Honey Bee T

Projet BTS SN-IR
La Salle Avignon 2019

Dossier v1.0
Revue finale

Table des matières

Présentation générale du système supportant le projet

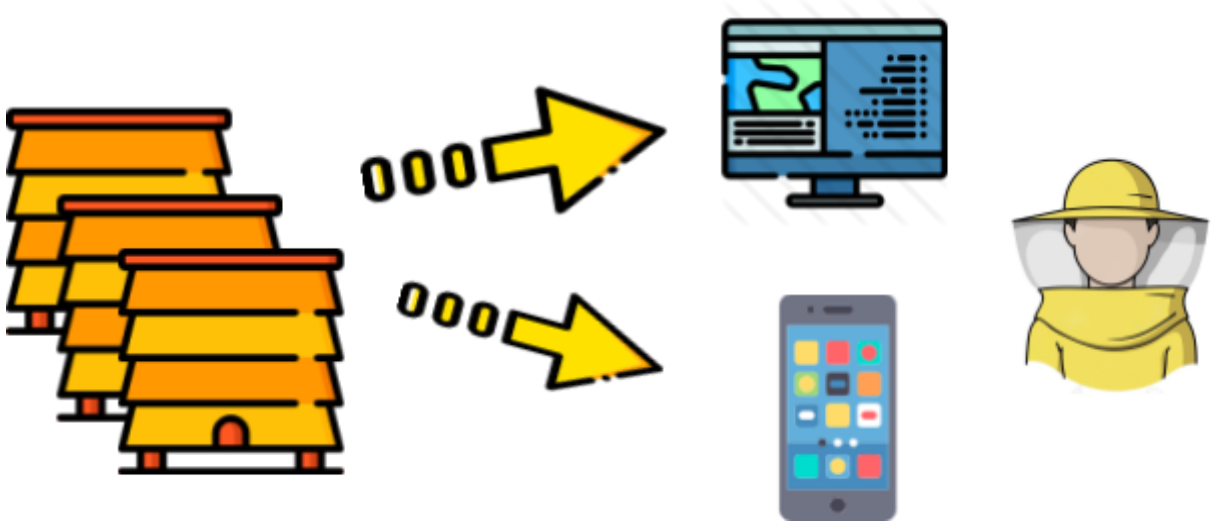
Il s'agit de réaliser un **système autonome** permettant de connaître à distance certains paramètres d'une ruche afin d'assurer son suivi et d'évaluer la santé des abeilles.

Les abeilles subissent une mortalité accrue chaque année, principalement en raison des pesticides présents dans l'environnement, auxquels elles sont particulièrement sensibles.

Une mortalité aiguë et anormale d'une colonie d'abeilles peut être un signe d'intoxication
aux pesticides et donc d'un environnement pollué. Évaluer la santé des abeilles c'est donc **analyser** indirectement la **qualité de l'environnement**.

Le projet consiste donc à équiper une ruche d'abeilles en y ajoutant des **capteurs** pour permettre d'obtenir différentes informations telles que la **température**, **l'humidité**, **le poids**, la **pression atmosphérique**, **l'ensoleillement**.

Cette équipement **ne doit en aucun cas gêner** l'apiculteur dans son travail et les abeilles.



Analyse de l'existant

La ruche connectée est un outil de suivi en temps réel de colonies d'abeilles. Les données
enregistrées par ces ruches équipées permettent de **générer des alertes** lors d'un changement anormale et soudain. comme par exemple si la ruche perd du
poids …

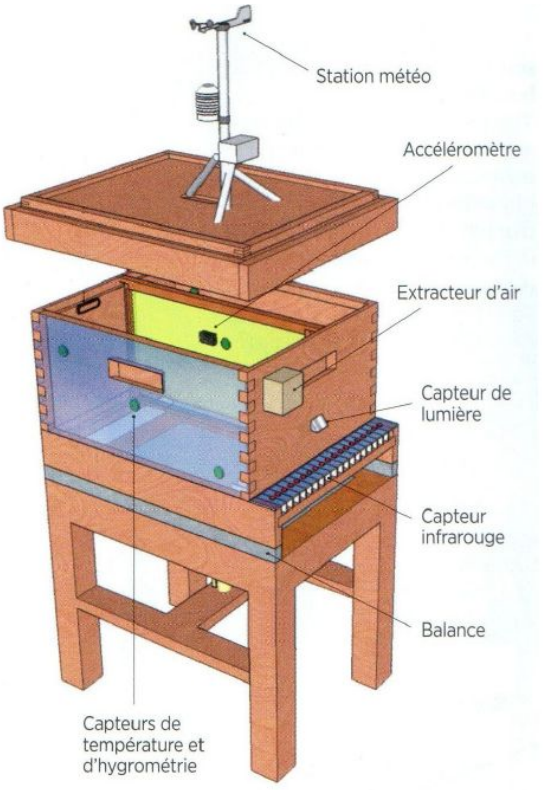
Il existe dans le commerce de nombreux modèles de « ruche connectée » :
● Le module connecté « B-Keep », associé à une application web, permet aux apiculteurs de suivre à distance le cycle de vie de leurs colonies d'abeilles. Ce module, qui s'adapte à tous types de ruche (Dadant, Langstroth, ...), mesure notamment la température et l'humidité des ruches. Lien : hostabee.com
● BeeGuard est une solution complète et modulaire pour obtenir une vision de l'activité des abeilles à distance. Lien : www.beeguard.fr
● Autres : www.icko-apiculture.com et www.label-abeille.org

L'Institut technique et scientifique de l'apiculture et de la pollinisation a pour objectif de concourir au développement de l'apiculture à travers l'expérimentation, la recherche appliquée, l'assistance technico-économique, l'animation, la diffusion et la formation. Lien :
itsap.asso.fr

Expression du besoin

La ruche connectée doit permettre à l'apiculteur d'optimiser le suivi de ses abeilles :

● **L'orientation** et **luminosité** permettent d'optimiser son rendement et d'influer sur la période de pollinisation des abeilles au cours de la
journée.
● **L'humidité** et la **température** signalent, par exemple, s'il faut donner à boire aux abeilles ou si la ruche peut être ouverte.
● La **masse** témoigne de la santé de la colonie et de l'état de la production.
● La **pression atmosphérique** et **l'humidité** préviennent d'un changement météorologique qui provoquerait un changement de comportement et un rassemblement de la colonie dans le ruche.
● La **géolocalisation** (complétée d'une alerte antivol) apportera un gain de temps dans l'organisation des tournées de récoltes et permet une intervention rapide de récolte ou de traitement.



Le système « ruche connectée » doit donc réaliser les missions suivantes :

● L'**enregistrement à intervalles réguliers** (10 min) des mesures effectuées suivantes :
  - Température intérieure et extérieure,
  - Humidité relative intérieure et extérieure,
  - Pression atmosphérique,
  - Poids de la ruche,
  - Ensoleillement,
  - Niveau de charge, tension et courant de la batterie
● Le comptage des abeilles sortantes et entrantes (en option)
● La **protection contre le vol**
● La géolocalisation (en option)
● L'**enregistrement toutes les heures des données collectées** (moyennes, min, max),
● L'affichage des grandeurs mesurées sous forme de vues **graphiques** et de **tableaux récapitulatifs journaliers**.
● L'alerte en cas de variation brutale d'une grandeur mesurée (perte de poids soudaine). L'alerte pourra être signalée sous la forme d'un email ou d'un message SMS envoyé sur le smartphone de l'apiculteur.

Contraintes :

❏ Le système **ne doit pas perturber les abeilles**. Une attention particulière doit être portée aux technologies employés, aux ondes et, aux fréquences utilisées.

❏ Le système **ne doit pas entraver le travail de l'apiculteur**. Les capteurs doivent pouvoir être déconnectées simplement.

❏ Le système doit être le plus longtemps possible autonome en énergie afin de pouvoir être installé dans un endroit isolé. (**autonomie 15 jours sans soleil**).

Le développement du système doit répondre aux exigences des utilisateurs :

- simplicité d'utilisation,
- correspondre aux contraintes définies,
- réalisable dans un délai de 200 heures (IR) et 170 heures (EC).

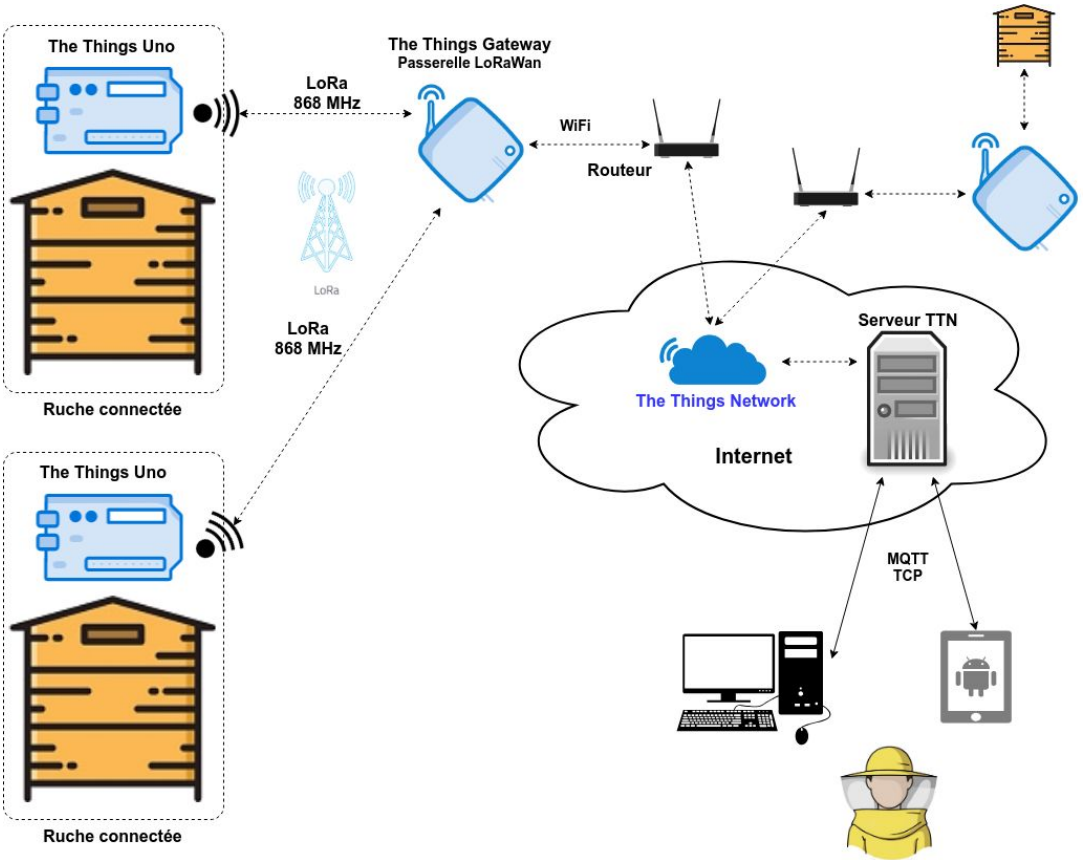
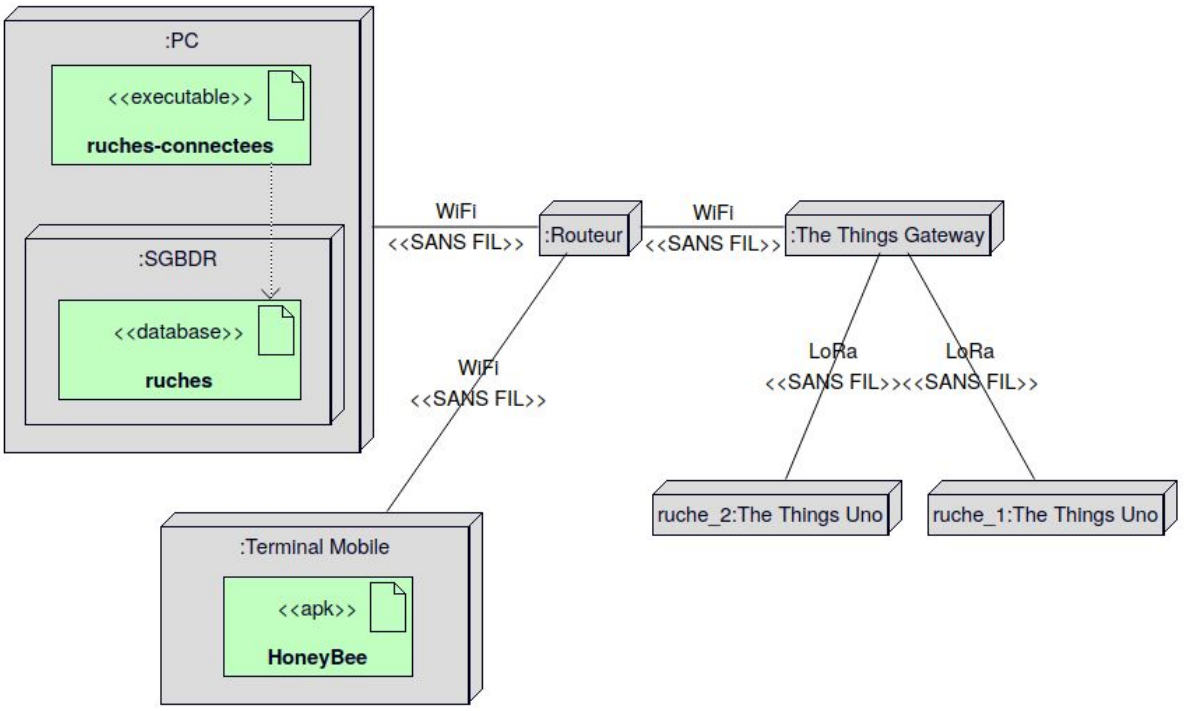


Diagramme de déploiement



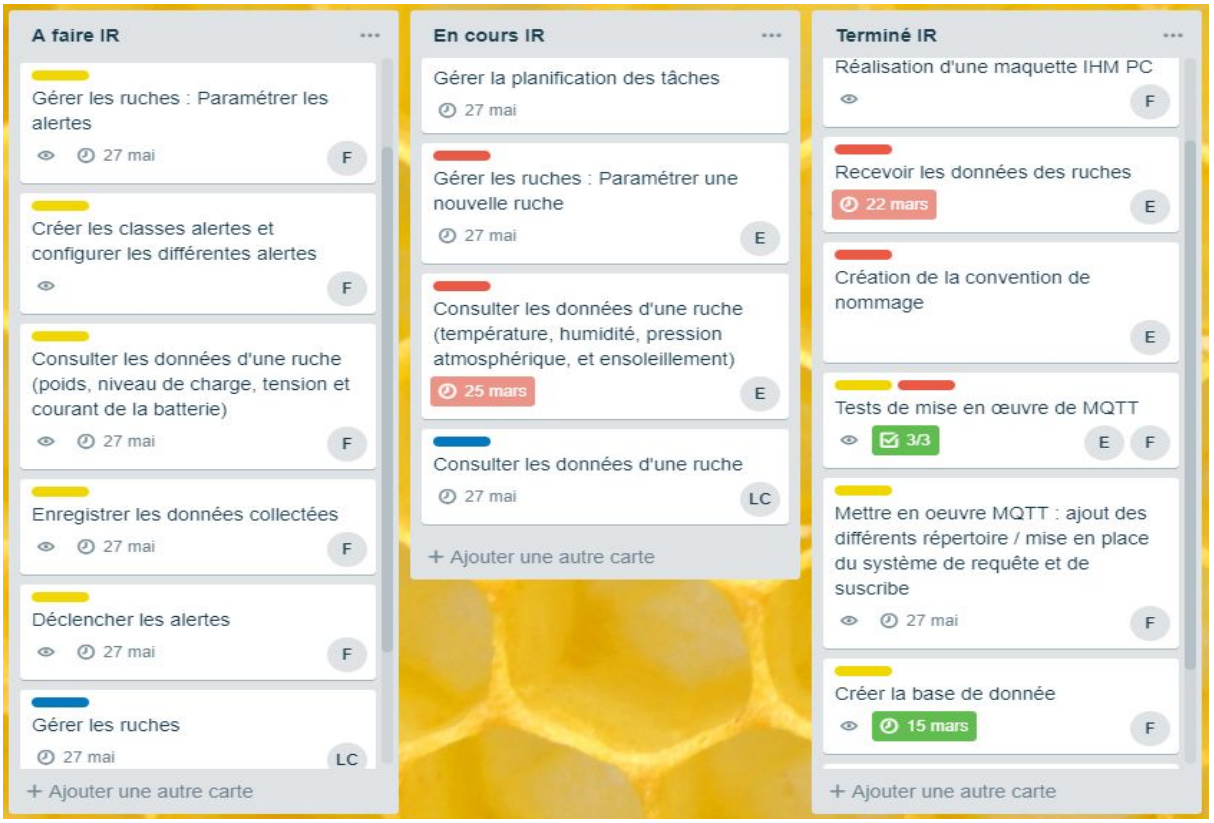
L'apiculteur peut disposer de plusieurs ruches. Elles seront toutes équipées d'une carte **The Things Uno** sur laquelle seront reliées les différents capteurs.

Les ruches se connecteront au réseau The Things Network via une passerelle **The things Gateway**. La zone couverte par une passerelle est d'environ 10 km. Cette passerelle sera reliée au réseau Internet via une liaison WiFi.

L'apiculteur pourra consulter les données de ses ruches à partir d'une application sur **PC** ou sur **terminal mobile** Android. Dans les cas, elle nécessite la présence d'une connexion Internet via un **routeur** d'accès.

Planification

Répartition des Tâches		
ROSSI Enzo Étudiant 3	LAURAIN Clement Étudiant 5	MELLAH Florentin Étudiant 4
❏ Gérer les ruches : Paramétrer une nouvelle ❏ Consulter les données d'une ruche (température, humidité, pression atmosphérique et ensoleillement) ❏ Recevoir les données des ruches	❏ Gérer les ruches ❏ Consulter les données d'une ruche ❏ Lire les données à partir de la base de données.	❏ Gérer les ruches : Paramétrer les alertes ❏ Consulter les données d'une ruche (poids, niveau de charge, tension et courant de la batterie) ❏ Enregistrer les données collectées ❏ Déclencher les alertes

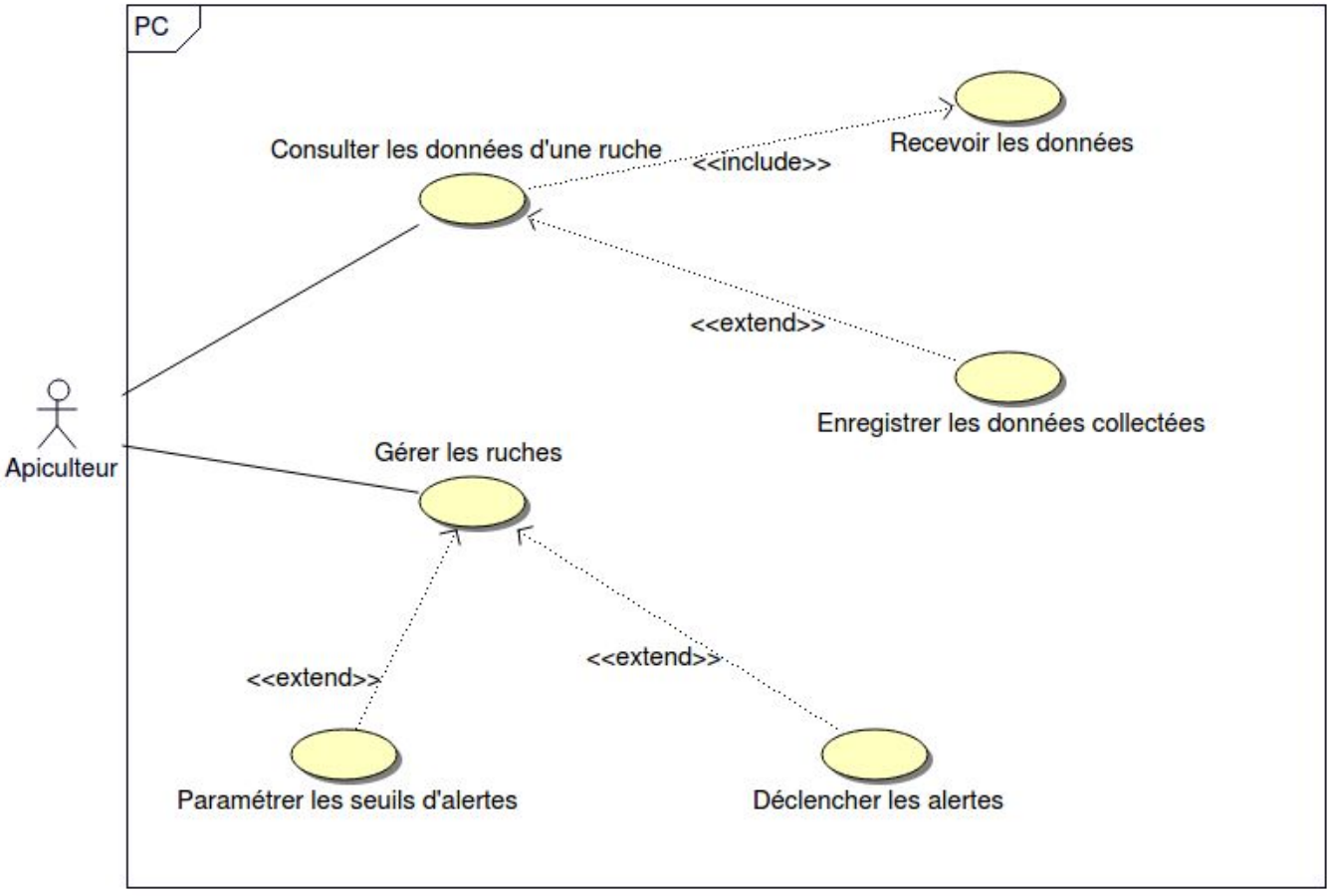

A faire IR
Gérer les ruches : Paramétrer les alertes
27 mai
F
Créer les classes alertes et configurer les différentes alertes
F
Consulter les données d'une ruche (poids, niveau de charge, tension et courant de la batterie)
27 mai
F
Enregistrer les données collectées
27 mai
F
Déclencher les alertes
27 mai
F
Gérer les ruches
27 mai
LC
+ Ajouter une autre carte
En cours IR
Gérer la planification des tâches
27 mai
Gérer les ruches : Paramétrer une nouvelle ruche
27 mai
E
Consulter les données d'une ruche (température, humidité, pression atmosphérique, et ensoleillement)
25 mars
E
Consulter les données d'une ruche
27 mai
LC
+ Ajouter une autre carte
Terminé IR
Réalisation d'une maquette IHM PC
F
Recevoir les données des ruches
22 mars
E
Création de la convention de nommage
E
Tests de mise en œuvre de MQTT
3/3
E
F
Mettre en oeuvre MQTT : ajout des différents répertoire / mise en place du système de requête et de suscribe
27 mai
F
Créer la base de donnée
15 mars
F
+ Ajouter une autre carte

Partie Etudiant 4 : MELLAH Florentin

Objectifs

- Gérer les ruches : Paramétrer les alertes
- Consulter les données d'une ruche (poids, niveau de charge, tension et courant de la batterie)
- Enregistrer les données collectées
- Déclencher les alertes

Diagramme des cas d'utilisation



L'apiculteur peut choisir une ruche pour consulter les données actuelles et/ou enregistrées et les alertes. Les données sont reçues par transmission sans fil via le réseau The Things Network.

