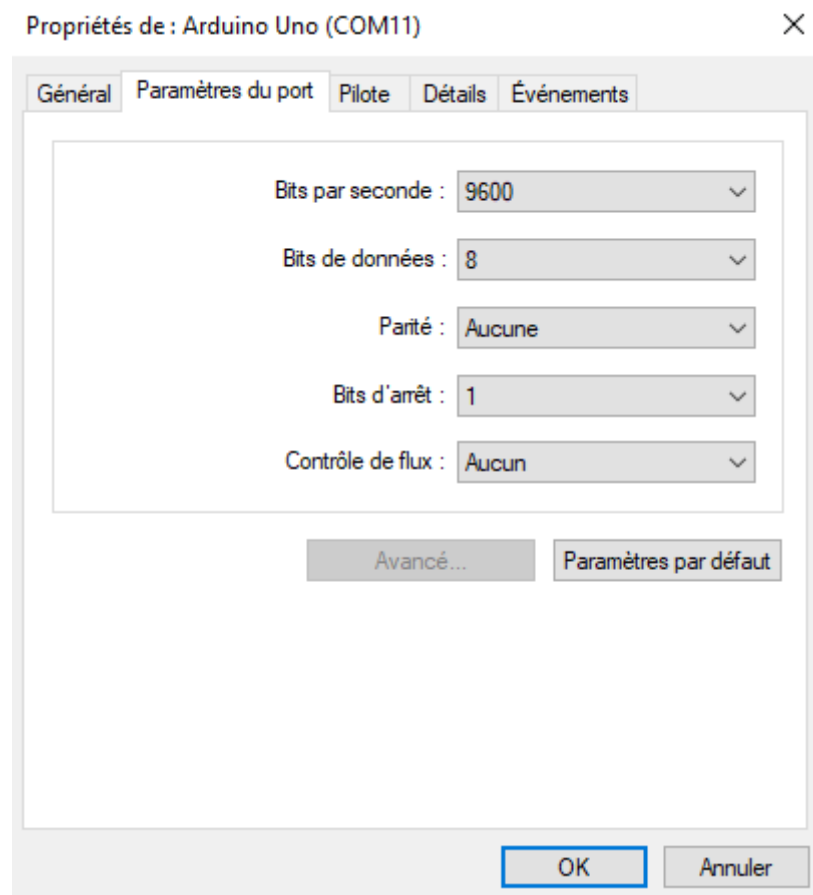
Ce qui donne donc pour le constructeur ce code avec la documentation doxygen, qui indique à quoi sert la fonction et indique les valeurs passées en paramètres :

```
5  /**
6   * @brief Constructeur de la classe MainWindow
7   * @param parent Le widget parent
8   */
9  MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
10 {
11     // Création des Label et des LineEdit à afficher dans la fenêtre
12     temperatureLabel = new QLabel("Température :");
13     humiditeLabel = new QLabel("Humidité :");
14     temperatureLineEdit = new QLineEdit;
15     humiditeLineEdit = new QLineEdit;
16
17     // Configuration de la hauteur des zones de texte
18     temperatureLineEdit->setFixedHeight(30);
19     humiditeLineEdit->setFixedHeight(30);
20
21     // Création des boutons pour afficher les mesures
22     QPushButton *boutonAfficher = new QPushButton("Afficher les mesures", this);
23     connect(boutonAfficher, &QPushButton::clicked, this, &MainWindow::afficherDonnees);
24
25     QPushButton *boutonConnexion = new QPushButton("Connexion base de données", this);
26     connect(boutonConnexion, &QPushButton::clicked, this, &MainWindow::connexionBaseDeDonnees);
27
28     QPushButton *boutonEnvoyer = new QPushButton("Enregistrer les mesures", this);
29     connect(boutonEnvoyer, &QPushButton::clicked, this, &MainWindow::envoyerDonnees);
30
31     // Ajout des widgets dans le layout
32     QVBoxLayout *layout = new QVBoxLayout; // Création d'un layout vertical pour organiser les widgets
33     layout->addWidget(temperatureLabel);
34     layout->addWidget(temperatureLineEdit);
35     layout->addWidget(humiditeLabel);
36     layout->addWidget(humiditeLineEdit);
37     layout->addWidget(boutonAfficher);
38     layout->addWidget(boutonConnexion);
39     layout->addWidget(boutonEnvoyer);
40
41     // Création d'un widget central pour contenir le layout
42     QWidget *centralWidget = new QWidget;
43     centralWidget->setLayout(layout);
44     setCentralWidget(centralWidget); // Définition du widget central de la fenêtre principale
45     ouvrirPortSerie(); // Fonction pour ouvrir le port série automatiquement
46 }
```