Les données collectées sont enregistrées dans la base de données à intervalle de temps régulier. Toutes les heures, on effectue une moyenne des mesures et on les enregistre dans la base de données.

L'apiculteur peut aussi paramétrer les seuils d'alerte (humidité, température, poids, ...) d'une ruche.

The Things Network

The Things Network (**TTN**) est un réseau communautaire et open source mondial pour l'Internet des objets utilisant la technologie **LoRa**. Il est possible d'utiliser librement ce réseau mais il est aussi possible d'aider à étendre le réseau en déployant des passerelles.

Etant donné que nous avons choisi d'utiliser le réseau sans fils **LoraWan** (choix technique car les contraintes techniques de distance et de prix sont respectés avec le LoraWan), le choix c'est porté sur *The Things Network* pour interconnecter les différentes ruches aux applications PC et tablette.

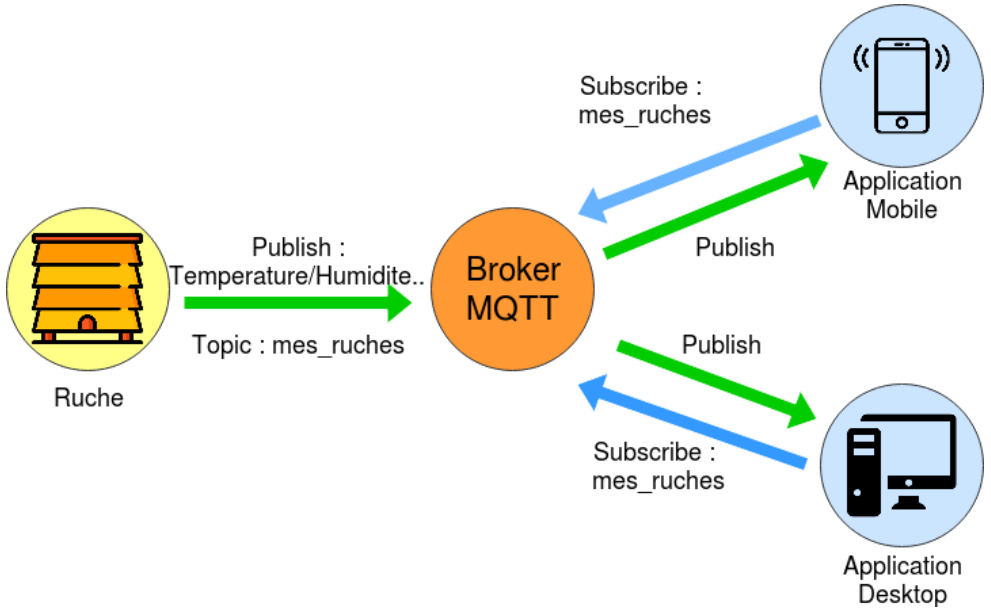
TTN met en oeuvre le protocole **MQTT** pour la transmission et la réception de données.

Protocole MQTT

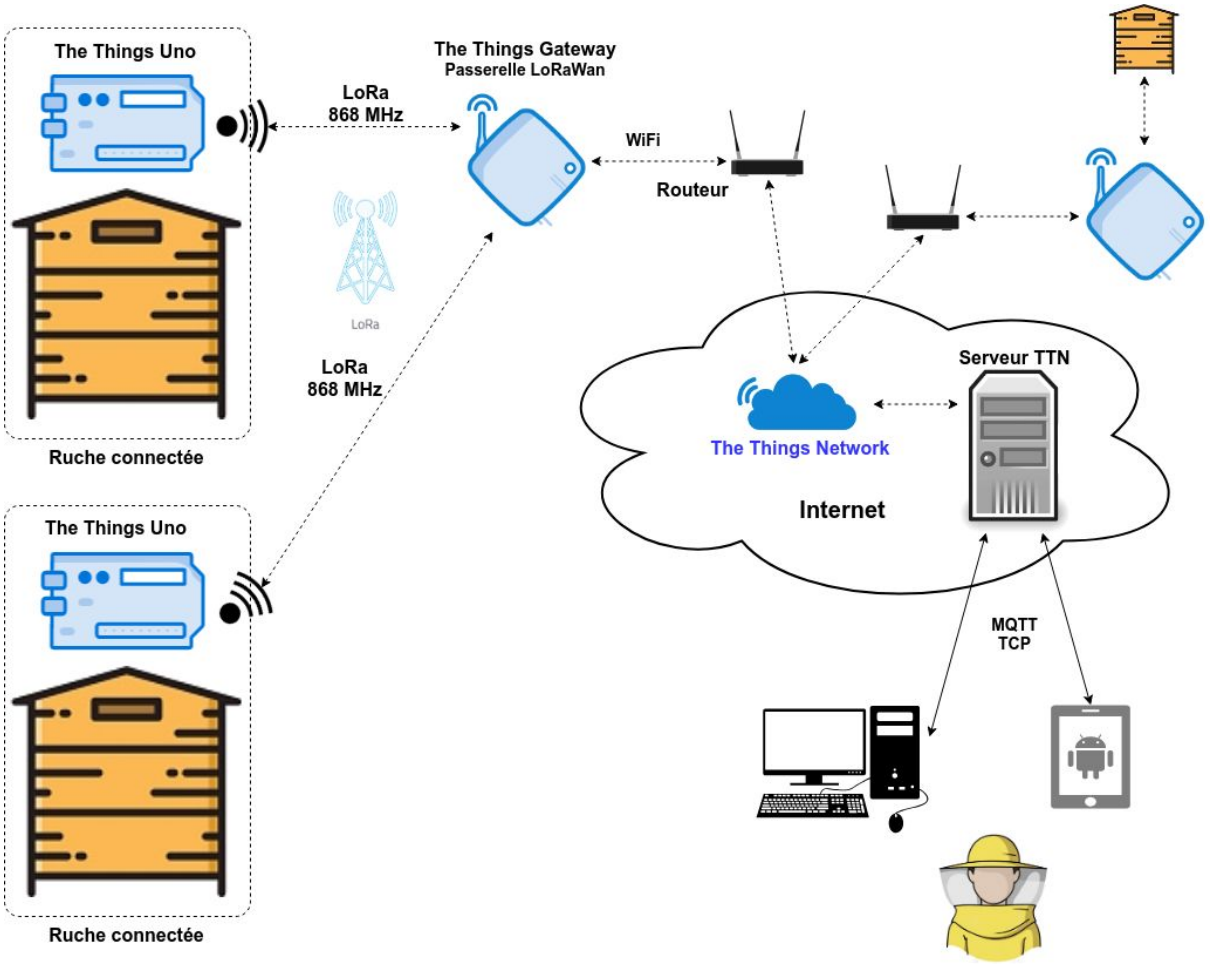
MQTT est un protocole de messagerie basé sur le système *publish-subscribe*. Il utilise les protocoles TCP/IP pour fonctionner.

Le protocole MQTT utilise différentes systèmes et technologies pour fonctionner :

- Broker : Service permettant la mise en oeuvre de gestion de flux.
- Topic : Objet du broker qui sert au fonctionnement du protocole. Permet la publication de données.
- Souscription (*subscribe*) : S'abonner à un Topic pour récupérer les données depuis le serveur MQTT.
- Publication (*publish*) : Envoyer des données sur un Topic du Serveur MQTT.


Subscribe :
mes_ruches
Application
Mobile
Publish
Publish :
Temperature/Humidite..
Broker
MQTT
Topic : mes_ruches
Publish
Subscribe :
mes_ruches
Ruche
Application
Desktop

Schéma


The Things Uno
The Things Gateway
Passerelle LoRaWan
LoRa
868 MHz
WiFi
Routeur
LoRa
LoRa
868 MHz
Serveur TTN
The Things Network
Internet
Ruche connectée
The Things Uno
MQTT
TCP
Ruche connectée

Afin de pouvoir publier des données sur le serveur *The Things Network* il faut créer un chemin d'accès au topic :

<AppID>/devices/DeviceID/down

Afin de pouvoir souscrire à un topic (pour recevoir des données) du serveur *The Things Network* il faut créer un chemin d'accès au topic :

<AppID>/devices/DeviceID/up

- *AppID* : Spécifie le nom de l'application sur lequel nous allons souscrire ou publier des données. Cela représente un ensemble de ruches pour un apiculteur.
- *devices* : Spécifie que nous allons souscrire ou publier des données sur un objet connecté.
- *DeviceID* : Spécifie le nom de l'objet connecté (ici, le nom de la carte intégrée à une ruche de l'apiculteur).
- *down* : Spécifie le sens vers la ruche..
- *up* : Spécifie le sens depuis la ruche.

Pour tester le fonctionnement du protocole MQTT, on a tout d'abord mis en place un serveur MQTT local avec **Mosquitto**.

Mosquitto est un broker MQTT (non utilisé dans le projet mais seulement pour les tests). Il fournit aussi des commandes pour faire des *subscribe* et des *publish*.

Ici, le broker MQTT est un serveur *The Things Network* (TTN).

Commandes avec mosquitto :

- *Subscribe* :

```
$ mosquitto_sub -h eu.thethings.network -t
'mes_ruches/devices/capteurs/up' -u 'mes_ruches' -P
'ttn-account-v2.vC-aqMRnLLzGkNjODWgy81kLqzxBPAT8_mE-L7U2C_w'
```

-h : spécifie l'adresse du *broker*.
-t : spécifie le *topic* auquel on souscrit.
-u : spécifie le nom du compte (ici l'application ID dans TTN) sur le *broker*.
-P : spécifie la clé d'accès pour l'authentification.

- *Publish* :

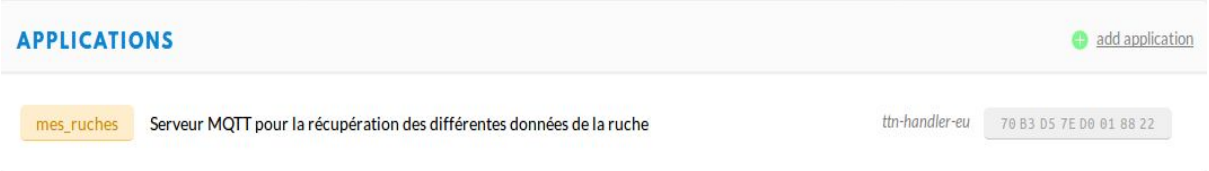
```
$ mosquitto_pub -h eu.thethings.network -t
```

```
'mes_ruches/devices/capteurs/down' -u 'mes_ruches' -P
'ttn-account-v2.vC-aqMRnLLzGkNjODWgy81kLqzxBPAT8_mE-L7U2C_w' -m
'{"payload_fields":{"temperature":21}}' -v
```

-m : Sert à écrire sur le *topic* avec le format JSON ou Octet.
-v : Spécifie la version de MQTT utilisé.

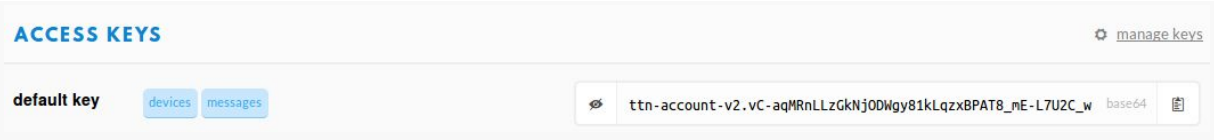
La console TTN

La gestion du réseau The Things Network est réalisée à partir d'une console une fois authentifié au service fourni :

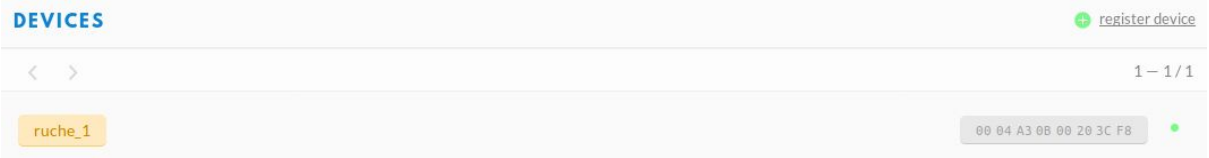


- Ici, l'*AppID* utilisé est **mes_ruches.**

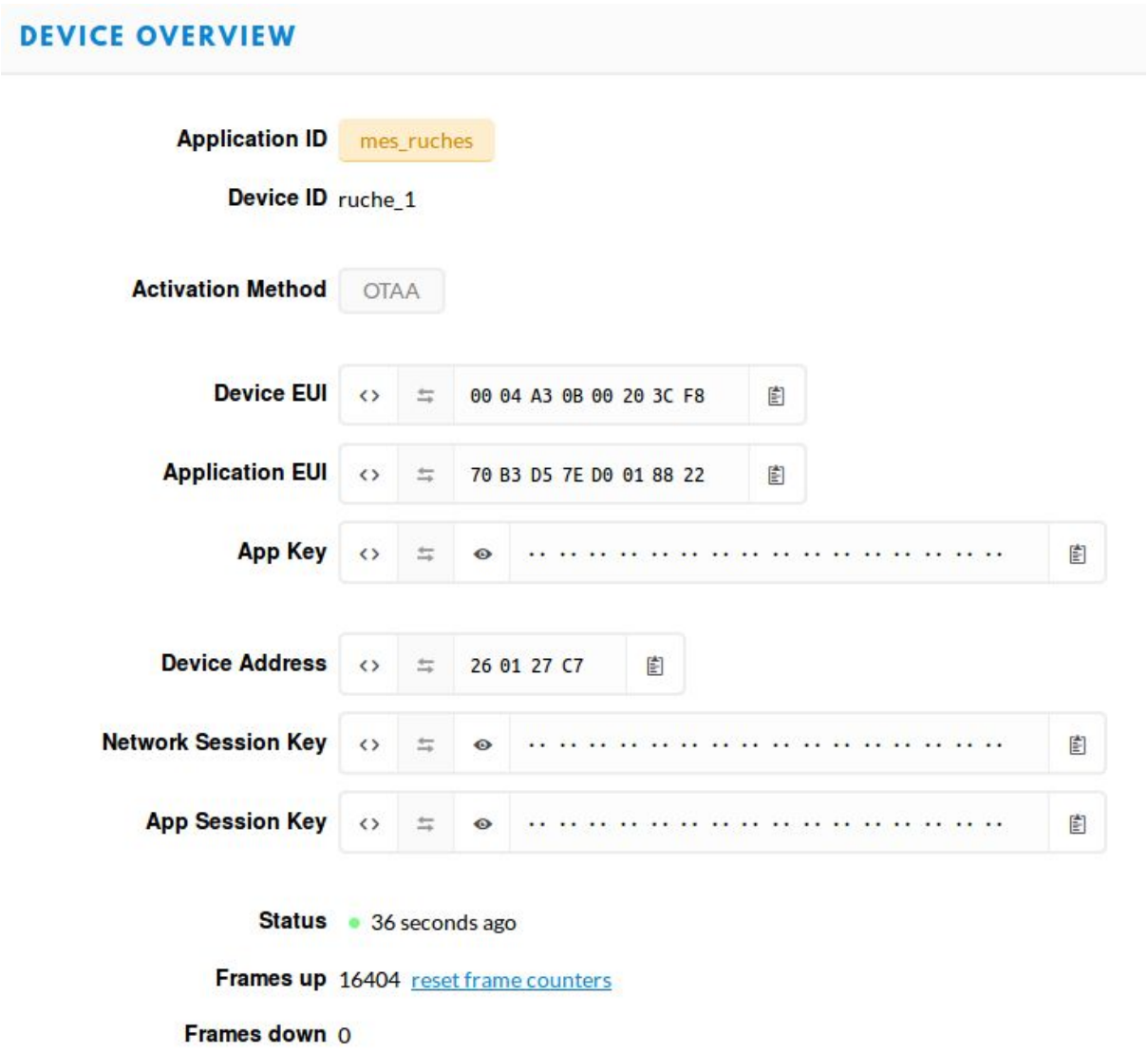
- Le détail de l'application.



- L'*Access Key* (Clé d'accès) permet d'accéder à l'application, dans le programme de la ruche elle est forcément intégrée afin de se connecter au ***Topic***.



- Le *DeviceID* qui identifie le nom de l'objet connecté, ici **ruche_1**.



- Détails de l'objet connecté.

Connexion au serveur TTN

- Dans la classe **parametre.h** : les différents paramètres de connexion par défaut au serveur.

```
// Paramètres de connexion au serveur The Things Network
#define TTN_SERVEUR   "eu.thethings.network"
#define TTN_PORT      1883
#define TTN_USERNAME  "mes_ruches"
#define TTN_PASSWORD
"ttn-account-v2.vC-aqMRnLLzGkNjODWgy81kLqzxBPAT8_mE-L7U2C_w"
```

```
#define TTN_TOPIC      "mes_ruches/devices/ruche_1/up"
```

- Dans la classe **communication.cpp**, les différentes méthodes fournies par la classe **QMqttClient** pour fixer les paramètres de connexion au serveur et l'appel à la méthode **connecToHost()** permettant de se connecter au serveur.

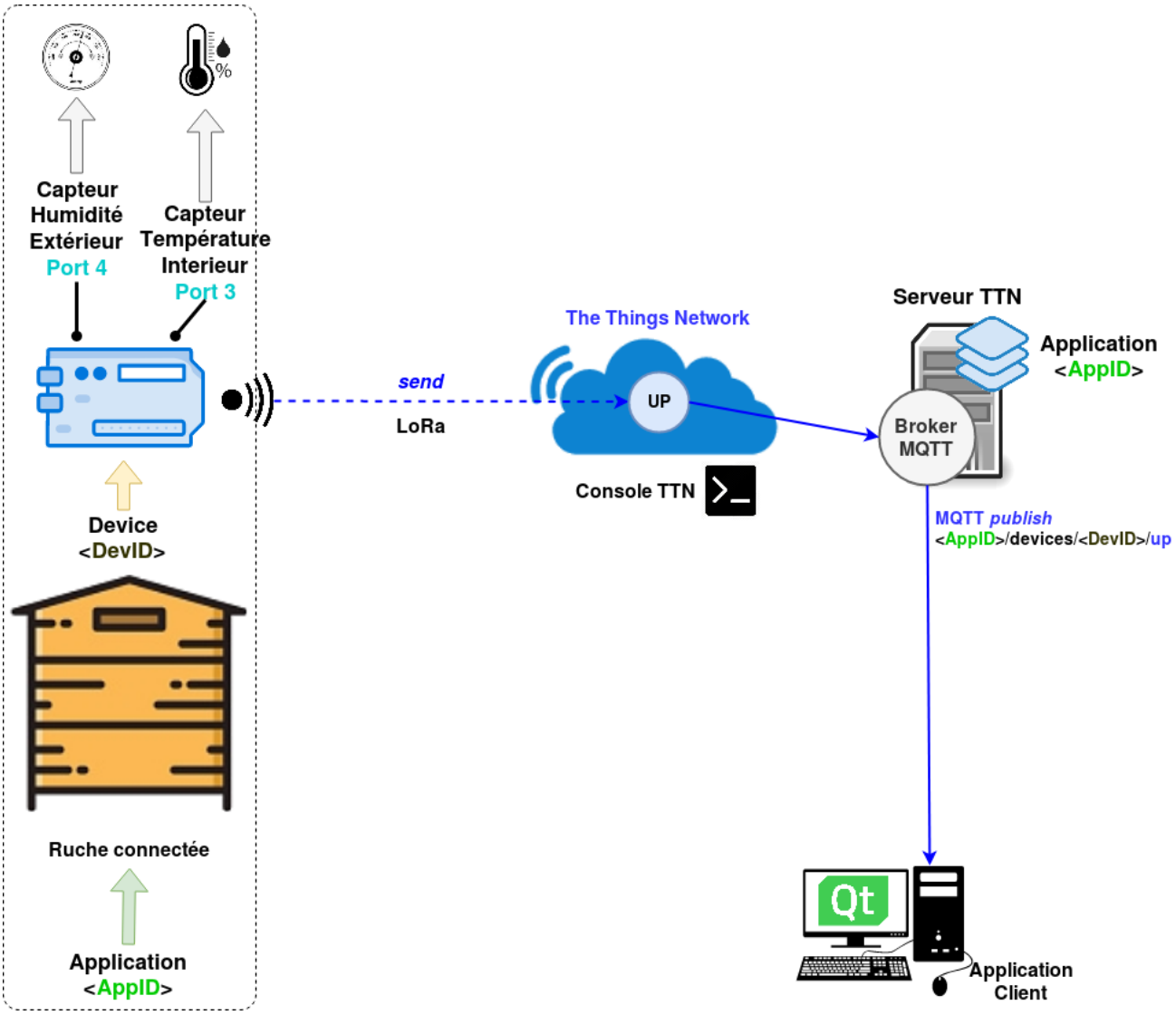
```
client = new QMqttClient();
client->setHostname(TTN_SERVEUR);
client->setPort(TTN_PORT);
client->setUsername(TTN_USERNAME);
client->setPassword(TTN_PASSWORD);
client->connectToHost();
```

La classe QMqttClient fournit deux signaux pour gérer l'état de connexion et de déconnexion au serveur : **connecte d()** et **disconnected()**. Ils ont été connectés au slots suivants :

```
void Communication::connecteTTN()
{
	qDebug() << Q_FUNC_INFO;
	// Le client est maintenant connecté
	emit etatClientConnexion(false); // pour l'IHM
	// Souscription à un topic
	abonnement = client->subscribe(nomTopic);
	if (!abonnement)
	{
	qDebug() << Q_FUNC_INFO << "Impossible de s'abonner au broker TTN
!";
	QMessageBox::critical(0, QString::fromUtf8(APP_TITRE),
QString::fromUtf8("Impossible de s'abonner au broker The Things
Network!"));
	}
}
void Communication::deconnecteTTN()
{
	qDebug()<< Q_FUNC_INFO;
	// Le client est maintenant déconnecté
	emit etatClientConnexion(false); // pour l'IHM
}
```

Notion de port TTN

- Plusieurs ports ont été configurés afin de séparer la communication des différentes données envoyées par la ruche
- Les ports servent à définir quelles données vont être envoyés sur le port en question



- Ici, l'objet connecté publie les données d'humidité extérieure sur le port 4 et de température intérieure sur le port 3.

```
typedef enum
```

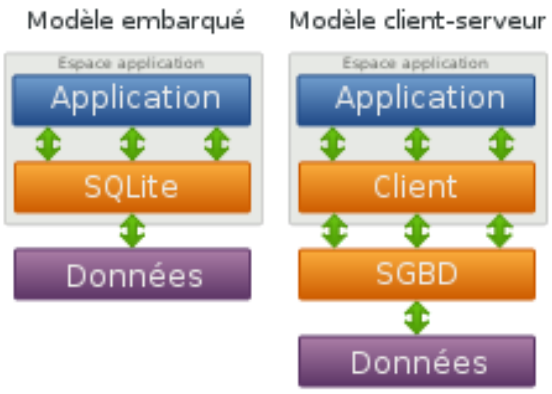
```
{
      portInconnu = 0,
      portMesureEnergie = 1,
      portMesurePoids,
      portMesureRuche,
      portMesureEnvironement,
      portMesureEnsoleillement,
      nbPortsTTN
} PortsTTN;
```

Base de données et MySQL

Pour la création et l'utilisation de base de données, le langage utilisé pour les bases de données est SQL, les deux plus grands systèmes de gestion de base de données sont **SQLite** et **MySQL**. Ces deux système de gestion de base de données sont basés sur un système **relationnel**. C'est à dire que la base de données stocke les informations sont stockés dans des tableaux à deux dimensions appelés **relations** ou **tables.** Les lignes de ces tables sont appelées des ***nuplets*** ou **enregistrements**. Les colonnes sont appelées des **attributs**.

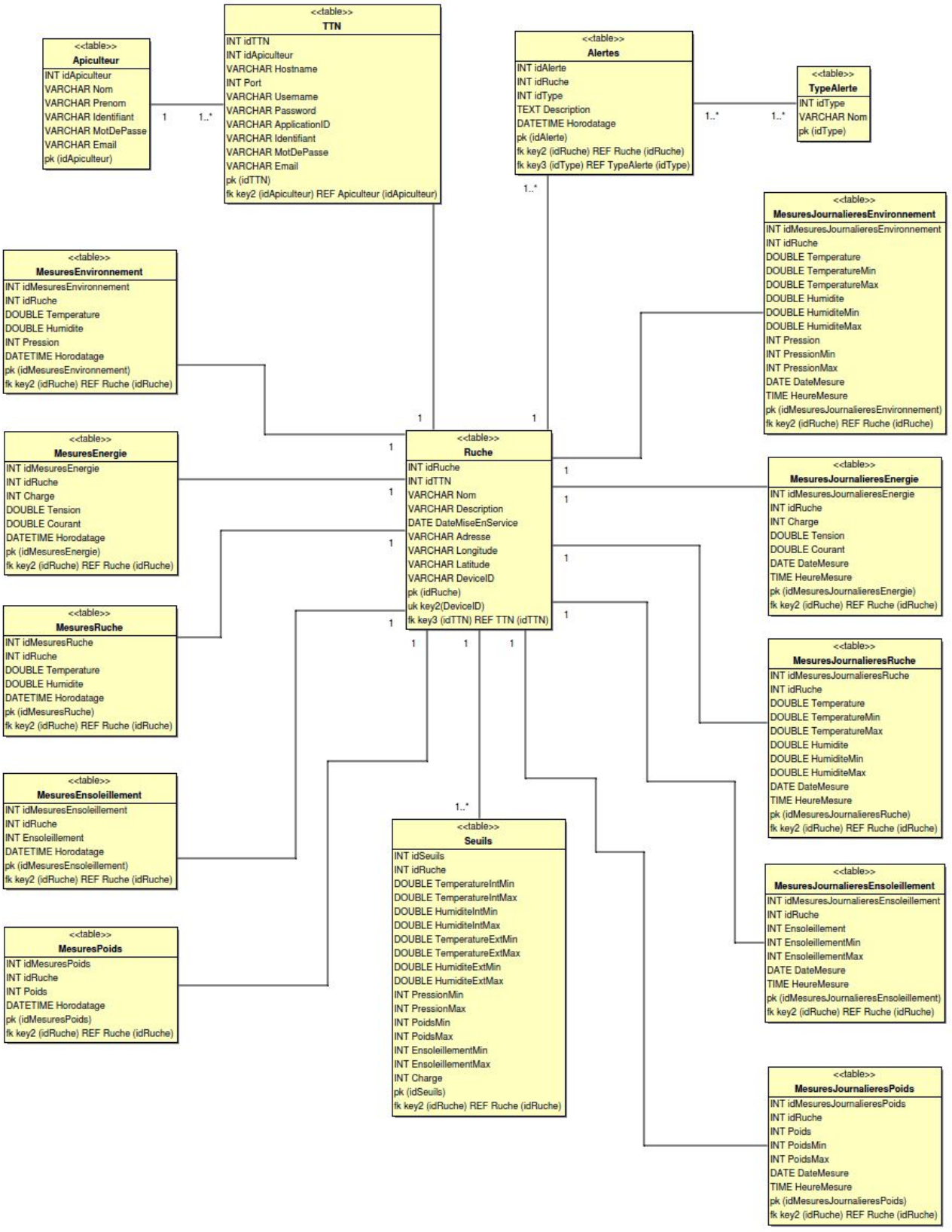
La grande différence entre SQLite et MySQL se fait à la façon de stocker la base de donnée :

- **MySQL** : système de gestion de base de données basé sur la relation client/serveur. La base de donnée est donc centralisée et est accessible par tout le monde.
- **SQLite** : système de gestion de base de données embarqué, c'est à dire que la base de donnée est personnel et est stockée dans un fichier.



Le choix c'est porté sur MySQL car le besoin du client est de centraliser les données sur une seul et même base de données afin de pourvoir faire des études de moyennes et des graphiques sur les différentes données de la ruche.

Schéma relationnel de la base de données



- **Ruche** : contient toutes les caractéristiques d'une ruche (Nom, Description; Date de mise en service, Adresse, Longitude, Latitude, le nom de l'objet connecté, les clefs étrangère vers Apiculteur et RucheTTN).
- **Apiculteur** : contient toutes les caractéristiques d'un apiculteur (Nom, Prénom, Identifiant, Mot de passe, Email).
- **TTN** : contient tous les paramètres de connection au serveur TTN.
- **MesuresEnvironnement** : contient les mesures d'ensoleillement, de température extérieure, d'humidité extérieure, de pression atmosphérique. Clé étrangère vers Ruche.
- **Alertes** : contient la description des alertes. Clé étrangère vers Ruche et TypeAlertes.
- **TypeAlertes** : contient le type des alertes.
- **MesuresEnergie** : contient les mesure énergétique de la ruche (tension, courant, charge, niveau de batterie). Clé étrangère vers Ruche.
- **MesuresRuche** : contient les mesures de température et d'humidité à l'intérieur de la ruche. Clé étrangère vers Ruche.
- **MesuresPoids** : contient le poids de la ruche. Clé étrangère vers Ruche.
- **MesuresJournalieresEnvironnement** : contient toutes les moyennes de mesure d'environnement par jour et les mesures minimum et maximum relevées.
- **MesuresJournalieresEnergie** : contient toutes les moyennes de mesure d'energie par jour et les mesures minimum et maximum relevées.
- **MesuresJournalieresEnsoleillement** : contient toutes les moyennes de mesure d'ensoleillement par jour et les mesures minimum et maximum relevées.
- MesuresJournalieresRuche : contient toutes les moyennes de mesure de la ruche par jour et les mesures minimum et maximum relevées.
- **MesuresJournalieresPoids** : contient toutes les moyennes de mesure de poids par jour et les mesures minimum et maximum relevées.
- **Seuils** : contient les limites que ne doivent pas dépasser les mesures. Clé étrangère vers Ruche.

Création de la base de données MySQL

- Ligne de commande :

```
$ mysql -ufmellah -ppassword -hlocalhost
```

Permet la connection à la base de données avec l'identifiant et mot de passe ainsi que l'adresse du serveur(ici *localhost*)

```
$ mysql> show databases;
+--------------------+
| Database           |
+--------------------+
```

```
| information_schema |
| ruches             |
+--------------------+
2 rows in set (0,03 sec)
```

La commande *show database* permet de voir toute les bases de données présentes sur notre serveur MySQL : ici, notre base de données est ruche.

```
+----------------------+
| Tables_in_ruches     |
+----------------------+
| Alertes              |
| Apiculteur           |
| MesuresEnergie       |
| MesuresEnvironnement |
| MesuresRuche         |
| Ruche                |
| RucheTTN             |
| Seuils               |
```

La commande *use ruches* permet de sélectionner la base de données sur laquelle on veut travailler. *show tables* permet de visualiser les différentes tables de la base de données.

- PhpMyAdmin :

PhpMyAdmin est une application web de gestion pour les systèmes de gestion de base de données MySQL.

	Table	Action	Lignes	Type	Interclassement	Taille	Perte
☐	Alertes	Afficher Structure Rechercher Insérer Vider Supprimer	0	InnoDB	utf8_general_ci	48 Kio	-
☐	Apiculteur	Afficher Structure Rechercher Insérer Vider Supprimer	2	InnoDB	utf8_general_ci	16 Kio	-
☐	MesuresEnergie	Afficher Structure Rechercher Insérer Vider Supprimer	58	InnoDB	utf8_general_ci	32 Kio	-
☐	MesuresEnsoleillement	Afficher Structure Rechercher Insérer Vider Supprimer	61	InnoDB	utf8_general_ci	32 Kio	-
☐	MesuresEnvironnement	Afficher Structure Rechercher Insérer Vider Supprimer	69	InnoDB	utf8_general_ci	32 Kio	-
☐	MesuresJournaliersEnergie	Afficher Structure Rechercher Insérer Vider Supprimer	0	InnoDB	utf8_general_ci	32 Kio	-
☐	MesuresJournaliersEnsoleillement	Afficher Structure Rechercher Insérer Vider Supprimer	3	InnoDB	utf8_general_ci	32 Kio	-
☐	MesuresJournaliersEnvironnement	Afficher Structure Rechercher Insérer Vider Supprimer	1	InnoDB	utf8_general_ci	32 Kio	-
☐	MesuresJournaliersPoids	Afficher Structure Rechercher Insérer Vider Supprimer	0	InnoDB	utf8_general_ci	32 Kio	-
☐	MesuresJournaliersRuche	Afficher Structure Rechercher Insérer Vider Supprimer	5	InnoDB	utf8_general_ci	32 Kio	-
☐	MesuresPoids	Afficher Structure Rechercher Insérer Vider Supprimer	58	InnoDB	utf8_general_ci	32 Kio	-
☐	MesuresRuche	Afficher Structure Rechercher Insérer Vider Supprimer	61	InnoDB	utf8_general_ci	32 Kio	-
☐	Ruche	Afficher Structure Rechercher Insérer Vider Supprimer	2	InnoDB	utf8_general_ci	48 Kio	-
☐	Seuils	Afficher Structure Rechercher Insérer Vider Supprimer	2	InnoDB	utf8_general_ci	32 Kio	-
☐	TTN	Afficher Structure Rechercher Insérer Vider Supprimer	1	InnoDB	utf8_general_ci	32 Kio	-
☐	TypeAlerte	Afficher Structure Rechercher Insérer Vider Supprimer	4	InnoDB	utf8_general_ci	16 Kio	-
	16 tables	Somme	327	InnoDB	latin1_swedish_ci	512 Kio	0 o

Ici, nous visualisons avec l'application web, les différentes base de données et les différentes tables de la base de données.

Requêtes SQL d'insertion des données de poids et de batterie :

```
void Ruche::insererDonneesPortPoids()
{
	QDateTime dateTimePortMesurePoids =
QDateTime::fromString(infosPoids->getHorodatagePoids(),"dd/MM/yyyy
HH:mm:ss");

	QString requete = "INSERT INTO MesuresPoids(idRuche, Poids,
Horodatage) VALUES ('" + donneesRucheTTN.at(0) + "','"  +
donneesRuche.poids + "','" +
dateTimePortMesurePoids.toString("yyyy-MM-dd  HH:mm:ss") + "')";
	bdd->executer(requete);
}
```

- La méthode `insererDonneesPortPoids()` utilise une requête SQL **INSERT** qui permet d'insérer des données dans une table de la base de données.
- Cette requête utilise la clé étrangère **idRuche** pour pouvoir identifier à quelle ruche correspond les mesures relevées.

```
void Ruche::insererDonneesPortPoids()
{
	QDateTime dateTimePortMesurePoids =
QDateTime::fromString(infosPoids->getHorodatagePoids(),"dd/MM/yyyy
HH:mm:ss");
	QString requete = "INSERT INTO MesuresPoids(idRuche, Poids,
Horodatage) VALUES ('" + donneesRucheTTN.at(0) + "','"  +
donneesRuche.poids + "','" +
dateTimePortMesurePoids.toString("yyyy-MM-dd  HH:mm:ss") + "')";
	bdd->executer(requete);
}
```

- Même requête **INSERT** qui permet d'insérer les données de poids correspondant à l'**idRuche**.
- Cette méthode utilise une méthode de la classe BaseDeDonnees grâce à un pointeur sur cette classe : `bdd->executer(requete)`. Cette requête permet d'exécuter toutes les requêtes de type **INSERT** , **UPDATE** et **DELETE** lorsque la connection à la base de données est effectué :

```
bool BaseDeDonnees::executer(QString requete)
{
	QMutexLocker verrou(&mutex);
	QSqlQuery r;
	bool retour;

	if(db.isOpen())
	{
	  if(requete.contains("UPDATE") || requete.contains("INSERT") ||
requete.contains("DELETE"))
	  {
	      retour = r.exec(requete);
	      if(retour)
	      {
	            return true;
	      }
	      else
	      {
	            return false;
	      }
	  }
	  else
	  {
	      return false;
```

```
        }
      }
      else
          return false;
}
```

Récupération des seuils d'alertes depuis la base de données :

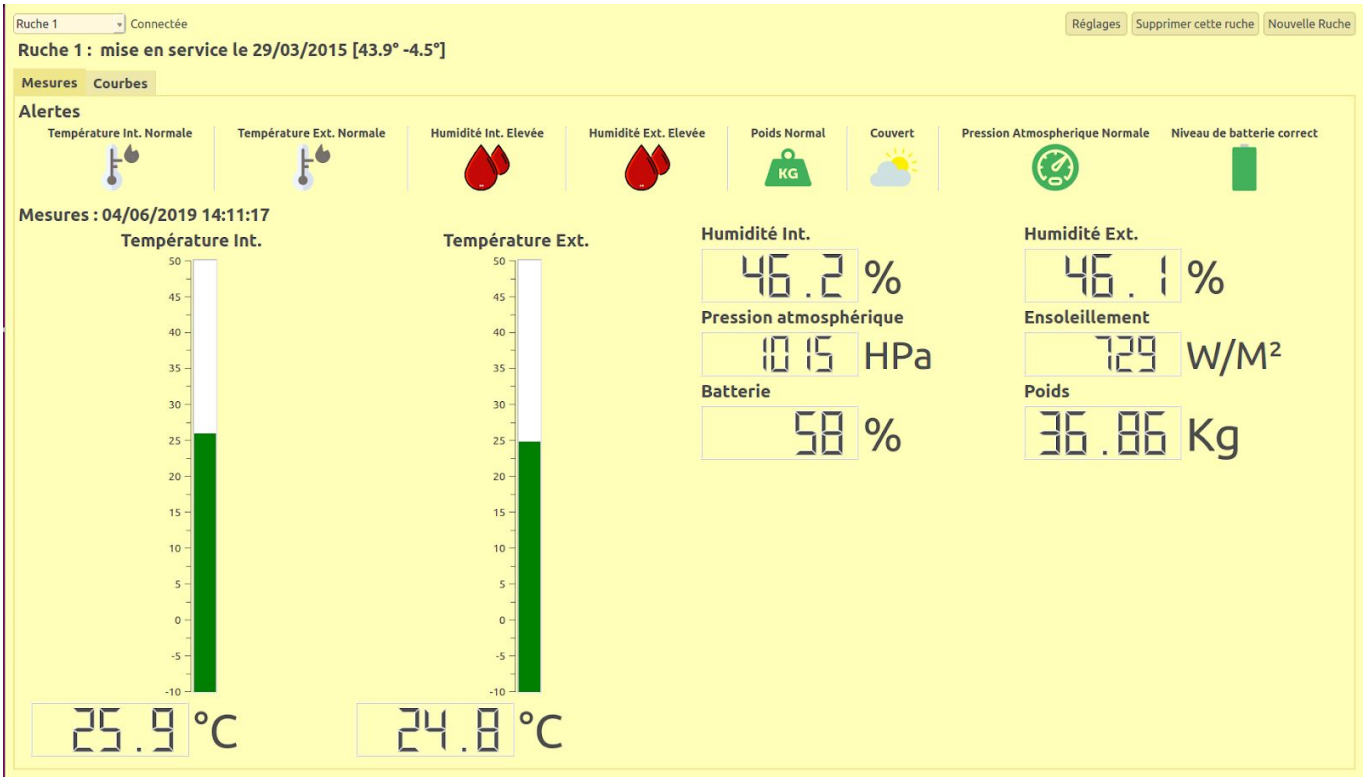
- La requête suivante permet de récupérer les seuils d'alertes de la base de données, cette requête se trouve dans le constructeur de la classe **Alerte** :

```
QString requete = "SELECT * FROM Seuils WHERE idRuche='"+ idRuche +"'";
```

- La requête suivante est dans la méthode `recevoirReglagesAlertes()` dans la classe **ReglagesAlertesIhm**, cette requête permet d'update les seuils dans la base de données lorsque l'utilisateur modifie les seuils depuis l'interface homme / machine :

```
QString requete = "UPDATE Seuils SET TemperatureIntMin='" +
temperatureInterieurMin + "', TemperatureIntMax='" +
temperatureInterieurMax + "', HumiditeIntMin='" + humiditeInterieurMin +
"', HumiditeIntMax='" + humiditeInterieurMax + "', TemperatureExtMin='"
+ temperatureExterieurMin + "', TemperatureExtMax='" +
temperatureExterieurMax + "', HumiditeExtMin='" + humiditeExterieurMin +
"', HumiditeExtMax='" + humiditeExterieurMax +  ", PressionMin='" +
pressionAtmospheriqueMin + ", PressionMax='" +
QString::number(PRESSION_ATMOSPHERIQUE_SEUIL_MAX) + ", PoidsMin='" +
poidsMin + "', PoidsMax='" + poidsMax + "' WHERE idRuche='" +
this->alertes->getIdRuche() + "'";
```

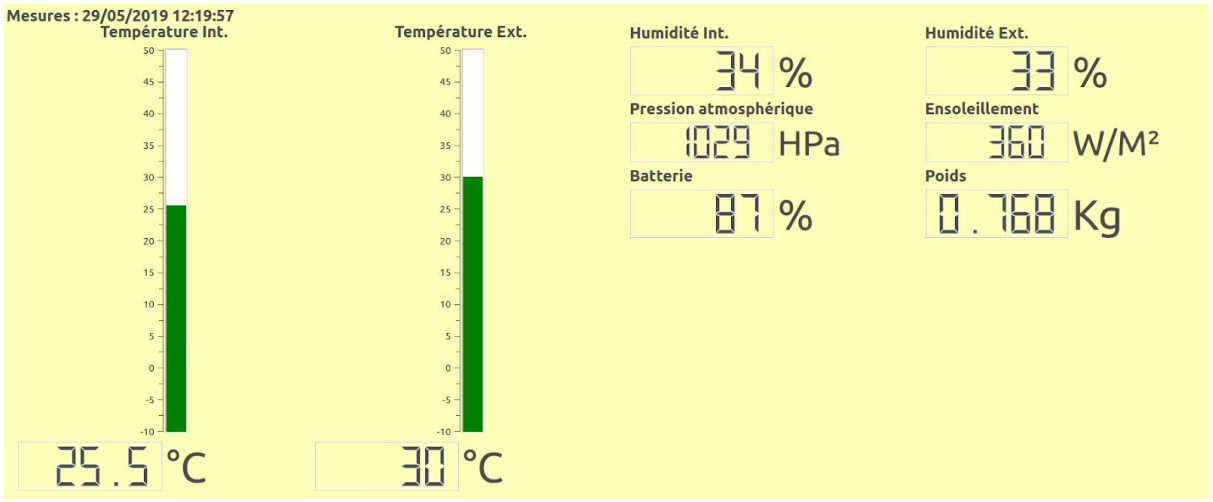
Interface Homme/Machine



Explication de l'interface homme machine :

- Cette partie permet de sélectionner la ruche que nous voulons afficher sur l'interface.
- L'emplacement de la ruche sélectionnée est défini ainsi que sa date de mise en service.
- Le paramétrages des différents seuils d'alerte sont définis par défaut mais il est possible de les modifier en cliquant sur le bouton Réglages.
- La création d'une nouvelle ruche peut être fait en cliquant sur le bouton Nouvelle Ruche.
- L'utilisateur pourra choisir d'afficher les mesures ou d'afficher les graphiques affichant les moyennes des mesures sur la dernière heure.

- Les différentes alertes sont affichés sur cette partie de l'interface.



- L'affichage des mesures s'effectue sur cette partie de l'interface.