Ensuite, on configure les paramètres du port série :

```
// Configuration du port série
portSerie.setPortName("COM11");
portSerie.setBaudRate(QSerialPort::Baud9600);
portSerie.setDataBits(QSerialPort::Data8);
portSerie.setParity(QSerialPort::NoParity);
portSerie.setStopBits(QSerialPort::OneStop);
portSerie.setFlowControl(QSerialPort::NoFlowControl);
```

Ces paramètres correspondent aux paramètres du port COM11 qu'on retrouve en allant dans les propriétés du port dans le gestionnaire de périphériques.

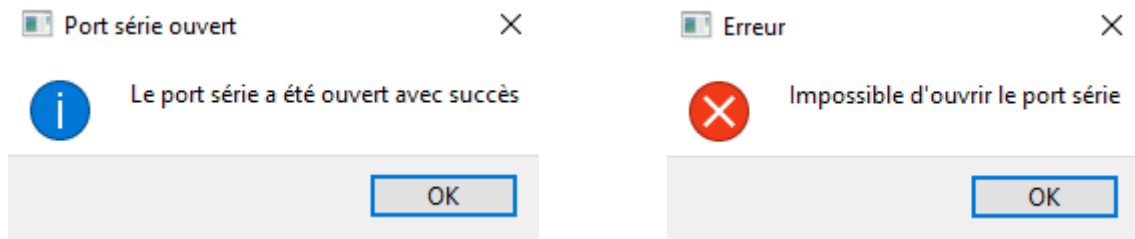


J'ai ensuite ajouté l'information qui va être retournée à l'utilisateur si le port série est ouvert ou non.

```
// Vérification de l'ouverture du port série
if (!portSerie.open(QIODevice::ReadWrite))
{
    // Message d'erreur
    QMessageBox::critical(this, tr("Erreur"), tr("Impossible d'ouvrir le port série"));
    return;
}

// Message de confirmation
QMessageBox::information(this, tr("Port série ouvert"), tr("Le port série a été ouvert avec succès"));
```

Il nous renvoie donc ces deux messages :



Les valeurs de la température et l'humidité dans la trame sont analysées dans la fonction `afficherDonnees()` :

```
void MainWindow::afficherDonnees()
{
    QByteArray donnees = portSerie.readAll(); // Lit toutes les données disponibles du port série
    QString dataString(donnees); // Convertit les données lues en chaîne de caractères
    QString line, str_temperature, str_humidity;

    // Boucle tant que la chaîne de données n'est pas vide
    while (!dataString.isEmpty())
    {
        line = dataString.section('\n', 0, 0); // Récupère la première ligne de la chaîne de données
        dataString = dataString.section('\n', 1); // Supprime la première ligne de la chaîne de données

        // On vérifie si la ligne commence par "Température"
        if (line.startsWith("Température")) // refaire
        {
            QStringList field = line.split(" "); // On définit l'espace comme délimiteur
            str_temperature = field[2]; //
            str_humidity = field[7];
            break; // Sort de la boucle
        }
    }
    // Affichage des valeurs
    temperatureLineEdit->setText(str_temperature);
    humiditeLineEdit->setText(str_humidity);
}
```