Diagrammes de classes :

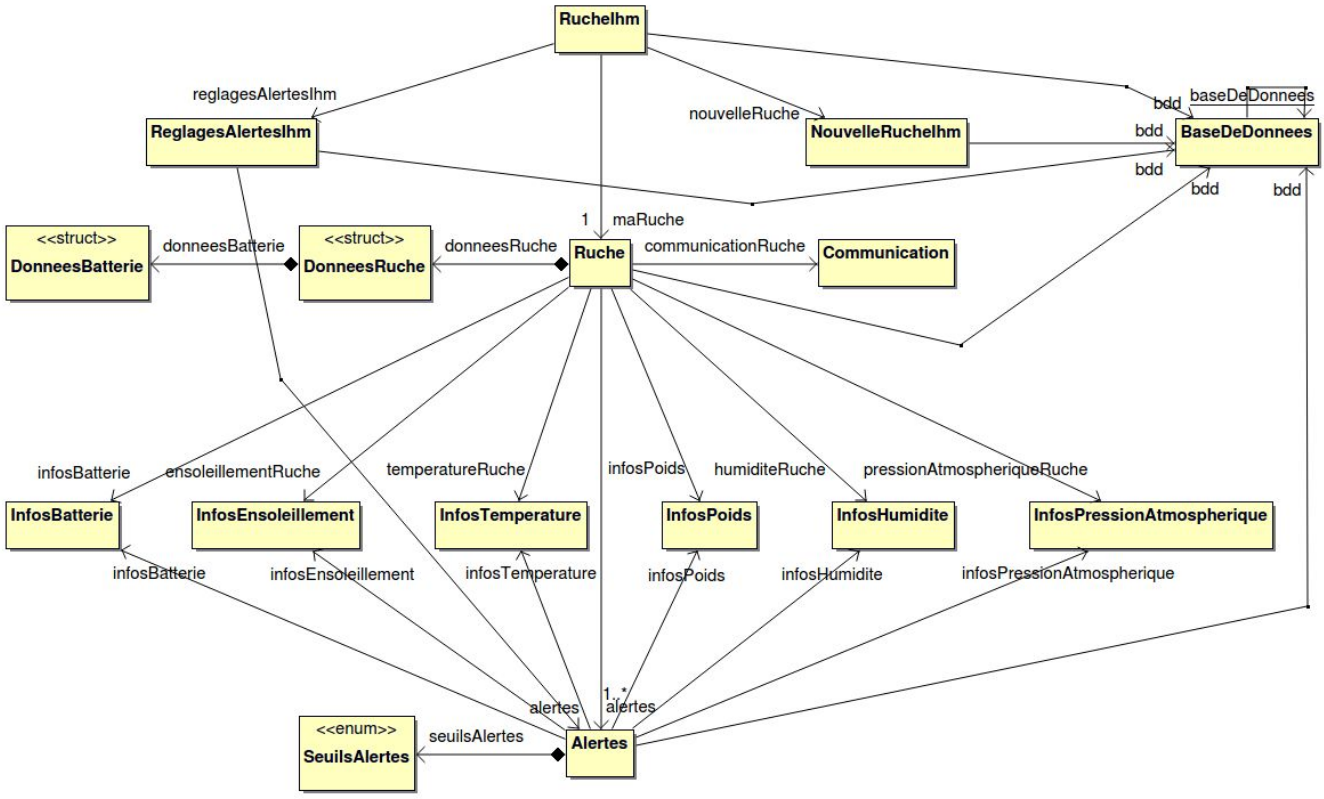
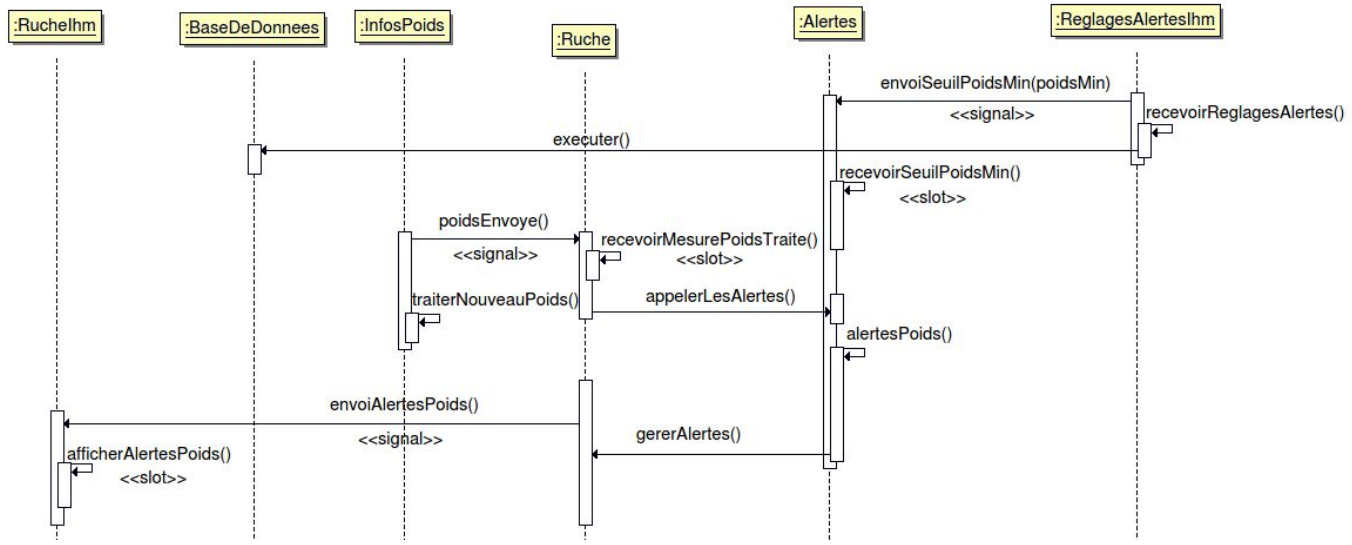


Diagramme de séquence : régler les seuils d'alerte et déclencher les alertes



- L'utilisateur modifie les seuils d'alertes en cliquant sur le bouton "Réglages", la classe **ReglagesAlertesIhm** récupère ces nouveaux seuils grâce à la méthode `recevoirReglagesAlertes()`. Elle enregistre ces nouveaux seuils dans la table **Seuils** de la base de données.

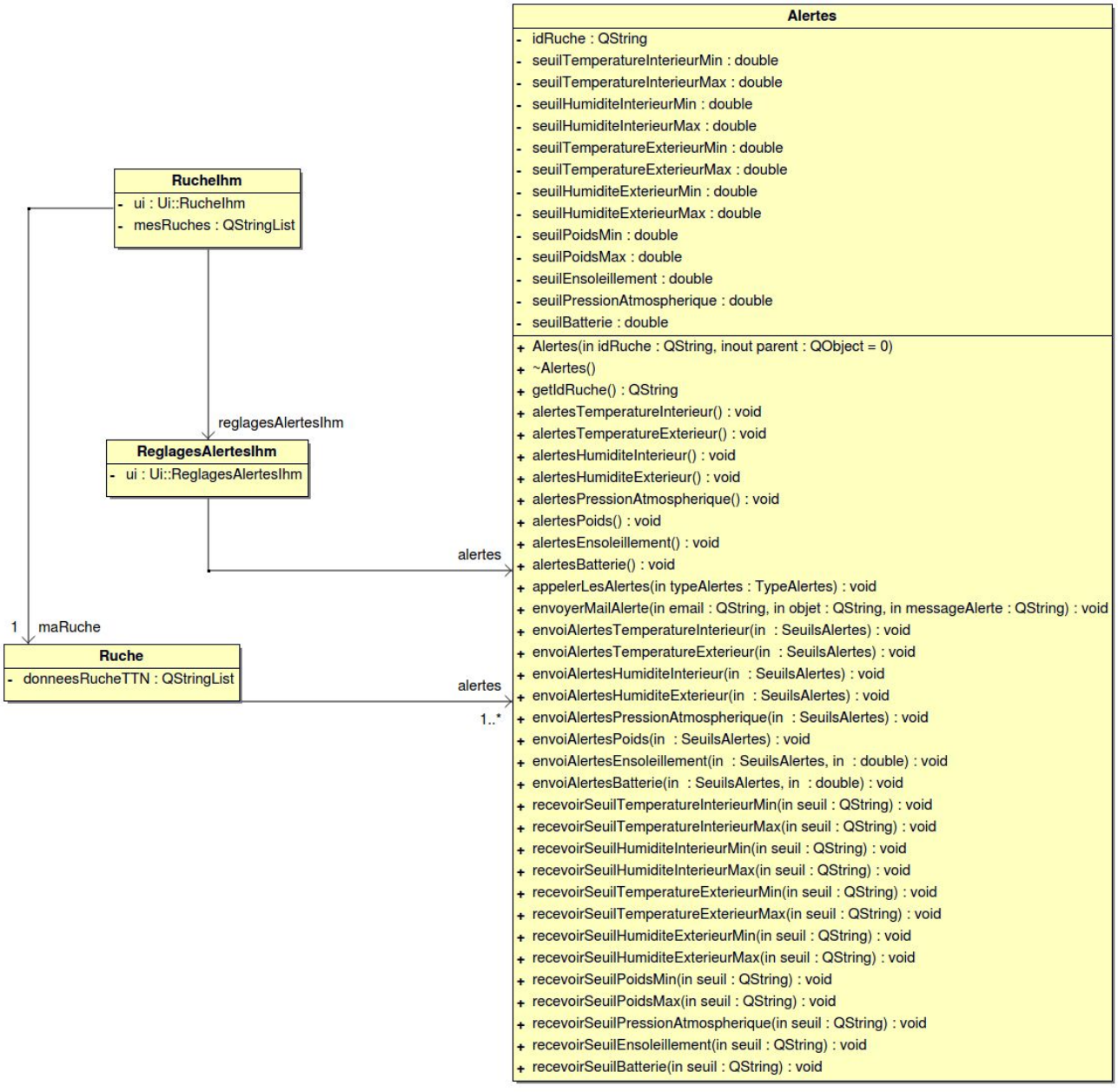
- La classe **ReglagesAlertesIhm** envoi les nouveaux seuil grâce à un signal `envoiSeuilPoidsMin(poidsMin)` qui déclenche le slot `recevoirSeuilPoidsMin()` de la classe **Alertes**. Un **signal** c'est un message envoyé lorsqu'un événement se produit. Ce signal sera **connecté** à un **slot**. Un slot c'est la fonction qui est appelée lorsqu'un événement s'est produit. On dit que le signal appelle le slot. Concrètement, un slot est une méthode d'une classe.

- Lorsque la classe **Ruche** reçoit des données venant des classes infos, elle appel une méthode `appelerLesAlertes(typeAlertes)` de la classe **Alertes** avec en paramètre le type d'alerte qui dépend de la données reçu, par exemple le poids.

- Une fois que la classe **Alertes** à les seuils et que sa méthode `appelerLesAlertes()` est déclenchée, elle appel la méthode `gereAlertes()` de la classe **Ruche**.

- La classe **Ruche** envoi grâce à un signal `envoiAlertesPoids()` l'alerte détecter à la classe **RucheIhm** qui affichera l'alerte grâce à son slot `afficherAlertesPoids()`.

Gestion des alertes



- La classe **Alertes** a des relations de type **Association** (une association indique qu'il peut y avoir des liens entre des instances des classes associées) avec les classes **ReglagesAlertesIhm** et **Ruche**.
- Cette classe a pour but de mettre en place un système d'alerte pour toutes les mesures physique et météorologique de la ruche.

- Pour pouvoir permettre la mise en place des alertes, un système de seuil à été choisit, des seuils minimums et maximum ont été définis par défaut mais l'utilisateur aura la possibilité de modifier ces seuils en cliquant sur le bouton **Reglages** de l'interface homme / machine. Ces seuils par défaut sont définis dans la classe **Parametres** :

```
// Seuils par défaut des alertes
#define TEMPERATURE_INTERIEUR_SEUIL_MAX         35.0
#define TEMPERATURE_INTERIEUR_SEUIL_MIN         25.
#define HUMIDITE_INTERIEUR_SEUIL_MAX            30.
#define HUMIDITE_INTERIEUR_SEUIL_MIN            20.
#define TEMPERATURE_EXTERIEUR_SEUIL_MAX         35.
#define TEMPERATURE_EXTERIEUR_SEUIL_MIN         5.
#define HUMIDITE_EXTERIEUR_SEUIL_MAX            35.
#define HUMIDITE_EXTERIEUR_SEUIL_MIN            20.
#define PRESSION_ATMOSPHERIQUE_SEUIL_MIN        1000.
#define PRESSION_ATMOSPHERIQUE_SEUIL_MAX        1200.
#define POIDS_SEUIL_MAX                          100.
#define POIDS_SEUIL_MIN                         10.
#define ENSOLEILLEMENT_SEUIL_MAX                1000.
#define ENSOLEILLEMENT_SEUIL_MIN                10.
#define BATTERIE_SEUIL_MIN                      25.
```

- Afin de définir le type l'alerte détecter les méthodes `void alertesXXX()` définissent le type d'alerte.
- Si la mesure relevé est trop haute par rapport au seuil maximum définit alors le type d'alerte envoyé à la classe ruche sera trop haut et inversement. Les méthodes testent aussi si elle ne détectent aucune alertes.

```
void  Alertes::alertesTemperatureInterieur()
{
      double mesureTemperatureInterieur =
infosTemperature->getTemperatureInterieur();
      if(mesureTemperatureInterieur > seuilTemperatureInterieurMax)
      {
         emit envoiAlertesTemperatureInterieur(tropHaut);
      }
      else if (mesureTemperatureInterieur <
seuilTemperatureInterieurMin)
      {
         emit envoiAlertesTemperatureInterieur(tropBas);
      }
      else
```

```
        {
            emit envoiAlertesTemperatureInterieur(bon);
        }
}
```

- Ici `envoiAlertesTemperatureInterieur()` est un **signal**. Ce signal envoi donc un type d'alerte qui est `tropHaut`, ce type d'alerte est défini dans un type enum **SeuilsAlertes** :

```
typedef enum
{
        tropHaut = 0,
        tropBas = 1,
        bon = 2,

} SeuilsAlertes;
```

- Les signaux sont envoyés à la classe **Ruche** afin que celle ci traite les alertes et envoi à son tour par signal à la classe **RucheIhm** qui affichera les alertes dans la partie spécifique à celle ci. Pour pouvoir envoyer un signal à la classe **Ruche** il faut connecter un signal et un slot ensemble :

```
connect(alertes,
SIGNAL(envoiAlertesTemperatureInterieur(SeuilsAlertes)),
this,SLOT(recevoirAlertesTemperatureInterieur(SeuilsAlertes)));
```

- Pour vérifier si il y a une alerte au niveau d'une mesure il faut que dès que la classe **Ruche** reçoit une mesure celle ci devra appeler une méthode de la classe alerte afin de tester la présence d'alerte :

```
void Ruche::recevoirTemperatureInterieurTraite(double
temperatureInterieur, QString horodatage)
{
        alertes->appelerLesAlertes(alerteTemperatureInterieur);
        emit
nouvelleMesureTemperatureInterieurTraite(temperatureInterieur,
horodatage);
}
```

- Ici la méthode `recevoirTemperatureInterieurTraite()` appellera la méthode `appelerLesAlertes()` avec en paramètre les types d'alertes que la méthode devra tester. Ce type d'alerte est définit dans un type enum appeler **TypeAlertes** :

```
typedef enum
{
	alerteTemperatureInterieur = 0,
	alerteTemperatureExterieur = 1,
	alerteHumiditeInterieur = 2,
	alerteHumiditeExterieur = 3,
	alertePressionAtmospherique = 4,
	alertePoids = 5,
	alerteEnsoleillement = 6,
	alerteBatterie = 7,
	toutesLesAlertes = 8,

} TypeAlertes;
```

- Cette méthode effectue un **switch/case** qui permet de tester quel est le type d'alerte reçu en paramètre et lorsque ce test est vérifié celle ci fera appel à la méthode vu plus haut qui effectuera le test afin de vérifier quel est le type d'alerte en fonction des seuils.

```
void Alertes::appelerLesAlertes(TypeAlertes typeAlertes)
{
	switch(typeAlertes)
	{
	case alerteTemperatureInterieur :
		alertesTemperatureInterieur();
		break;
	case alerteTemperatureExterieur :
		alertesTemperatureExterieur();
		break;
	case  alerteHumiditeInterieur :
		alertesHumiditeInterieur();
		break;
	case  alerteHumiditeExterieur :
		alertesHumiditeExterieur();
		break;
	case  alertePressionAtmospherique :
		alertesPressionAtmospherique();
		break;
	case alertePoids :
		alertesPoids();
		break;
```

```
        case alerteEnsoleillement :
              alertesEnsoleillement();
              break;
        case alerteBatterie :
              alertesBatterie();
              break;
        case toutesLesAlertes:
              alertesHumiditeExterieur();
              alertesHumiditeInterieur();
              alertesPressionAtmospherique();
              alertesTemperatureExterieur();
              alertesTemperatureInterieur();
              alertesEnsoleillement();
              alertesPoids();
              break;
        }
}
```

Modifications des seuils :

- Les seuils peuvent être modifiés depuis l'interface homme / machine grâce à la classe **ReglagesAlertesIhm** :

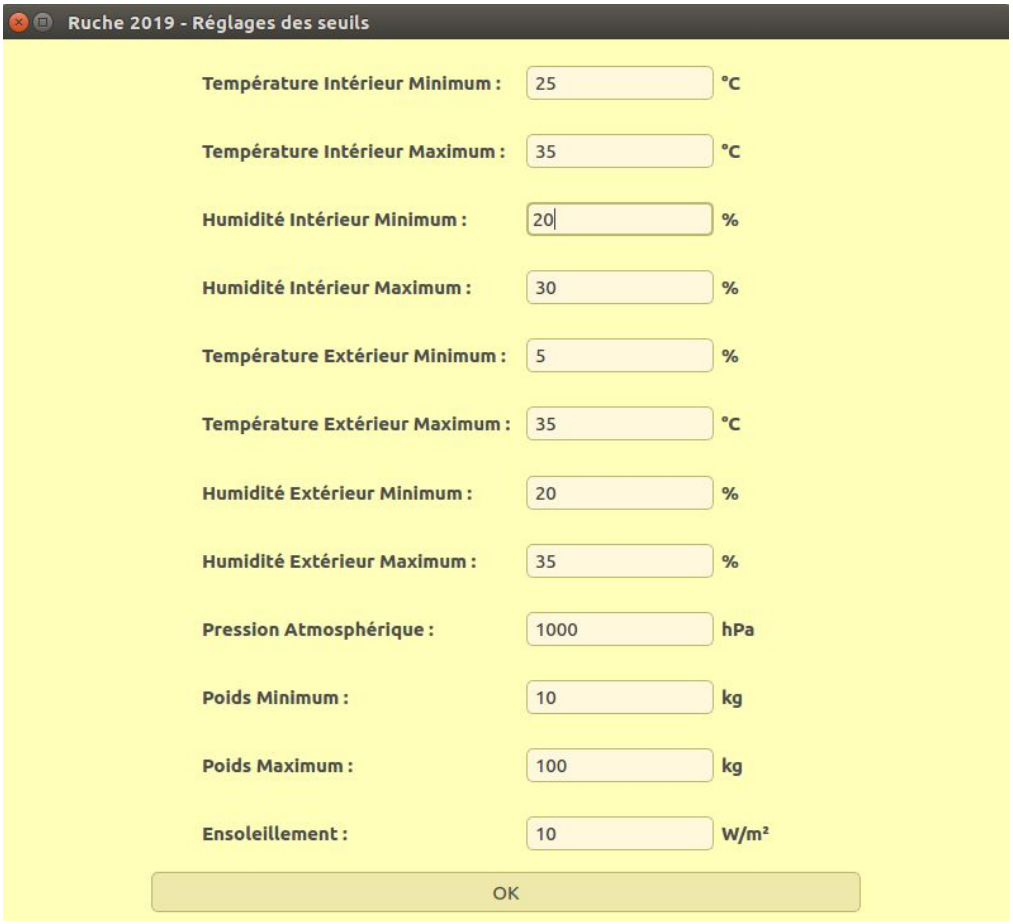
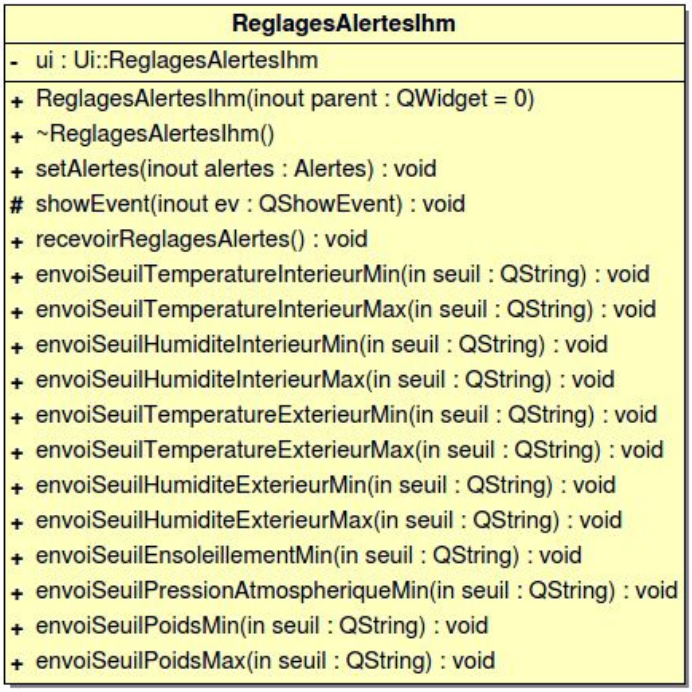


Diagramme de la classe **ReglagesAlertesIhm** :



- La méthode `recevoirReglagesAlerte()` est définie afin que lorsque l'utilisateur modifie un ou plusieurs seuil, ces nouveaux seuils sont stockés dans une variable.

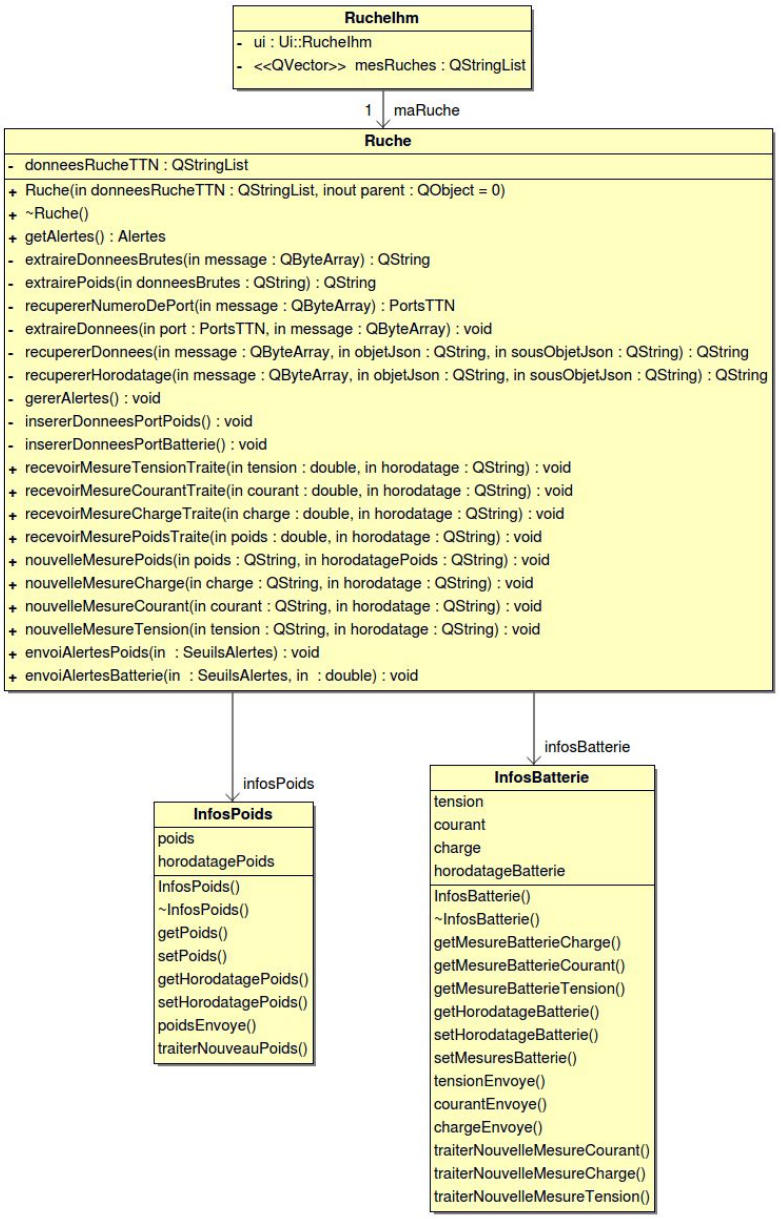
```
void ReglagesAlertesIhm::recevoirReglagesAlertes()
{
      QString temperatureInterieurMin;
      temperatureInterieurMin =
ui->lineEditSeuilTemperatureInterieurMin->text();
}
```

- Les signaux `envoiSeuilXXX()` sont connéctes à des slots de la classe **Alertes** qui recevra les nouveaux seuils et les remplacera les seuils par défaut par ceux ci.

```
connect(this, SIGNAL(envoiSeuilTemperatureInterieurMin(QString)),
this->alertes, SLOT(recevoirSeuilTemperatureInterieurMin(QString)));
```

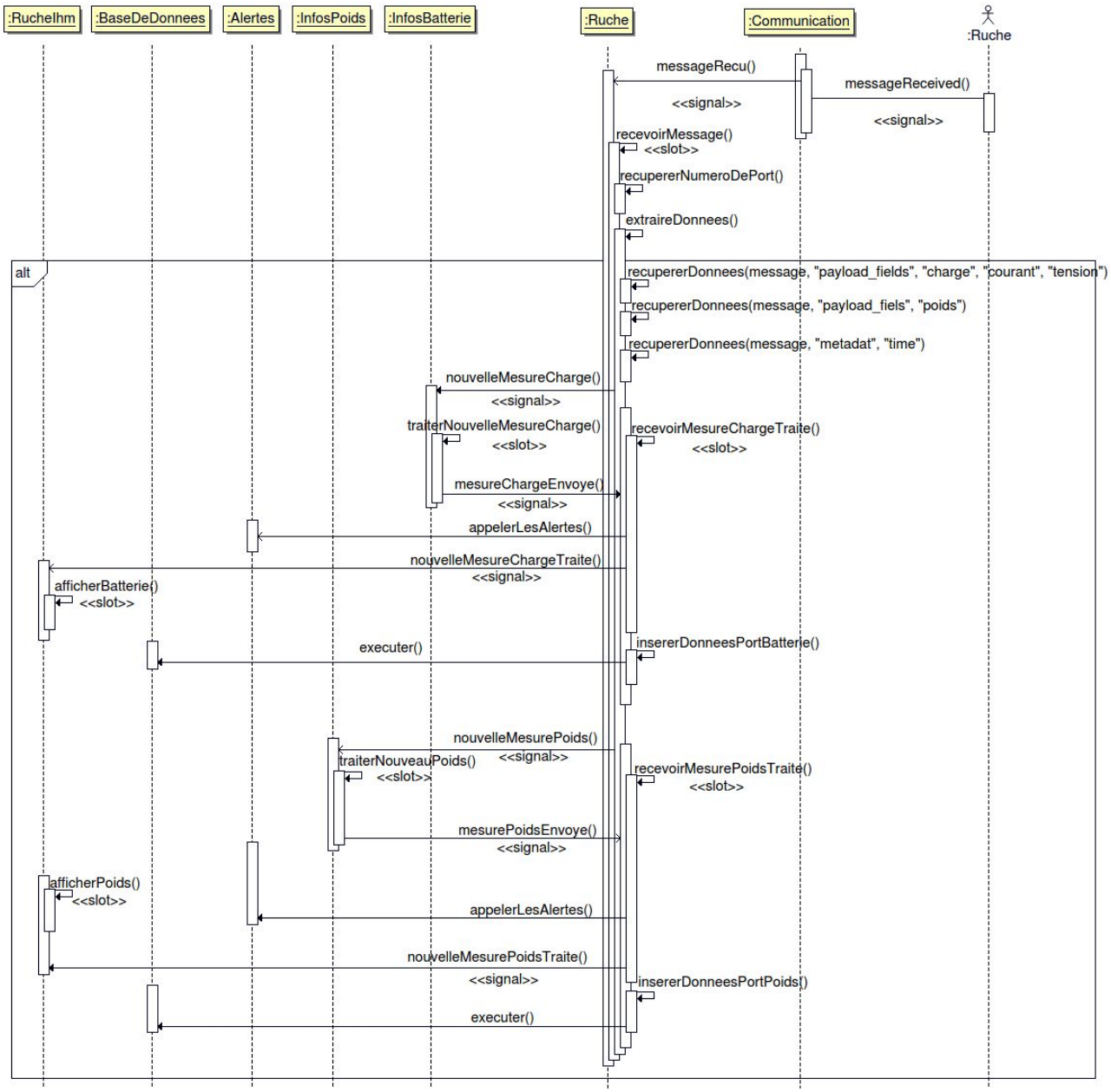
- La classe **Alertes** recupere le nouveau seuil avec une methode `recevoirSeuilXXX()`.

```
void Alertes::recevoirSeuilTemperatureInterieurMin(QString seuil)
{
      seuilTemperatureInterieurMin = seuil.toDouble();
}
```



- Les classes **InfosPoids** et **InfosBatterie** servent à récupérer les données de tension, de courant, de charge et de poids des différents capteurs de la ruche.

Diagramme de séquence : afficher les mesures de batterie et de poids et enregistrer les données dans la base de données



- La méthode `recupererNumeroDePort()` de la classe **Ruche** permet d'extraire le numéro de port dans le fichier Json.

- Apres avoir recuperer le numero de port, la méthode `recupererDonnees()` est appelé est permet d'extraire du fichier JSON, les données correspondant au numéro de port.

- Une fois la donnée extraite elle est envoyée à la classe **Infos** par le signal `nouvelleMesureCharge()`, le slot connecté à ce signal est `traiterNouvelleMesureCharge()` qui remplacera l'ancienne mesure par la nouvelle. Puis la classe **Infos** renvoi la mesure à la classe **Ruche** grâce à un signal.

- La classe **Ruche** appel les alertes afin de vérifier si la mesure reçu dépasse un seuil.

- La classe Ruche finit par envoyer la mesure à la classe **RucheIhm** qui affichera la mesure grâce à la méthode `afficherBatterie()`.

Tests de validation

Désignation	Objectif attendu	Résultat
Gérer les ruches : Paramétrer les alertes	Régler les seuils minimum et maximum de toutes les mesures	Possibilité d'effectuer les réglages en cliquant sur le bouton "Réglages"
Consulter les données d'une ruche (poids, niveau de charge, tension et courant de la batterie)	Visualiser le poids, le niveau de charge de la batterie	Possibilité de visualiser le pourcentage de batterie ainsi que le poids de la ruche en kilogramme
Enregistrer les données collectées	Enregistrement des mesures de poids et de batterie dans la base de données	Les données sont enregistrées dans la base de données, possibilité de les visualiser grâce à l'onglet "Courbe"
Déclencher les alertes	Afficher lorsqu'une mesure dépasse le seuil maximum ou minimum	Visualisation des alertes sur l'interface homme/machine dans la partie spécifique aux alertes

Partie Physique

LoraWan :

- LoRa est un **protocole radio** (niveau physique) conçu par Semtech (Cycleo). LoRa utilise la bande de fréquences **868Mhz**.

- LoRaWAN est la gestion de la **couche MAC**, et permet de façon dynamique d'optimiser le lien entre l'objet LoRa et la station de base : canal de fréquence, puissance d'émission, débit, … . LoRaWAN peut être opéré par un opérateur TélécomBT, Orange, … ou utilisé un réseau privé. Le protocole LoRa est **bidirectionnel** sous conditions.
- Le réseau LoRaWan définit **3 typologies** de composants :
    1. Classe A : nœud de fin de réseau comme les capteurs
    2. Classe B : passerelle
    3. Classe C : serveur de données
-

SigFox :

- Sigfox est un opérateur. Sigfox utilise la bande **868Mhz** en Europe (902Mhz aux US). Un objet SIGFOX peut envoyer entre **0 et 140 messages à 300bits/s** par jour et le payload de chaque message ne peut pas dépasser **12 octets.**

- Le protocole Sigfox est **bidirectionnel** sous condition : un objet Sigfox peut recevoir **4 messages par jour** à des instants définis.

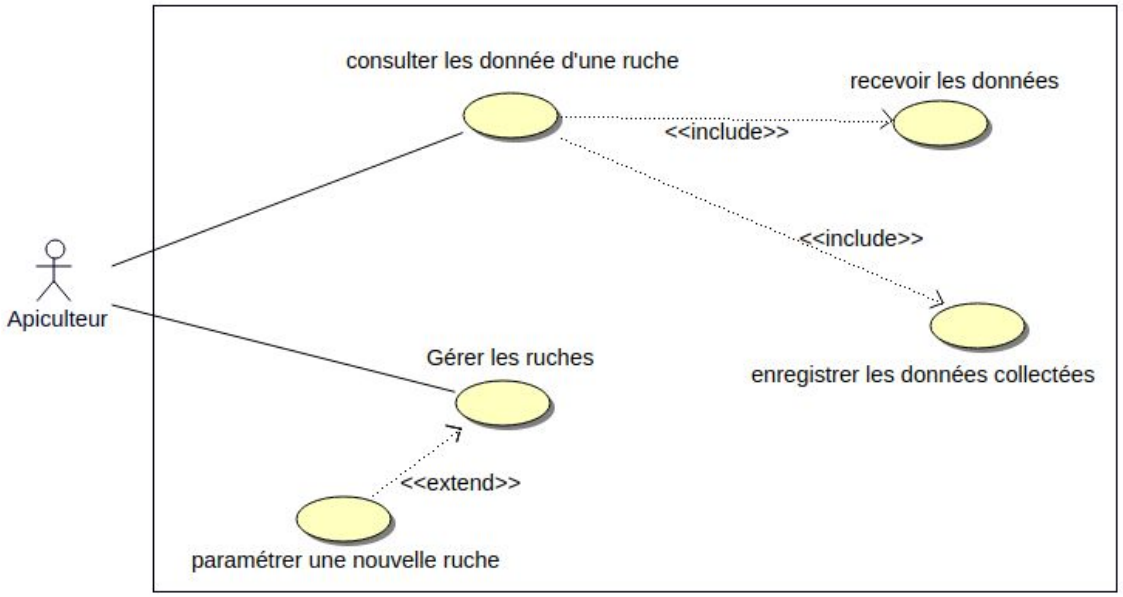
Fonctionnalités	LoraWan	SigFox
Modulation	SS chip	UNB / GFSK / BPSK
Bande Passante	500 - 125 KHz	100 Hz
Débit	290 bps - 50 Kbps	100 bit/sec
Message Max/jour	Illimité	140 / jour
Puissance Émission	20 dBm	151 dB
Durée de vie de la batterie	105 mois	90 mois
Sécurité	Oui	Non
Mobilité/Localisation	Oui	Mobilité réduite, pas de localisation

Partie Etudiant 3 : ROSSI Enzo

Objectifs

- Gérer les ruches : Paramétrer une nouvelle ruche
- Consulter les données d'une ruche (température, humidité, pression atmosphérique, et ensoleillement)
- Recevoir les données des ruches
- Enregistrer les données de (température, humidité, pression atmosphérique, et ensoleillement) dans une base de donnée

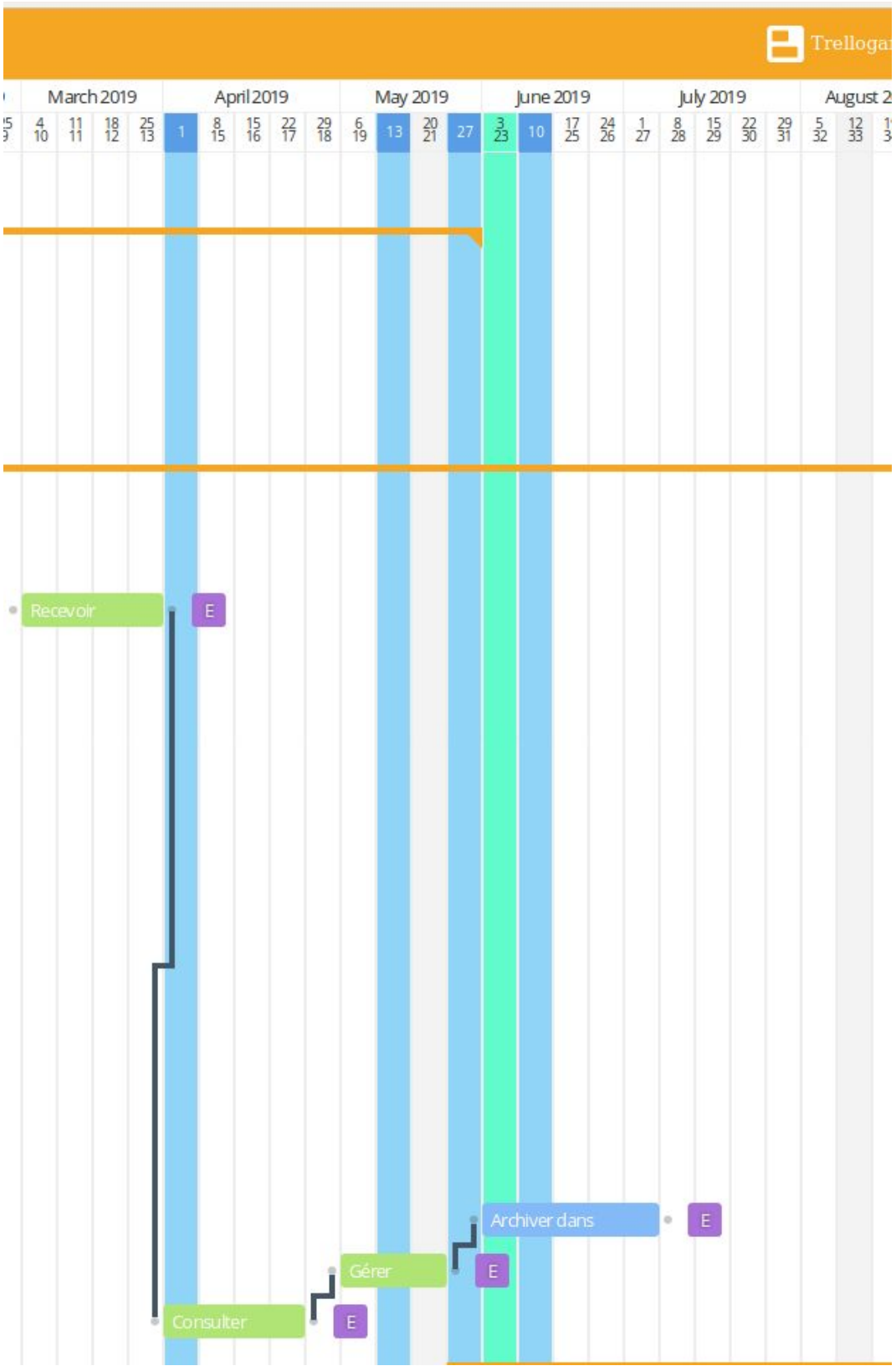
Diagramme de cas d'utilisation



Le diagramme de cas d'utilisation représente le point de vu de l'apiculteur et les différentes actions qu'il peut effectuer.

En effet l'apiculteur pourra paramétrer une nouvelle ruche il pourra aussi consulter les données environnement (température,humidité,pression atmosphérique,ensoleillement) des différentes ruche qui auront été paramétrées. De plus ces données pourront être stockées dans une base de données.

Planification



Planification Itérative

En respectant un développement itératif, on a choisi de répartir les différentes tâches suivant les itérations en tenant compte de critères de priorité :

Tâche	Priorité	Iteration
Recevoir les données des ruches	haute	1
Consulter les données d'une ruche (température, humidité, pression atmosphérique, et ensoleillement)	moyenne	1
Gérer les ruches : Paramétrer une nouvelle ruche	haute	2
Enregistrer les données dans la base de données	moyenne	3

Pour l'itération 1, la réception (et la consultation des données) pour une seule ruche a été retenue car la réception des données est une priorité absolue pour le besoin du client. La création de l'IHM principale en dépend aussi.

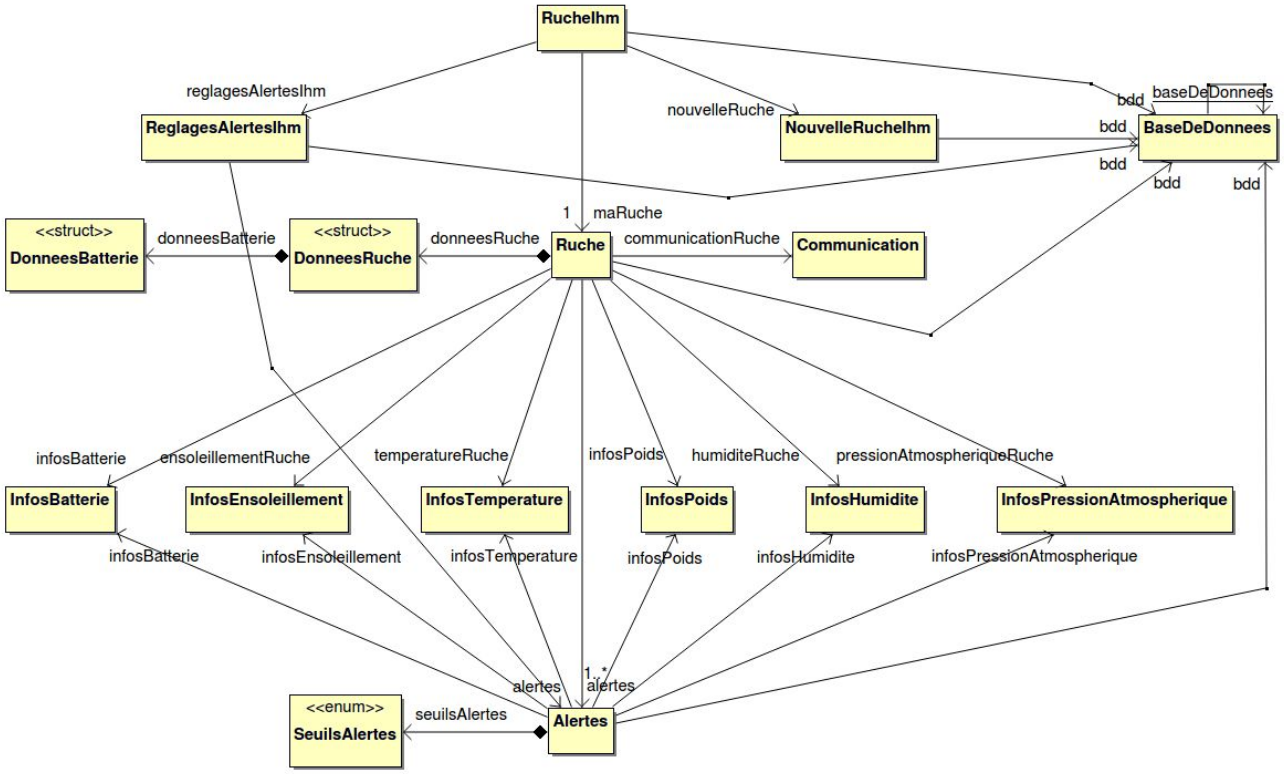
En effet le fait de connaître le fonctionnement du protocole de réception de données me permettra de mieux appréhender le concept de “multi ruches” pour l'itération 2 et donc d'assurer la création de nouvelles ruches.

Enfin une fois que toutes ces itérations seront réalisés, la troisième itération consistera à la conservation des données dans une base de données.

Ressources logicielles du projet

Désignation	Caractéristiques
Système d'exploitation du PC	GNU/Linux Ubuntu 16.04 LTS
Base de données	MySql 5.7
Logiciel de gestion de versions	subversion (RiouxSVN)
Générateurs de documentation	Doxygen version 1.8
Environnement de développement	Qt Creator et Qt Designer
API GUI	Qt 5.5.1

Diagramme de classes



Présentations des classes

Les classes sont organisées selon une architecture en trois couches :

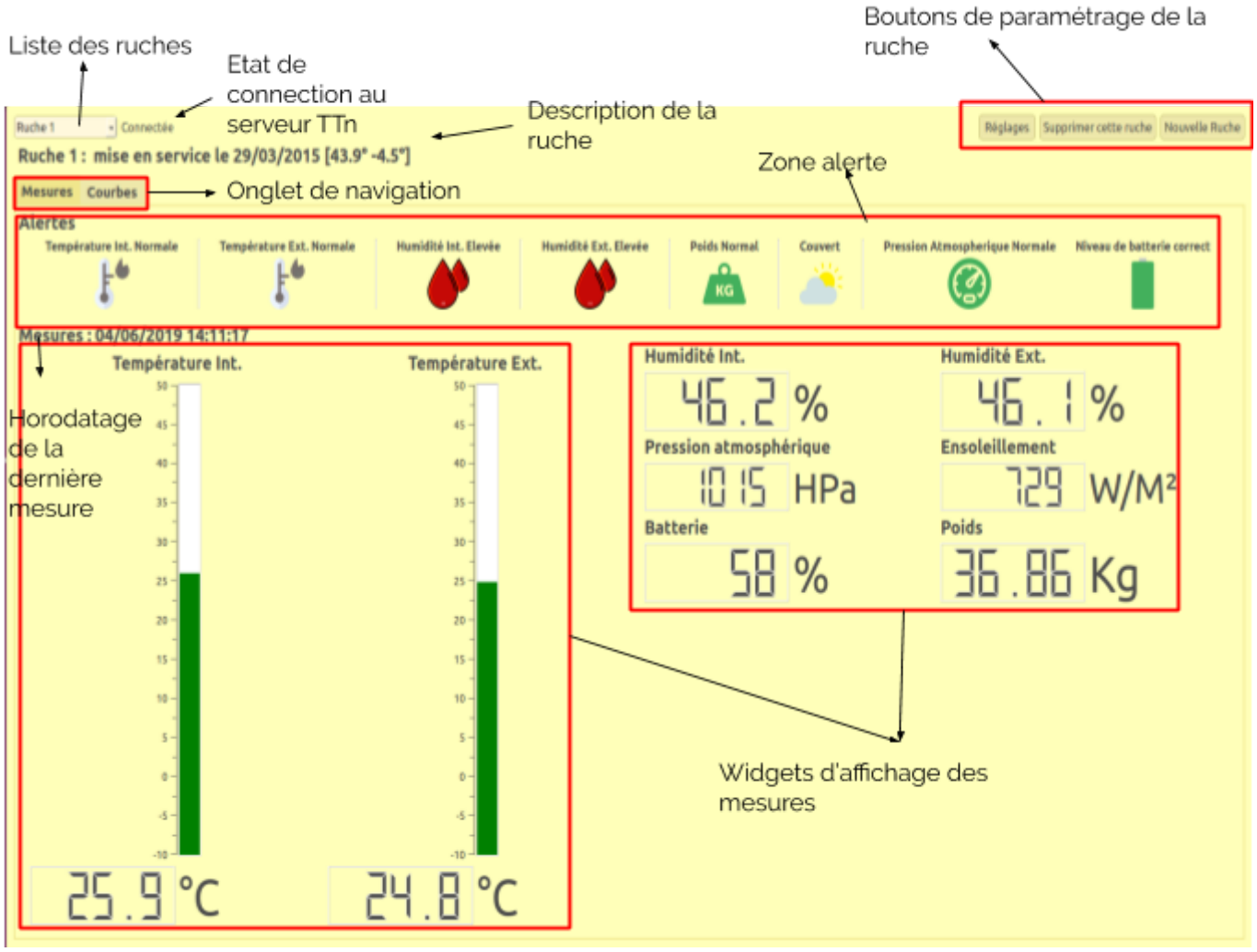
- couche effectuant le traitement : cette couche est confiée aux classes **information** qui traitent les mesures et effectuent des calculs. Le nommage de ces classes respecte ce modèle (**infosNomDeLaMesure**). On retrouve aussi la classe **BaseDeDonnees** (qui s'occupe de la gestion de la base de la base de données MySQL) et la classe **Communication** (qui s'occupe de la gestion du protocole MQTT pour la réception de données).