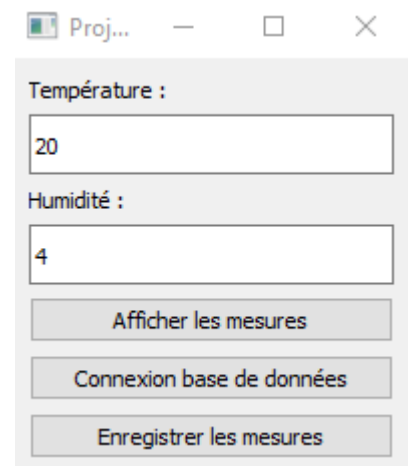
On lit les données du port série grâce à la fonction `portSerie.readAll()` que je mets dans un tableau `QByteArray` que je convertis ensuite en chaîne de caractère.

Ensuite, la boucle `while` tourne tant que la chaîne de données n'est pas vide, on récupère chaque ligne de la trame avec `line = dataString.section('\n', 0, 0)` ; puis on supprime la première ligne ensuite afin de prendre chaque ligne de la trame.

La boucle `if` est définie si la trame commence par `Température`, donc oui, puis je définis l'espace comme délimiteur avec la ligne `field = line.split(" ")`. `str_temperature` est attribué à la valeur de la deuxième partie de la chaîne « `line` » donc au deuxième espace et au septième espace pour `str_humidity`.

Si on regarde la trame que l'on envoie : `Température : x °C --- Humidité : y %`, après le deuxième espace il y a bien la valeur de `x`, la température, et après le septième espace il y a bien la valeur `y`, l'humidité.

Enfin, on envoie ces valeurs dans les zones de textes `LineEdit` de l'interface graphique ce qui permet à l'utilisateur de pouvoir voir ses mesures.



Après avoir récupérer les valeurs, il faut d'abord se connecter à la base de données avant de pouvoir les envoyer avec la fonction connexionBaseDeDonnees().

Cette fonction permet de se créer l'objet de connexion DataBase, que je vais me servir pour me connecter à la base de données qui est définie par les informations que je lui donne, son adresse IP, son port, ses identifiants et son nom.

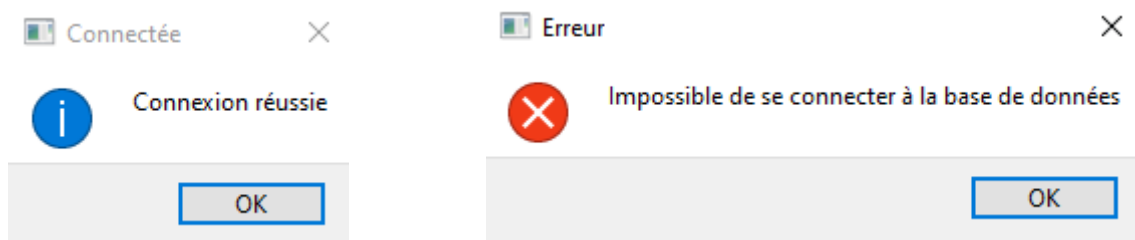
La fonction renvoie aussi un message confirmant la connexion ou un message d'erreur si on ne s'est pas connecté.

```
/**
 * @brief Établit une connexion à la base de données
 * @note Affiche un message d'erreur en cas d'échec de la connexion à la base de données
 */
void MainWindow::connexionBaseDeDonnees()
{
    // Créé l'objet de connexion
    DataBase = QSqlDatabase::addDatabase("QMYSQL");

    // Initialisation des informations de connexion
    DataBase.setHostName("172.17.50.114"); // Nom de l'hôte ou adresse IP de l'ordinateur qui héberge
    DataBase.setPort(3306); // Port sur lequel le serveur MySQL écoute les connexions
    DataBase.setUserName("root69"); // Nom de l'utilisateur
    DataBase.setPassword("root"); // Mot de passe
    DataBase.setDatabaseName("botaki"); // Nom de la base de données

    // Vérifie si on est bien connecté à la base de données
    if(DataBase.open())
    {
        // Message qui confirme la connexion
        QMessageBox::information(this, tr("Connectée"), tr("Connexion réussie"));
    }
    else
    {
        // Message d'erreur si la connexion a échoué
        QMessageBox::critical(this, tr("Erreur"), tr("Impossible de se connecter à la base de données"));
    }
}
```

Le programme nous renvoie donc ces messages :



Il faut pour finir envoyer les mesures à la base de données. Je récupère les valeurs des QLineEdit dans des QString, ensuite je vérifie que les QString ne sont pas nul.