- couche effectuant le contrôle : cette couche est assurée par une classe contrôleur (la classe **Ruche**). Son but est de servir d'intermédiaire entre les différentes autres classes.

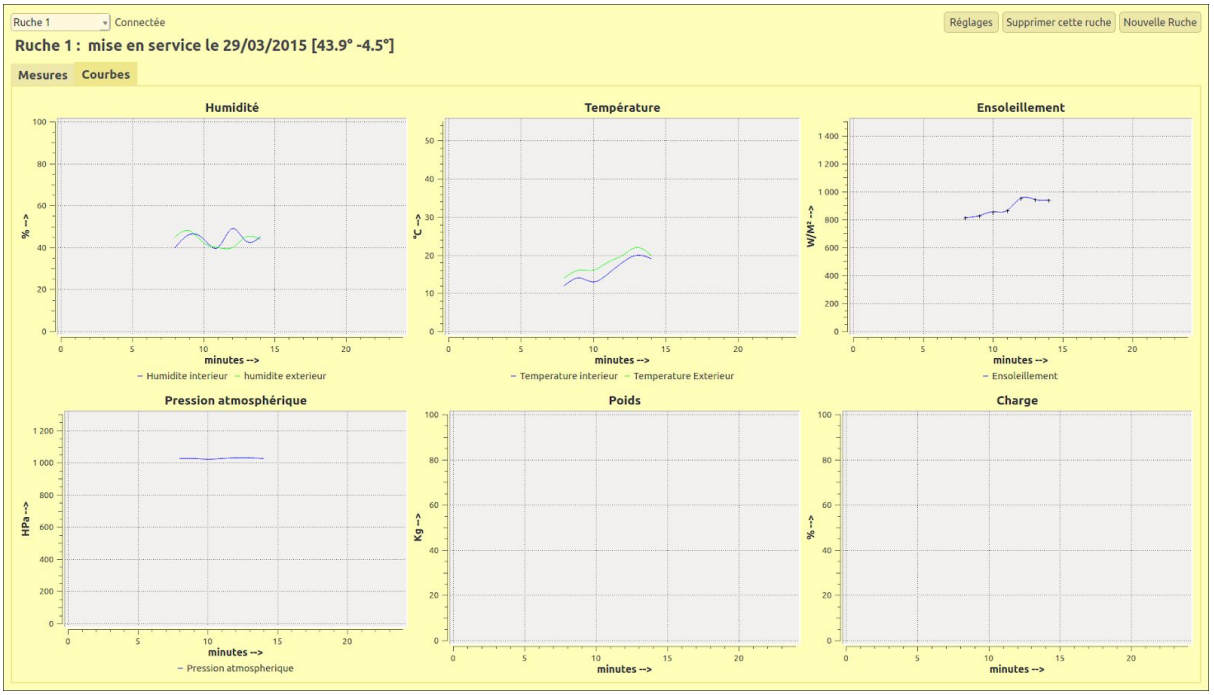
- couche effectuant l'interface homme machine : cette couche est dédiée à l'IHM (la classe RucheIhm et NouvelleRucheIhm). Le but est de fournir une interface utilisateur et de recueillir les données de l'utilisateur pour les émettre au contrôleur mais aussi de recueillir les données venant du contrôleur et de les afficher à l'aide de widgets graphiques.

Le choix de ce type d'architecture a été choisi pour permettre une séparation entre classes tout en permettant la centralisation et la gestion des données dans le contrôleur. En effet le fait de pouvoir centraliser ces informations permet ensuite de traiter les données plus simplement.

L'IHM principale



On peut voir sur l'Ihm principale les mesures de la ruche sélectionnée. Cette Ihm possède aussi un onglet "Courbes" :



On peut voir les courbes de l'ensoleillement, la pression atmosphérique la température intérieur, extérieur et l'humidité intérieur, extérieur correspondant aux valeur moyennes recueillies en fonction des heures (les données ont été recueillies sur la plage horaire 8h-14h).

Décodage des trames sur le serveur The Things Network

Les mesures reçues sur le serveur sont codées en base64. Il y a deux solutions pour assurer le traitement de celles ci.

première solution : traiter la données reçues directement dans chaque programme grâce avec une fonction de décodage pour reconstituer la mesure.

deuxième solution (adoptée) : décoder directement les données grâce à une fonction écrite et exécutée sur le serveur TTN.

Exemple de fonction de décodage :

```
function Decoder(bytes, port) {
  var decoded = {};

  switch (port)
  {
    case 1: // Batterie
      decoded.tension = ((bytes[0] << 8) + bytes[1]) * (5/1024);
      decoded.courant = ((bytes[2] << 8) + bytes[3]) * (5/1024);
      decoded.charge = (bytes[4]);
      break;
    case 2: //poids
      decoded.poids = ((bytes[0] << 8))
      break;
    case 3: // DHT22
      decoded.temperature = ((bytes[0] << 8) + bytes[1])/100;
      decoded.humidite = ((bytes[2] << 8) + bytes[3])/100;
      break;
    case 4: // DHT22 exterieur
      decoded.temperature = ((bytes[0] << 8) + bytes[1])/100;
      decoded.humidite = ((bytes[2] << 8) + bytes[3])/100;
      decoded.pression = (bytes[4] << 8);
      break;
    case 5: //ensoleillement
      decoded.ensoleillement = (bytes[0] << 8);
      break;
  }
  return decoded;
}
```

Format d'échange de données Json

Un document **JSON** ne comprend que deux types d'éléments structurels :

- des ensembles de paires « nom » (alias « clé ») / « valeur » : "*id*": "*file*"
- des listes de valeurs séparées par des virgules : "*value*": "*New*", "*onclick*": "*CreateNewDoc()*".

Ces mêmes éléments représentent trois types de données :

- des objets : { ... } ;
- des tableaux : [...] ;
- des valeurs génériques de type tableau, objet, booléen, nombre, chaîne de caractères ou *null* (valeur vide).

Dans le cadre du projet les données en **Json** s'organise comme ceci :

```
"{
   "app_id": "mes_ruches",
   "dev_id": "ruche_1",
   "hardware_serial": "0004A30B00203CF8",
   "port": 3,
   "counter": 16138,
   "payload_raw": "CCoPFA==",
   "payload_fields":
   {
      "humidite":38.6,
      "temperature":20.9
   },
   "metadata":
   {
      "time":"2019-04-04T12:16:07.226020821Z",
      "frequency":868.5,
      "modulation":"LORA",
      "data_rate":"SF7BW125",
      "airtime":51456000,
      "coding_rate":"4/5",
      "gateways":[
      {
         "gtw_id\": "btssn-lasalle-84",
```

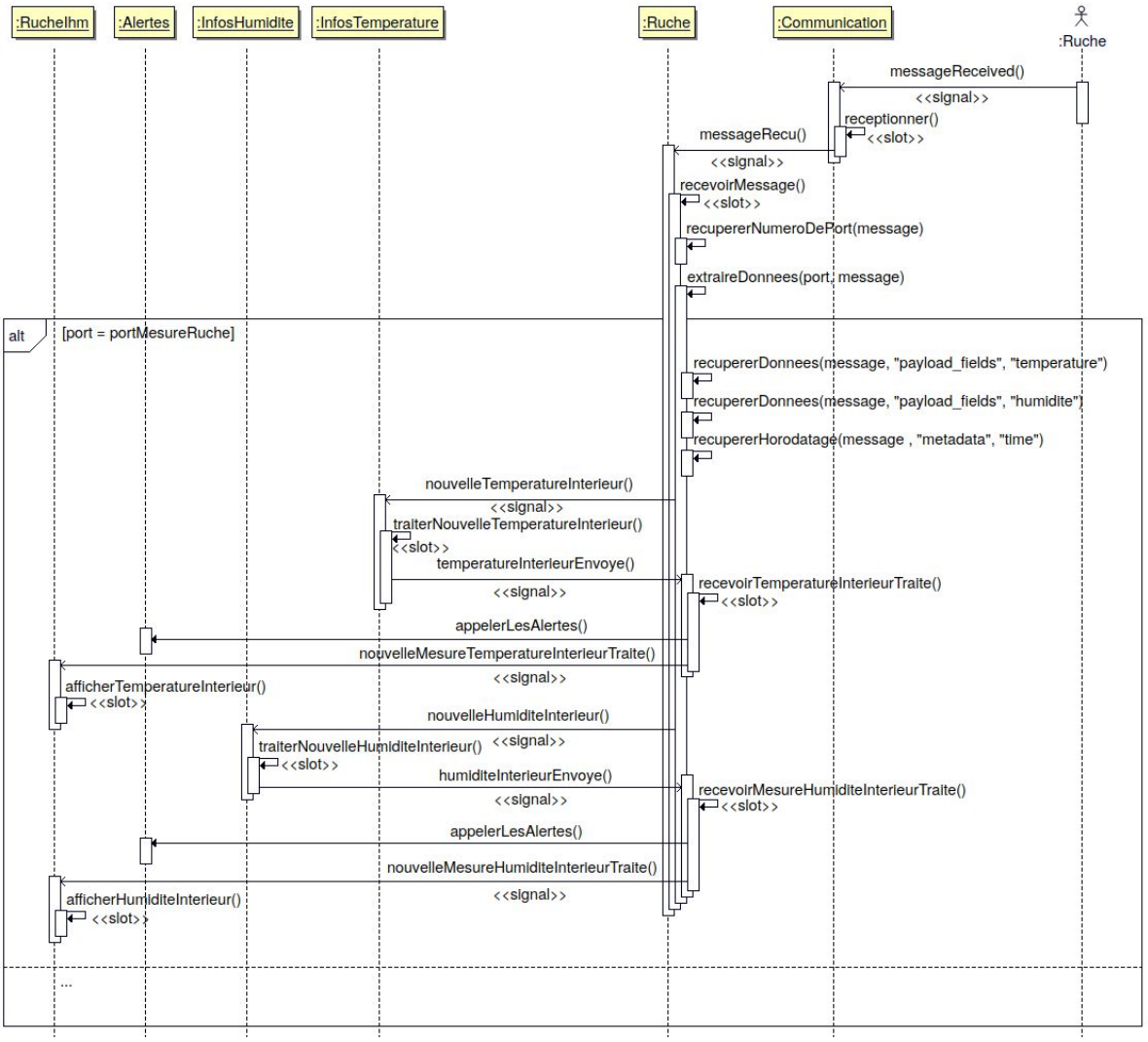
```
        "gtw_trusted":true,
        "timestamp":1360002747,
        "time\":"2019-04-04T12:16:07Z",
        "channel":2,
        "rssi":-45,
        "snr":7.75,
        "rf_chain":1,
        "latitude":43.948326,
        "longitude":4.8169594,
        "location_source":"registry"
      }]
   }
}"
```

Pour pouvoir extraire ces données, Qt propose un module QJson que l'on va utiliser pour récupérer les données envoyées par la ruche.

Extraction et gestion des données

Pour pouvoir extraire les données il faut tout d'abord déterminer le port TTN à laquelle la mesure est associée, comme on peut voir dans ce tableau :

Fport	Nom	Mesures	IR
1	portMesureEnergie	tension,courant,charge	MELLAH F.
2	portMesurePoids	poids	MELLAH F.
3	**portMesureRuche**	**température,humidité (intérieur)**	**Enzo ROSSI**
4	**portMesureEnvironement**	**température, humidité, pression atmosphérique(extérieur)**	**Enzo ROSSI**
5	**portMesureEnsoleillement**	**ensoleillement**	**Enzo ROSSI**



Voici la liste des différentes étapes pour l'affichage des mesures.

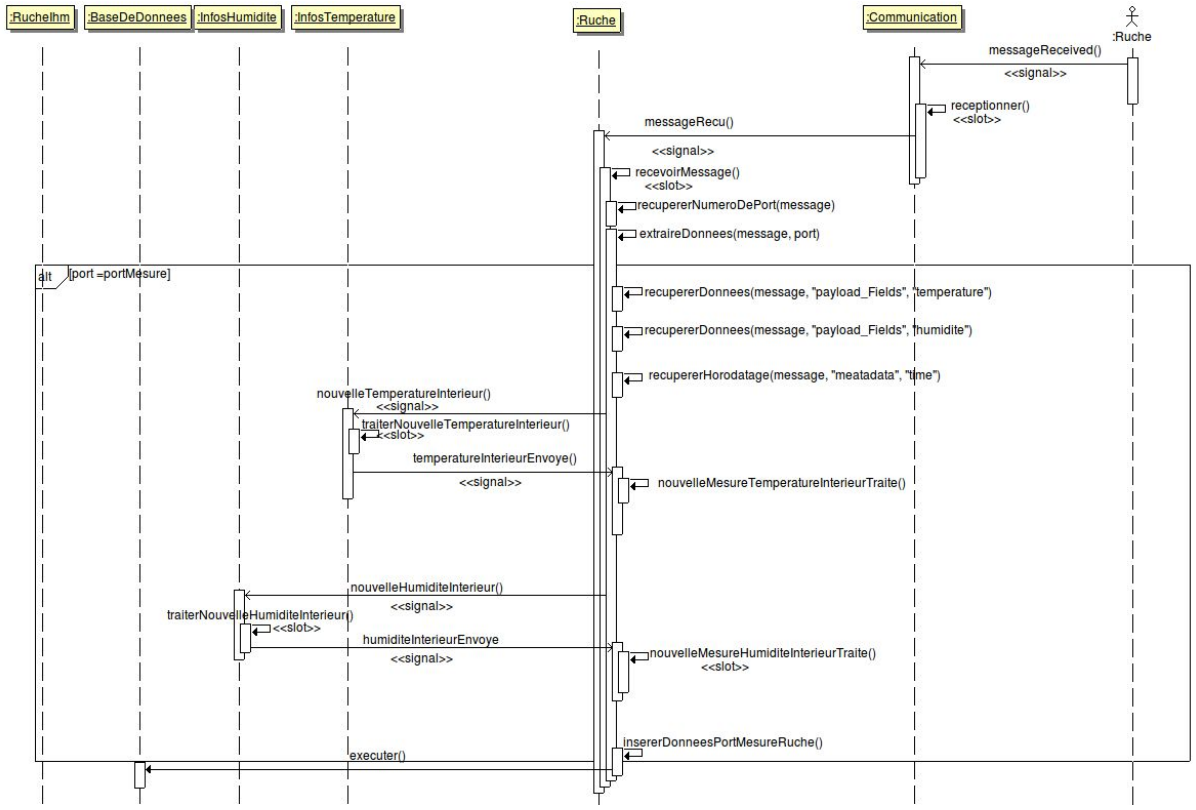
- Donc la première étape est d'extraire le numéro de port dans le fichier Json. (classe **Ruche**)
- Une fois que le numéro de port est extrait, on peut extraire la mesure qu'on souhaite en fonction du port correspondant (Classe **Ruche**)
- Une fois la donnée extraite elle est envoyée à la classe infos concernée
- La mesure est traitée dans la classe et celle-ci traitée est envoyée à la classe **Ruche**
- La classe Ruche reçoit la donnée traitée et l'envoie à la classe **RucheIhm**
- La Classe **RucheIhm** affiche la donnée traitée à l'aide de widget graphique.

Insertion des données dans la base de données

Pour enregistrer des données de température et d'humidité on effectue la commande suivante en langage SQL :

```
INSERT INTO MesuresRuche (idRuche, Temperature, Humidite, Horodatage)
VALUES (...)
```

Réception et enregistrement des mesures intérieures

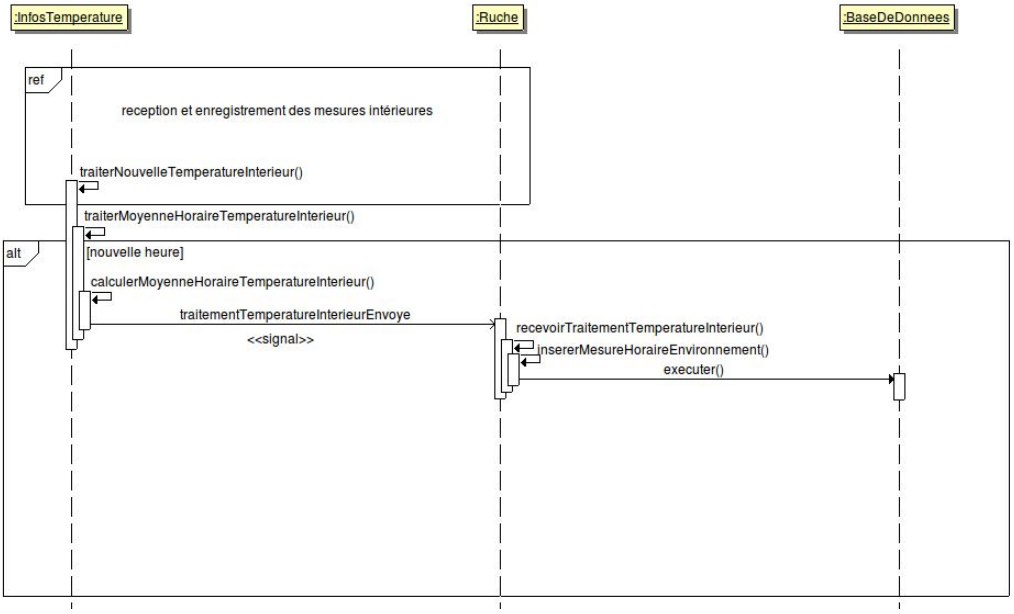


Voici la liste des etapes pour l'enregistrement des mesures dans la base de données :

- On émet les données reçu vers les classes **infos**
- les classes **info** traitent les données.
- les données sont ensuite émis à la classe **ruche**
- On les regroupes on les enregistre dans la base de données avec la requête précédemment énoncé

Enregistrement des données journalières

L'application effectue un enregistrement chaque heure des moyennes, minimums et maximums des mesures reçues. Cette fonctionnalité se retranscrit dans le diagramme de séquence suivant :



- Tout d'abord on récupère les donnée à partir de la méthode traiter NouvelleTempératureInterieur() de la classe **infosTemperature**.
- de cette méthode on traite et calcule les moyenne horaires.
- Si on passe à l'heure suivante alors un signal est émis à la classe Ruche.
- la classe Ruche reçoit la valeur et l'insère la base de données.

L'insertion dans la base de données s'effectue par la requête suivante dans la table journalière concernée :

```
INSERT INTO MesuresJournalieresRuche (MesuresJournalieresRuche.idRuche,
MesuresJournalieresRuche.Temperature,
MesuresJournalieresRuche.TemperatureMin,
MesuresJournalieresRuche.TemperatureMax,
MesuresJournalieresRuche.Humidite, MesuresJournalieresRuche.HumiditeMin,
MesuresJournalieresRuche.HumiditeMax,
MesuresJournalieresRuche.DateMesure,
MesuresJournalieresRuche.HeureMesure) VALUES (...)
```

Affichage des courbes

L'affichage des courbes s'effectue à l'aide d'une librairie fournissant des widgets graphique nommé **qwt**.

Les repères sont définis comme ceci : le repère représentant les mesures de températures ainsi que le repère de l'humidité regrouperont les données intérieures et extérieures.

Les axes sont définis comme ceci :

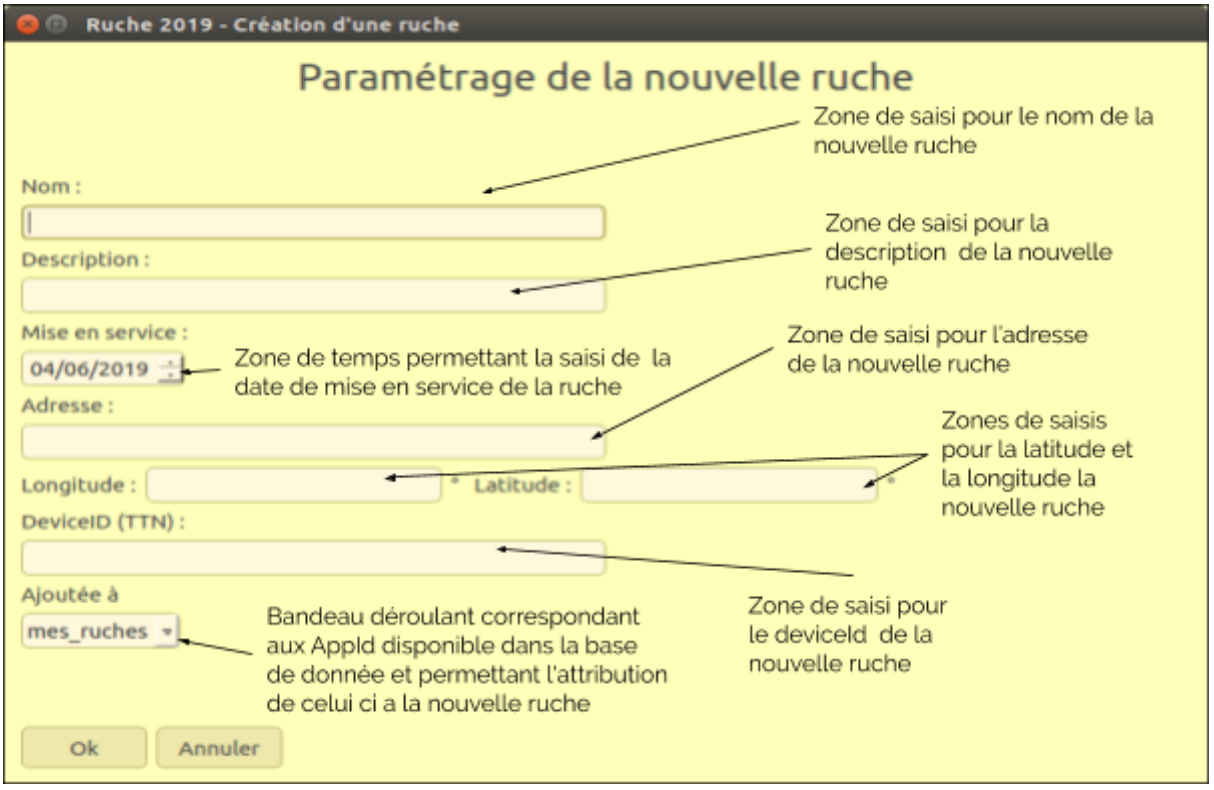
- l'axe x en heures
- l'axe y dans l'unité de la mesure affichée

Pour afficher ces valeurs on récupère celles ci dans les tables MesuresJournalieres dans la base de données avec la requête SQL :

```
SELECT Temperature, Humidite, HeureMesure FROM MesuresJournaliersRuche
WHERE DateMesure = '" + dateCourante.toString("yyyy-MM-dd") + "' AND
idRuche = '" + mesRuches[positionDeLaRuche].at(0) + "' ORDER BY
HeureMesure ASC";
```

Gestion et paramétrage de nouvelle ruche

Le paramétrage d'une nouvelle ruche s'effectue par l'interface homme/machine (classe **NouvelleRucheIhm**).