Ensuite avec des requêtes SQL j'insère avec INSERT INTO à la base de données les valeurs que j'ai récupéré. Puis on envoie un message qui confirme si les valeurs ont bien été envoyées ou un message d'erreur si ce n'est pas le cas.

```
/**
 * @brief Enregistre les données dans la base de données.
 */
void MainWindow::envoyerDonnees()
{
    // Récupérer les valeurs depuis les QLineEdit
    QString temperature = temperatureLineEdit->text();
    QString humidity = humiditeLineEdit->text();

    // Vérifie que les valeurs ne sont pas vides
    if(!temperature.isEmpty() && !humidity.isEmpty())
    {
        // Prépare la requête d'insertion
        QSqlQuery query;

        // On insert dans la table "mesuressol" de la base de données "botaki" les valeurs
        query.prepare("INSERT INTO mesuressol (temperature, humidite) VALUES (:temperature, :humidite)");

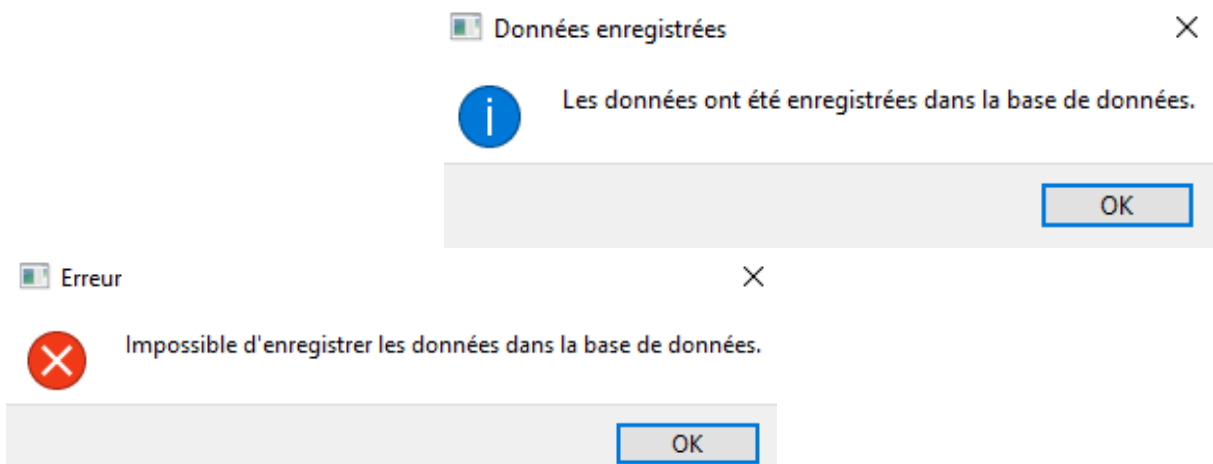
        // Associe les valeurs aux paramètres de la requête
        query.bindValue(":temperature", temperature.toInt());
        query.bindValue(":humidite", humidity.toInt());

        // On vérifie si on interagit bien avec la base de données
        if(query.exec())
        {
            // Message de confirmation
            QMessageBox::information(this, tr("Données enregistrées"),
                                   tr("Les données ont été enregistrées dans la base de données."));
        }
        else
        {
            // Message d'erreur
            QMessageBox::critical(this, tr("Erreur"),
                                 tr("Impossible d'enregistrer les données dans la base de données."));
        }
    }
}
```