Pour avoir la liste des AppID disponibles on les récupère dans la base de données avec la requête suivante :

```
SELECT idTTN, ApplicationID FROM TTN
```

Les données recueillies sont ensuite enregistrées dans la table Ruche par la requête SQL suivante :

```
INSERT INTO Ruche INSERT (idTTN, Nom, Description, DateMiseEnService,
Adresse, Longitude, Latitude, DeviceID) VALUES (...)
```

De plus on ajoute les seuils par défaut dans la table Seuils avec la requête suivante :

```
INSERT INTO Seuils (idRuche, TemperatureIntMin, TemperatureIntMax,
HumiditeIntMin, HumiditeIntMax, TemperatureExtMin, TemperatureExtMax,
HumiditeExtMin, HumiditeExtMax, PressionMin, PressionMax, PoidsMin,
PoidsMax, EnsoleillementMin, EnsoleillementMax, Charge) VALUES (...)
```

Donc chaque ruche sera enregistrée dans la base de données.

Pour récupérer les ruches dans la base de données on effectue la requête sql suivante :

```
SELECT Ruche.idRuche, Ruche.Nom, Ruche.DeviceID, TTN.idTTN,
TTN.Hostname, TTN.Port, TTN.Username, TTN.Password, TTN.ApplicationID,
Ruche.Adresse, Ruche.DateMiseEnService, Ruche.Longitude, Ruche.Latitude
FROM Ruche INNER JOIN TTN ON Ruche.idTTN = TTN.idTTN;
```

Tests de validation

Désignation	**Résultat attendu**	**Oui / Non**	**Remarques**
Recevoir les données des ruches	Recevoir les données des ruches	Oui	
Consulter les données d'une ruche (température, humidité, pression atmosphérique, et ensoleillement)	Visualiser les mesures avec leur unité sur l'IHM	Oui	
Gérer les ruches : Paramétrer une nouvelle ruche	Créer ou supprimer une ruche	Oui	
enregistrer les données de (température, humidité, pression atmosphérique, et ensoleillement)	Enregistrer les données dans la base de données	Oui	

Partie physique

Lora

LoRaWAN est un protocole de télécommunication permettant la communication à bas débit, par radio, d'objets à faible consommation électrique communiquant selon la technologie LoRa et connectés à l'Internet via des passerelles, participant ainsi à l'Internet des objets. Ce protocole est utilisé dans le cadre des villes intelligentes, le monitoring industriel ou encore l'agriculture. La technologie de modulation liée à LoRaWAN est LoRa, née à la suite de l'acquisition de la startup grenobloise Cycléo par Semtech en 2012. Semtech promeut sa plateforme LoRa grâce à la LoRa Alliance, dont elle fait partie. Le protocole LoRaWAN sur la couche physique LoRa permet de connecter des capteurs ou des objets nécessitant une longue autonomie de batterie (comptée en années), dans un volume (taille d'une boîte d'allumettes ou d'un paquet de cigarettes) et un coût réduits.
LoRaWAN est l'acronyme de Long Range Wide-area network que l'on peut traduire par « réseau étendu à longue portée ».

Voici un comparatif avec la technologie sigfox, qui est lui aussi un protocole de communication bas débit

Même famille LPWA (Low Power, Wide Area)
Réseaux dédiés spécifiquement aux applications IoT à longue portée et faible consommation consommation

	sigfox	VS	LoRa
Type de réseau :	Ouvert (open source)		Propriétaire
Origine :	Start-up Française toulousaine		Technologie et Protocole développés par SMETECH (société Américaine) à partir d'une technologie Française (Cycleo)
Portée urbaine :	3 à 10 kms		3 à 8 kms
Portée rurale :	30 à 50 kms		15 à 20kms
Modulation :	SS chip Ultra Narrow Band (UNB)		UNB/GFSK/BOSK
Abonnement par appareil :	1 et 15€ /an		1 et 15€ /an
Bidirectionnel :	Oui		Oui
Messages max :	12 octets		242 octets
Messages par jour :	140		Illimité
Débit :	100bits/s		0,3 à 50kbits/s
Couverture :	93% de la population française (supporté par Bouygues et Orange)		94% de la population française (sup-portée par SFR)
Sécurité :	Appareil et station protégés par id unique		Encryptage AES128
Immunité aux bruits :	Basse		Très haute
Autonomie batterie 2000mAh :	90 mois		105 mois

voici un autre comparatif avec d'autre protocole de communication sans fil

	Courte portée			Moyenne portée			Longue portée	
Technologie	**NFC**	**Bluetooth**	**Zigbee**	**Z-Wave**	**Wi-Fi**	**BLE**	**SigFox**	**LoRa**
Portée moyenne (en intérieur)	<10 cm	10 m	10 m	50 m	50 m	50 m	>2km	>2km
Débit (Mbit/s)	1.10^{-3}	1.10^{-3}	1.10^{-2}	1.10^{-2}	1.10^{2}	1.10^{-3}	1.10^{-3}	1.10^{-3}
Autonomie	Mois	Jours	Années	Années	Jours	Mois	Années	Années
Fréquence	2,4 GHz	2,4 GHz	2,4 GHz 868 MHz	868 MHz	2,4 GHz 5 GHz	2,4 GHz	868 MHz	868 MHz
Usages	Téléphonie Cartes de paiement	Périphériques informatiques et multimédia	Domotique		Navigation Internet Transferts conséquents de données	Périphériques informatiques et multimédia	Prévention d'incidents Collecte de données Gestion de réseaux	

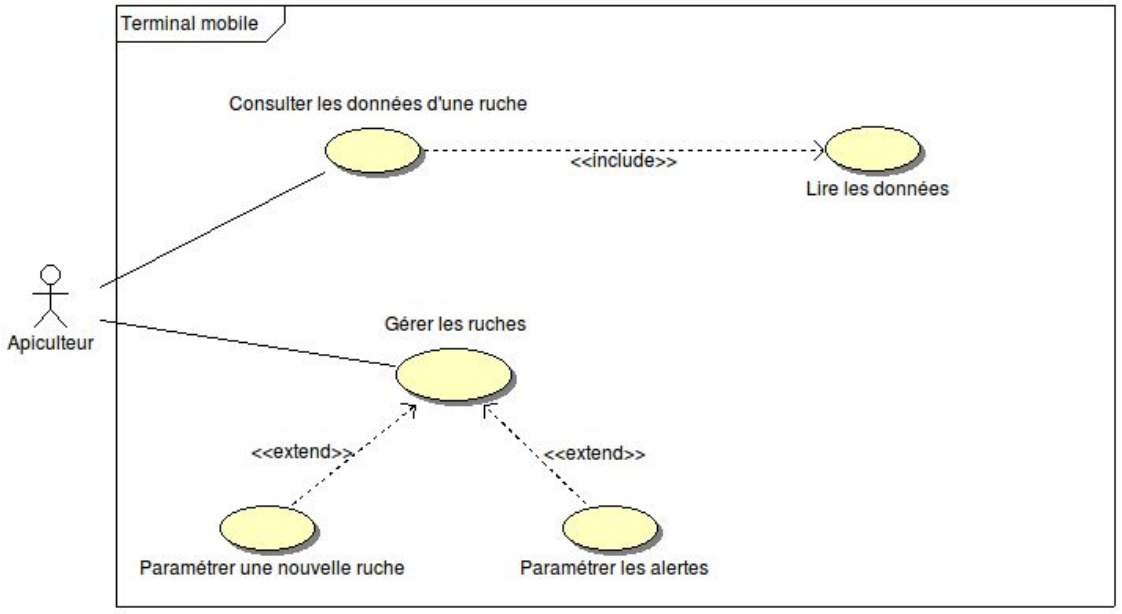
On peut donc voir que Lora est protocole approprié pour le cas d'une ruche connectée.

Partie Etudiant 5 (terminal mobile) : LAURAIN Clément

Objectifs

- Gérer les ruches
- Consulter les données d'une ruche
- Lire les données à partir de la base de données

Diagramme Cas d'utilisation



Le diagramme des cas d'utilisation permet d'avoir une vue d'ensemble des fonctionnalitées du point de vue de l'utilisateur autrement dit l'apiculteur.

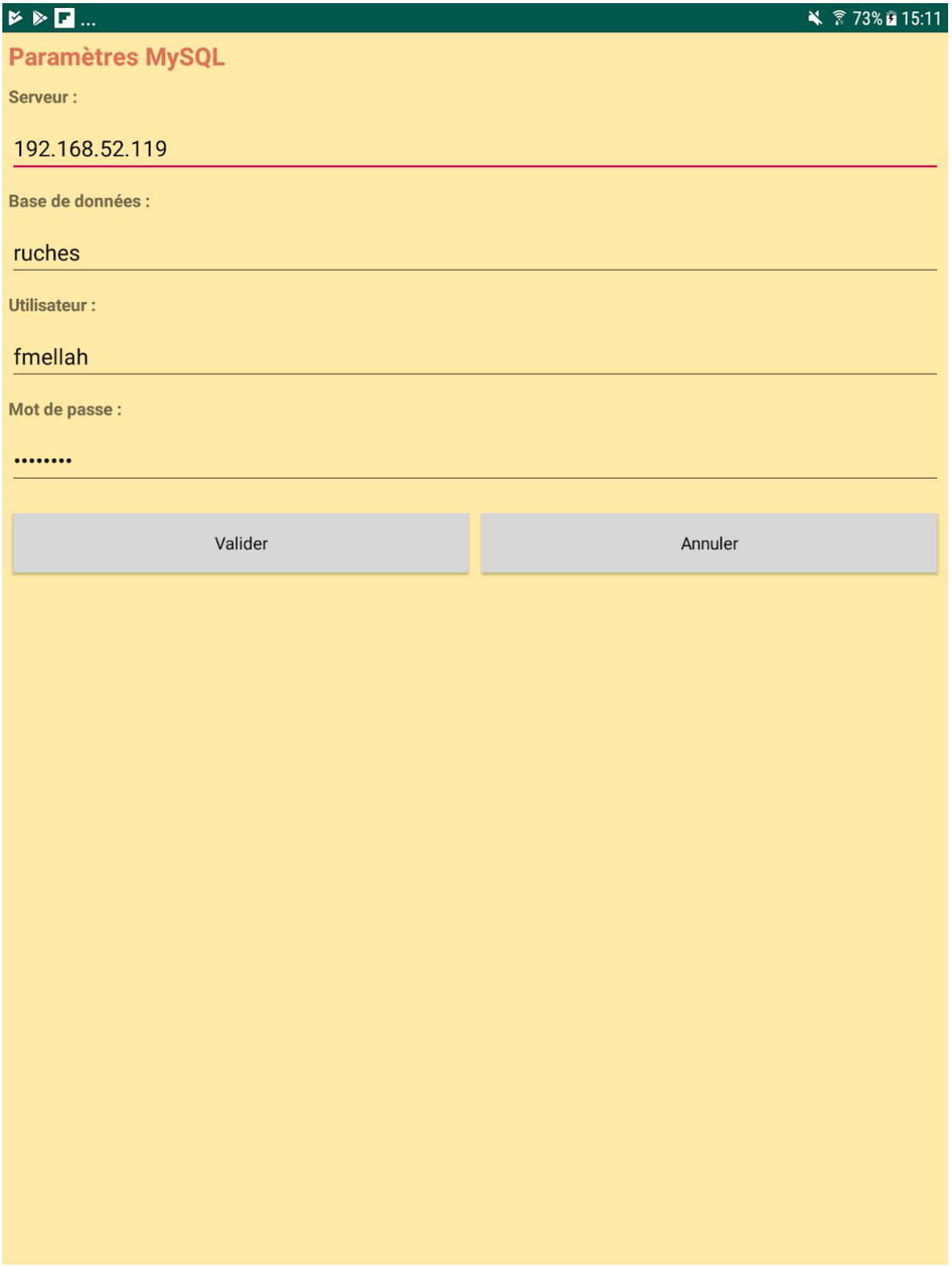
L'apiculteur doit pouvoir consulter les données essentiels à la survie de la ruche, tout ceci à distance depuis une interface mobile. Il peut gérer ses ruches en **paramétrant une nouvelle ruche** ou / et **paramétrer de nouvelles alertes** pour différentes mesures afin de pouvoir agir le plus rapidement possible.

Le terminal mobile représente tout appareil fonctionnant sur le système d'exploitation

Android, dans le cas du projet une tablette Galaxy Tab S2.

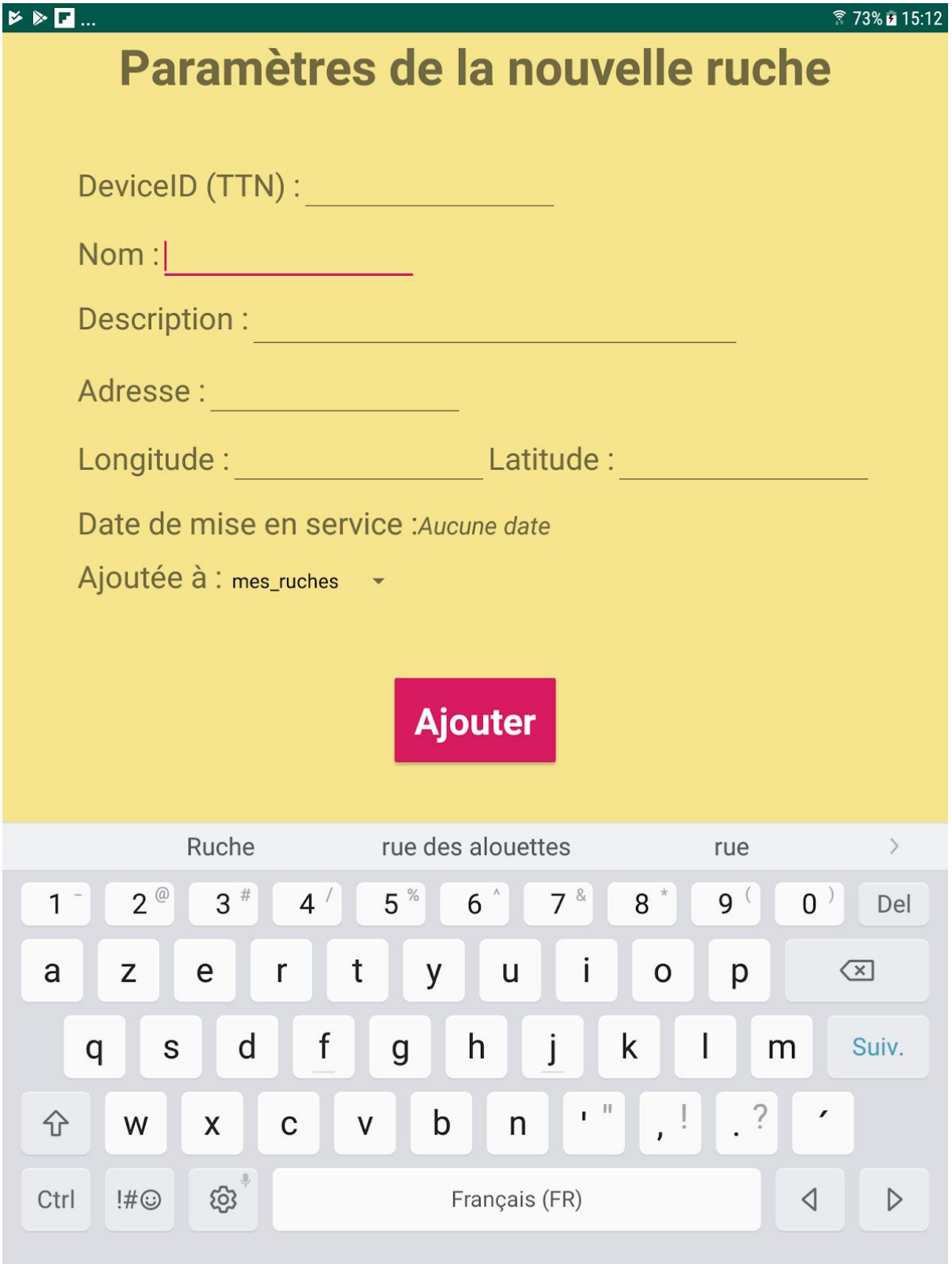
Maquette Interface Graphique

Maquette de l'activité d'accueil



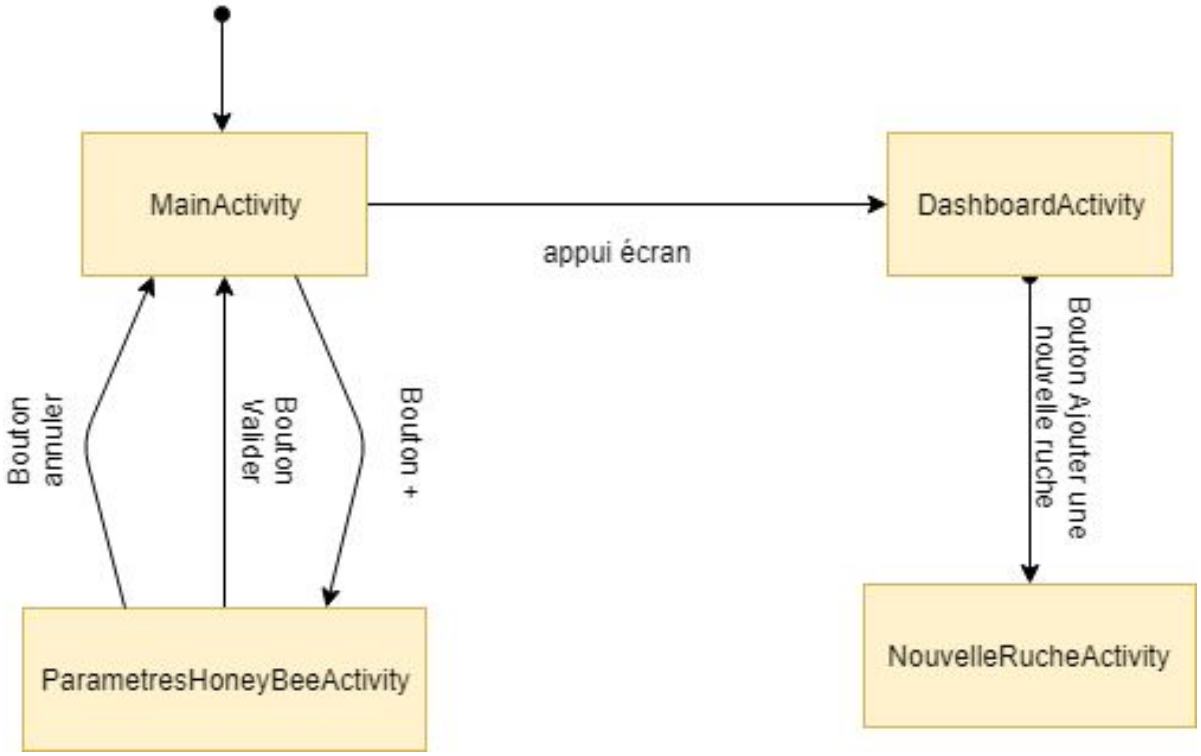
Maquette de l'activité de paramétrages de la base de données