Pour bien vérifier que je ne me trompe pas dans l'envoi des valeurs, il faut regarder le type que j'envoie et le type que la base de données reçoit. On envoie des valeurs en int dans le code et dans la base de données on reçoit aussi des valeurs en int :

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra	Action
<input type="checkbox"/> 1	temperature	int(11)			Non	Aucun(e)			Modifier  Supprimer  Plus
<input type="checkbox"/> 2	humidite	int(11)			Non	Aucun(e)			Modifier  Supprimer  Plus

Le code peut donc nous afficher ces interfaces :







Après ce code j'ai eu l'erreur suivante :

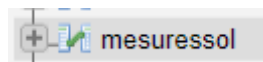
```
Sortie de l'application | [icônes]
botaki
QSqlDatabase: QMYSQL driver not loaded
QSqlDatabase: available drivers: QSQLITE QMYSQL QMYSQL3 QODBC QODBC3 QPSQL QPSQL7
13:54:32: E:/Projet/QT/build-botaki-Desktop_Qt_5.12.2_MSVC2017_64bit2-Debug/debug/ProjetBotaki.exe exited with code 0
```

Elle signifie que les drivers pour utiliser le SQL sur Qt ne sont pas installés. J'ai trouvé ensuite sur internet ce qu'il fallait installer, c'était les fichiers libmysql.dll et libmysqld.dll à mettre dans le chemin suivant : Ce PC>OS(C:)>Qt>5.12.2>msvc2017\_64>bin

	libmysql.dll	23/05/2017 14:20	Extension de l'app...	4 765 Ko
	libmysqld.dll	08/11/2018 09:41	Extension de l'app...	13 131 Ko

Après avoir mis ces deux fichiers je n'avais plus l'erreur et j'ai pu envoyer mes valeurs à la base de données.

Je me connecte ensuite à la base de données depuis mon PC afin de vérifier si j'ai bien mes mesures enregistrées, il y a la table mesuressol où je rentre donc mes valeurs.



Et ensuite on remarque qu'il y a bien nos mesures dans la base de données.

Proj...

Température :

Humidité :

Afficher les mesures

Connexion base de données

Enregistrer les mesures

+ Options

temperature	humidite
23	4
23	42
22	4
22	4
22	4
23	4
23	4
25	4
20	4

## 2.6 Fiche de test et fiche de suivi

Afin de visualiser les tests que j'ai fait et afin de voir s'ils sont cohérent ou non, j'ai fait un cahier de recette qui comprend tous les tests que j'ai fait pour voir si mon projet est fonctionnel ou non, et s'il manque une partie que je n'ai pas fait.

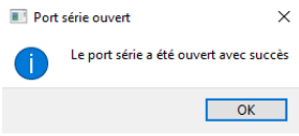
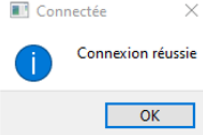
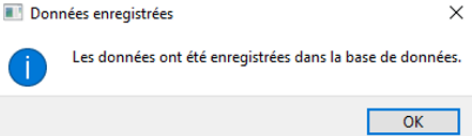
### 2.6.1 Fiche de test - Arduino

Pour la fiche de test de la partie Arduino, les tests descriptifs sont si la carte Arduino et les capteurs sont bien câblés, ensuite si j'arrive à mesurer la température et l'humidité du sol, puis si j'ai mis en forme les mesures et enfin si j'arrive à envoyer la trame. J'ai donc tous ces tests en état bon.

id	Description du cas de test	résultat	Etat
1	décrire le test à réaliser. La description doit permettre à un utilisateur de réaliser l'action à mener	Présenter le résultat attendu	
2	Brancher et câbler la carte Arduino selon le schéma établi	La carte Arduino fonctionne correctement.	Ok
3	Brancher et câbler les différents capteurs selon le schéma établi	Les capteurs fonctionnent correctement.	Ok
4	Mesurer l'humidité et la température du sol	Les mesures sont bien recueillies	Ok
5	Mettre en forme les mesures issues des capteurs	Les mesures sont bien affichées et sur la plage de valeur attendue.	Ok
6	Envoie de la trame	On arrive à bien capter la trame depuis MobaXterm.	Ok

### 2.6.2 Fiche de test – Qt Creator

Pour la fiche de test de la partie Qt Creator, les tests descriptifs sont si j'arrive à me connecter à la carte Arduino avec le port série, ensuite si j'arrive bien à récupérer la trame et les mesures des capteurs, puis si j'arrive bien à me connecter à la base de données et enfin si j'arrive bien à envoyer les mesures à la base de données. J'ai donc tous ces tests en état bon.

7	Connexion à la carte Arduino depuis Qt Creator à son port de communication	Affichage du message "Le port série a été ouvert avec succès". 	Ok
8	Vérifier que la trame comporte bien les mesures issues des capteurs	La trame comporte les mesures des capteurs.	Ok
9	Connecter Qt Creator à la base de données	Affichage du message "Connexion réussie". 	Ok
10	Envoyer les mesures à la base de données	Affichage du message "Les données ont été enregistrées dans la base de données". 	Ok

### 2.6.3 Fiche de suivi

Semaine 1 à semaine 2	Prise de connaissance du sujet et analyse du cahier des charges
Semaine 3	Mise en place de l'environnement Azure Devops pour faciliter la gestion de projet
Semaine 4	Préparation de la revue 0
Semaine 5	Revue 0 et découverte du matériel
Semaine 6 et semaine 7	Vacances : Rédaction du rapport Préparation de la revue 1
Semaine 8	Préparation de la revue 1
Semaine 9	Revue 1 Établissement du schéma de la carte Arduino et des capteurs Début de programmation sous Arduino
Semaine 10 à semaine 13	Programmation sous Arduino
Semaine 14	Commencement de programmation sous Qt Creator
Semaine 15 et semaine 16	Vacances Rédaction du rapport Préparation de la revue 2
Semaine 17	Préparation de la revue 2 Programmation sous Qt Creator – Connexion au port série et début d'analyse de trame
Semaine 18	Revue 2 Programmation sous Qt Creator – Analyse de trame
Semaine 19	Finalisation de la programmation sous Qt Creator – Connexion à la base de données et envoi des mesures
Semaine 20 et semaine 21	Rédaction du rapport et préparation pour la revue finale

## 2.7 Conclusion

En tant qu'étudiant en BTS Informatique et Réseaux option Systèmes Numériques, j'ai saisi l'opportunité unique qu'a représenté le projet Botaki pour mettre en pratique les compétences que j'ai acquises au cours de ces deux années d'études. Ce projet m'a permis de développer mes compétences en programmation, en gestion de projet et en communication technique.

Pour la réalisation de ce projet, j'ai utilisé une carte Arduino équipée de différents capteurs, d'humidité et température afin de mesurer ces valeurs dans le sol d'une serre. J'ai configuré la trame avec des mesures avec Arduino. Ensuite, j'ai configuré le port série sur Qt Creator afin de récupérer ensuite les mesures puis je me suis connecté à la base de données pour y envoyer les mesures.

La création du programme a constitué une étape cruciale du projet. J'ai écrit du code en c++ et en c sur Arduino dans un premier temps, puis j'ai écrit du code en c++ sur Qt Creator. Tout en m'assurant que le programme était correct à chaque étape de la programmation avec des tests pour vérifier le bon fonctionnement et corriger les erreurs.

Ensuite, j'ai réalisé une documentation technique complète en détaillant tous les aspects du projet et les étapes, depuis la carte Arduino jusqu'au bon fonctionnement du programme.

De plus, j'ai créé des diagrammes SysML afin de faciliter la compréhension du système global.

En résumé, le projet Botaki a été une expérience extrêmement enrichissante pour moi, me permettant de développer mes compétences en programmation ainsi qu'en gestion de projet.