Maquette de l'activité tableau de bord

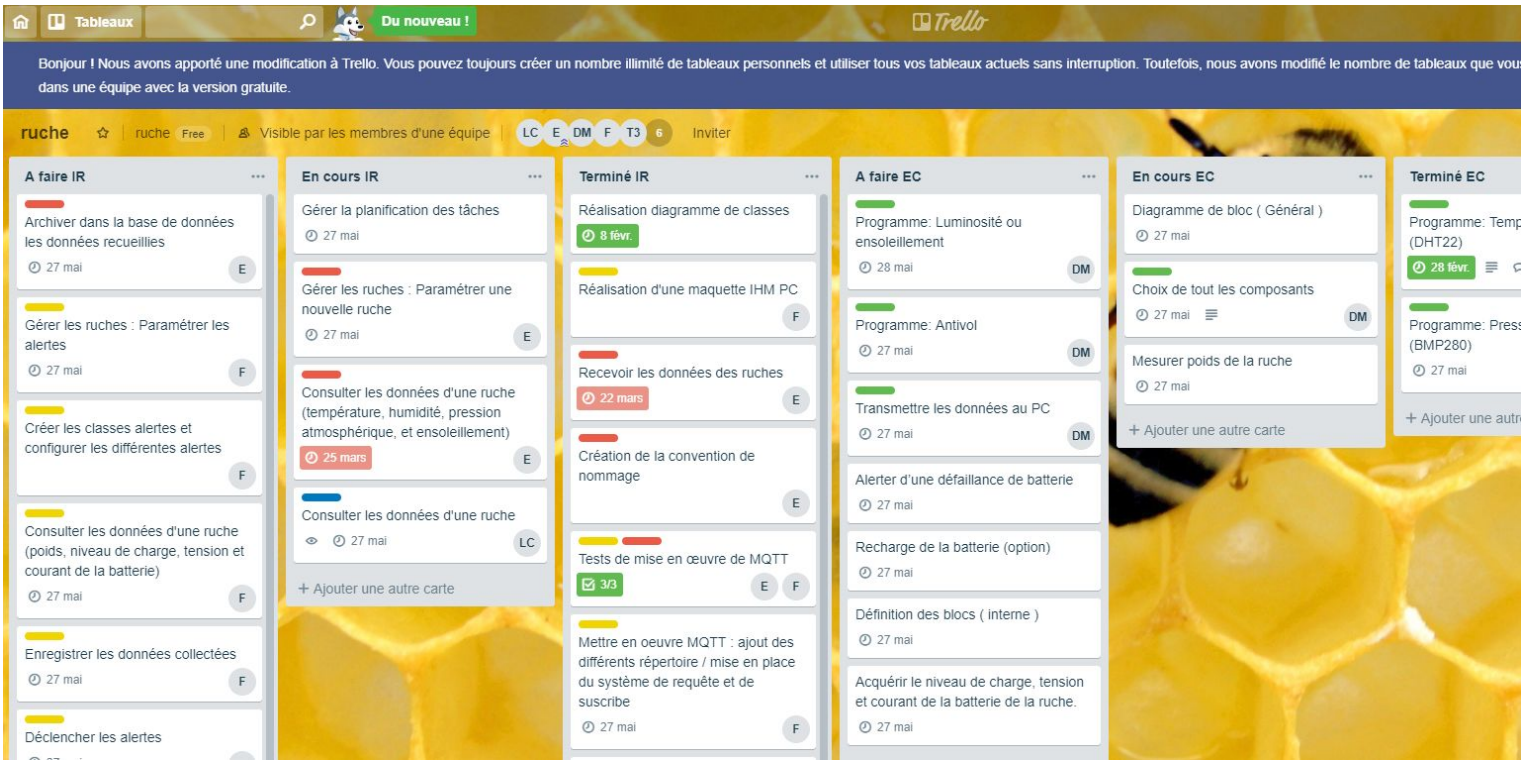


Maquette de l'activité de l'ajout d'une ruche

Diagramme de navigation



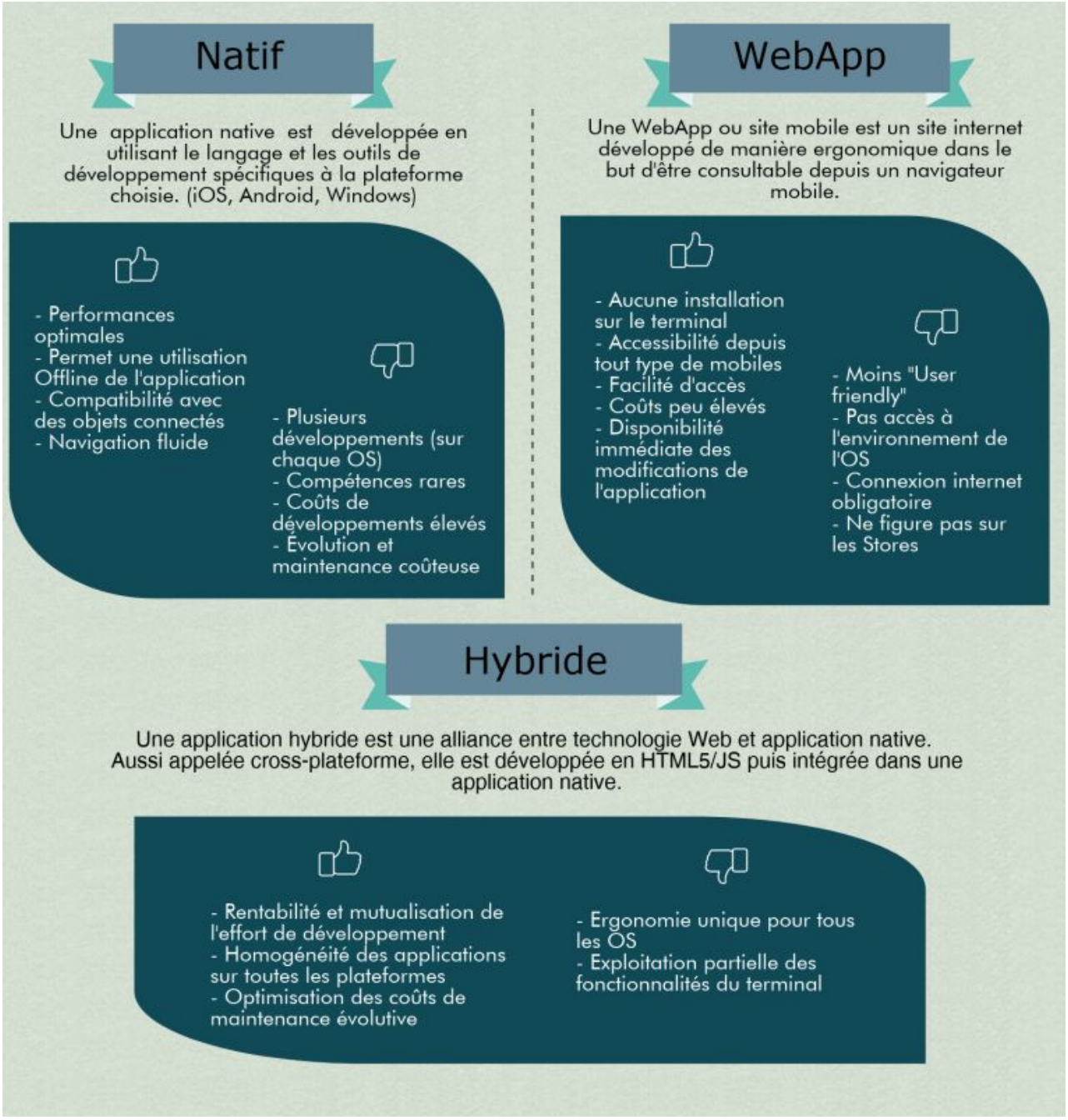
Planification des tâches



Répartition des Tâches		
ROSSI Enzo	LAURAIN Clement	MELLAH Florentin
❑ Gérer les ruches : Paramétrer une nouvelle ❑ Consulter les données d'une ruche (température, humidité, pression atmosphérique et ensoleillement) ❑ Recevoir les données des ruches	❑ Gérer les ruches ❑ Consulter les données d'une ruche ❑ Lire les données à partir de la base de données.	❑ Gérer les ruches : Paramétrer les alertes ❑ Consulter les données d'une ruche (poids, niveau de charge, tension et courant de la batterie) ❑ Enregistrer les données collectées ❑ Déclencher les alertes

Ressources nécessaires au développement

Trois catégories séparent le développement mobile, le développement mobile dit natif, le développement hybride, et enfin le développement dit webapp.

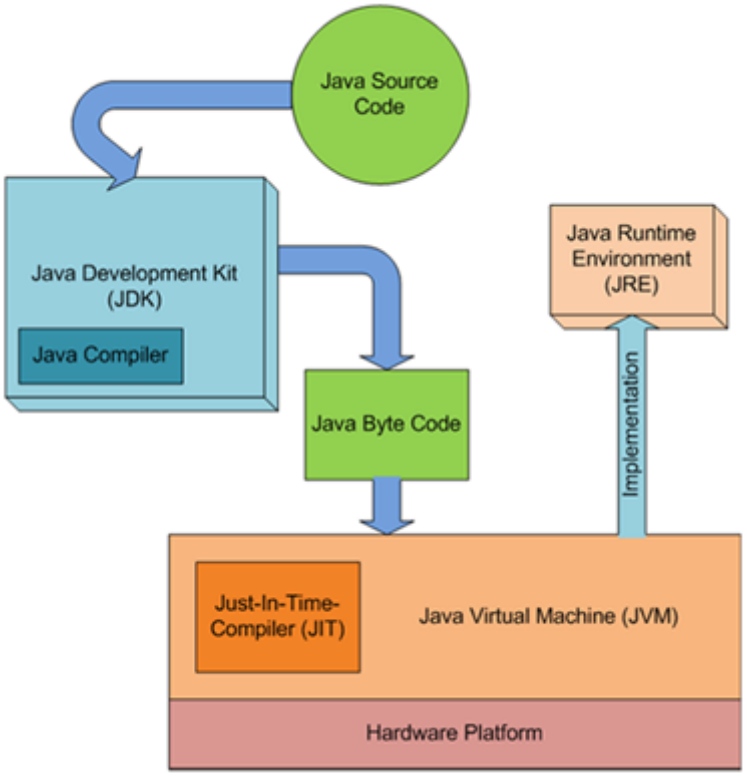


Après comparatif le choix c'est porté sur le développement dit Natif pour plusieurs raisons. Le développement WebApp est déjà une compétence dans laquelle j'ai déjà eu de la pratique, j'ai donc hésité entre Hybride et Natif et je me suis donc finalement

orienté sur Natif car j'avais besoin pour la partie réception de données d'une application performante et ce serait l'opportunité pour moi de m'initier au **Java** par l'intermédiaire de ce projet.

Une tablette Samsung Galaxy Tab S2 nous a été fourni pour le projet, la tablette fonctionnant donc sur le système d'exploitation **Android** il m'a donc fallu m'initier au langage Natif de cette plateforme à savoir le langage Java.

Afin de développer en Java l'utilisation de certains outils sont nécessaires tels que : Java Runtime Environment, Java Development Kit.



Le Java Development Kit (JDK) est composé d'un ensemble de bibliothèques pour commencer à coder en Java ainsi qu'un compilateur appelé **Javac**. Le compilateur s'occupera donc de compiler le programme et de le transformer en fichier écrit en **Java Byte Code** et non en binaire au contraire du langage C++ par exemple. Cette différence lors de la compilation permet donc par l'intermédiaire de cette génération d'un fichier **Java Byte Code** comprenant des **instructions machines** universelle, permet ensuite la retranscription ensuite en instructions machine spécifiques (en binaire donc) à l'architecture du processeur de l'appareil sur lequel l'application va s'exécuter. La traduction du **Java Byte Code** en code machine se fait par l'intermédiaire du Java Runtime Environment (JRE) qui est donc compris dans le Java Development Kit, en

pratique le JRE va créer une machine virtuelle Java appelé Java Virtual Machine (JVM) afin d'exécuter l'application et compiler le **Java Byte Code** en **temps réel** en instruction machine grâce au Just-In-Time-Compiler (JIT).

La partie JDK est installé sur la machine servant au développement et le JRE sur la machine exécutant le programme (cette machine peut être la même, ou dans notre cas deux, un ordinateur pour le développement et une tablette pour l'exécution).

Une petite spécificité cependant l'intégration d'un Android Native Development Kit (NDK) est nécessaire quand l'application est prévu pour une utilisation sur **Android** car le NDK nous apporte la possibilité de gérer les ressources de l'appareil fonctionnant sous **Android** exécutant l'application.

Le choix de l'environnement de développement intégré Android Studio a été fait afin de faciliter le développement. En effet lors de l'installation de celui ci tous les outils nommés précédemment sont inclus dans celle ci, nous apporte l'apport un **module de simulation** d'appareil fonctionnant sur la version **d'Android** souhaité et nous apporte la **correction syntaxique**, et un **assistant designer** pour aider à la création de l'interface,

J'ai donc utilisé en final Android Studio, le JRE et JDK dans leurs versions 1.8.0. La tablette fonctionne sous le système d'exploitation Android dans sa version 7 (Nougat API 24).

Désignation	**Caractéristiques**
Terminal mobile	Samsung Galaxy Tab S2
Environnement de développement	Android Studio
API	API 24
Système d'exploitation du terminal mobile	Android 7.0
Interface binaire-programme	arm64-v8a
Logiciel de gestion de versions	Subversion (RiouxSVN)
Système de gestion de bases de données relationnelles	MySQL
Atelier de génie logiciel	Bouml version 7.8

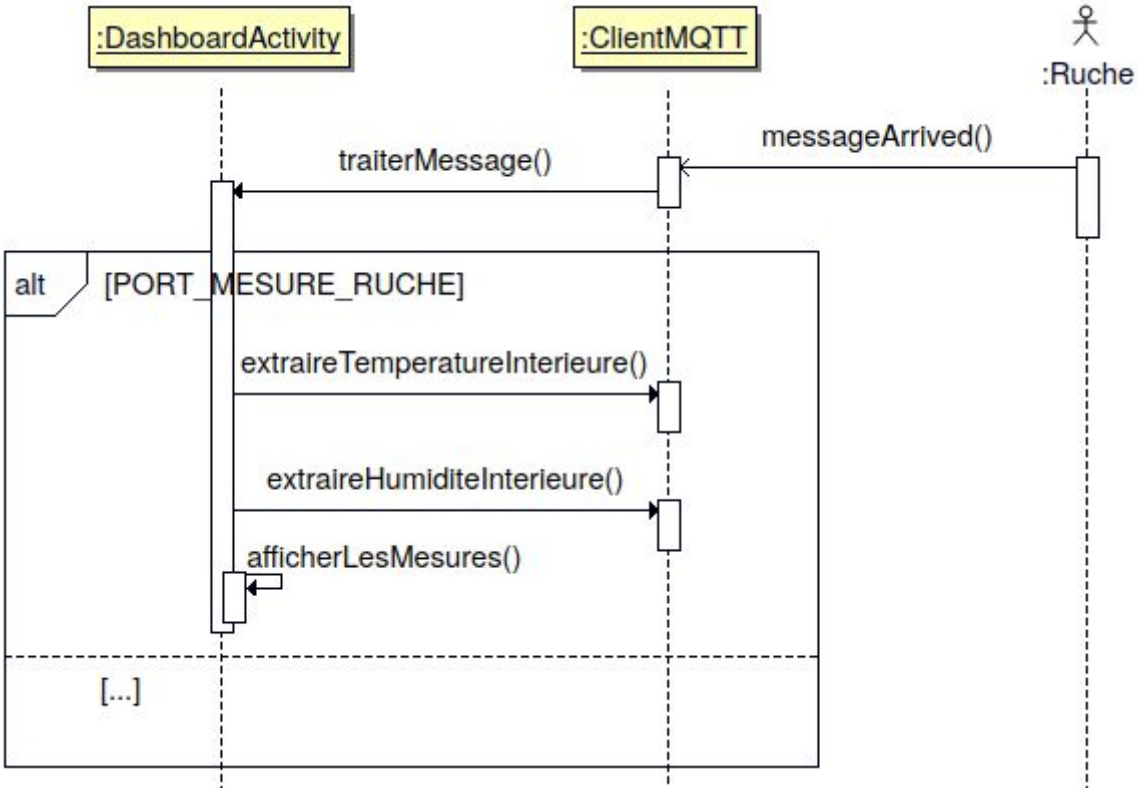
Diagramme de classes

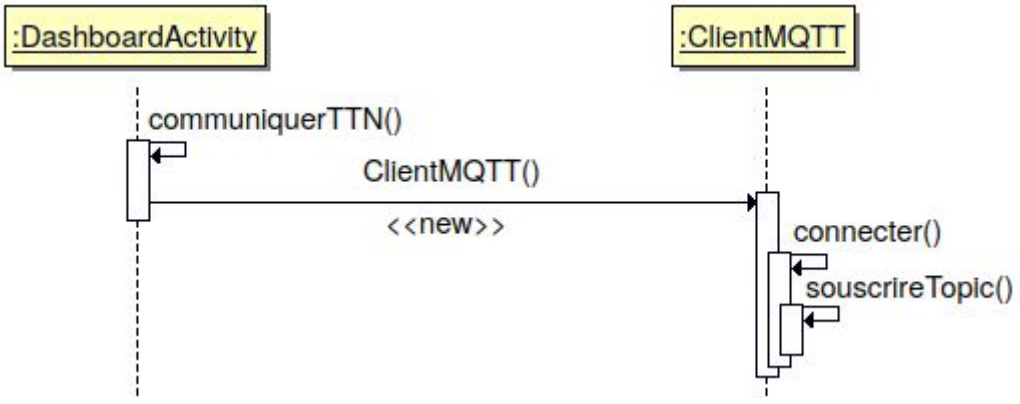
Présentation des classes

Les classes finissant par le mot-clé Activity correspondent aux différentes fenêtres de l'application.

- MainActivity correspond à la page d'accueil
- DashboardActivity correspond à la page tableau de bord
- ParametresHoneyBeeActivity correspond à la page des paramètres de la base de données, accessible depuis la page MainActivity.
- NouvelleRucheActivity correspond à la page d'ajout d'une nouvelle ruche dans la base de données, accessible depuis la page DashboardActivity.
- La classe Ruche permet l'instanciation d'une ruche et donc la lecture des données depuis la base de données
- La classe BaseDeDonnées est un singleton permettant l'accès à la base de données, et l'exécution de requête.
- La classe ClientMQTT permet la récupération en temps réel des données reçu par les capteurs.

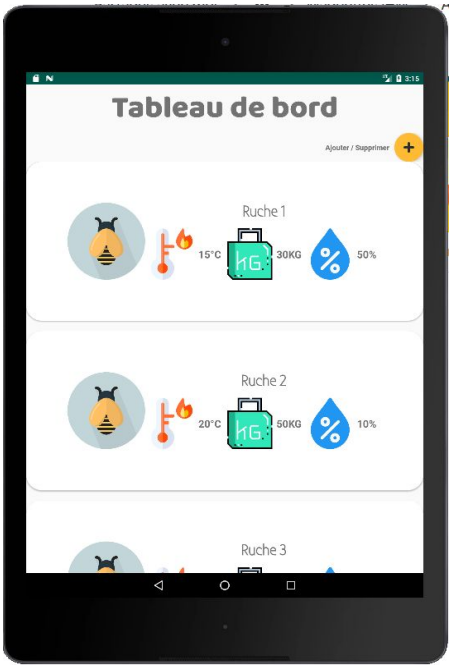
Diagramme de séquences





La gestion d'application dans Android

Dans le jargon d'Android, une fenêtre d'application est appelée **Activity** ou **Activité** en bon Français. Une Activité contient différent **éléments graphiques** tels que des boutons, des champs de saisies de texte, des images, etc …



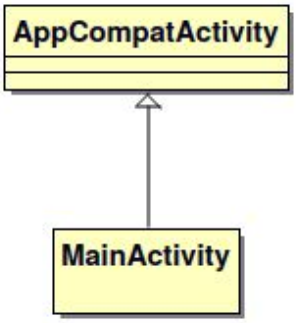
Exemple d'Activity

Chaque Activity possède son **cycle de vie** géré par le système d'exploitation Android. Le système, pour des raisons de priorisation d'activités, peut tuer une activité quand il a besoin de ressources.
L'Activity peut être dans les différents états suivants :

- **« Active » :** l'activité est lancée par l'utilisateur, elle s'exécute au premier plan.
- **« En Pause » :** l'activité est lancée par l'utilisateur, elle s'exécute et est visible, mais elle n'est plus au premier plan. Une notification ou une autre activité lui a volé la priorité et une partie du premier plan.
- **« Stoppée » :** l'activité à été lancée par l'utilisateur, mais n'est plus au premier plan et est invisible. L'activité ne peut interagir avec l'utilisateur qu'avec une notification.
- **« Morte » :** l'activité n'est pas lancée.

Pour créer une Activity, il suffit de créer une nouvelle classe héritant de la classe Activity (ou **AppCompatActivity** pour des raisons de compatibilité) :

```
public class MainActivity extends
AppCompatActivity
{
  ...
}
```

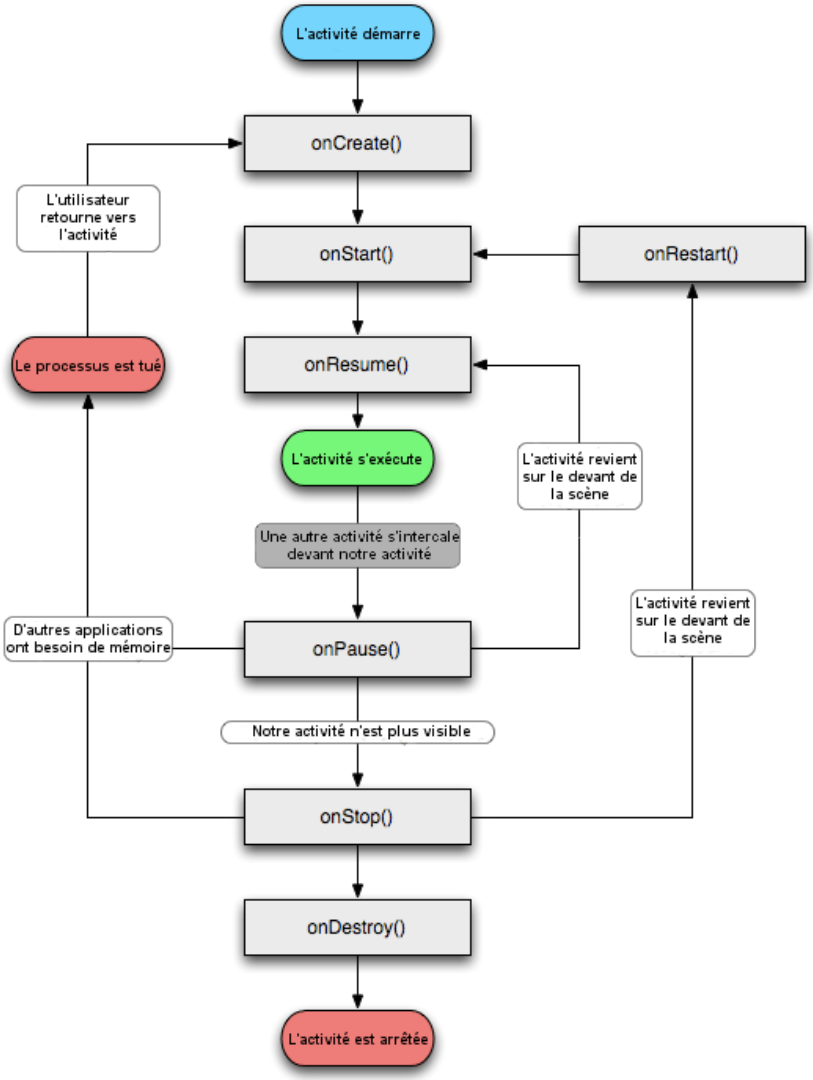


Afin de passer d'un état à un autre l'Activité appele différentes méthodes qui lui sont propre.

- **onCreate()** : elle est appelé lors du premier lancement de l'activité, si l'état du terminal change et que l'activité est associée à cet état par exemple passage du mode portrait au mode paysage. Cette méthode permet aussi les initialisations qui sont effectués qu'une seule fois au lancement de l'activité.

- **onDestroy()** : est appelé lors la mort de l'activité qu'elle soit naturelle (créer par l'action de l'utilisateur) ou provoqué par le système par manque de ressources.

- **onPause()** & **onResume()** : sont appelées lors de l'entrée ou la sortie de l'état en pause de l'activité. Elles s'occupent de sauvegarder ses états et les restituer.

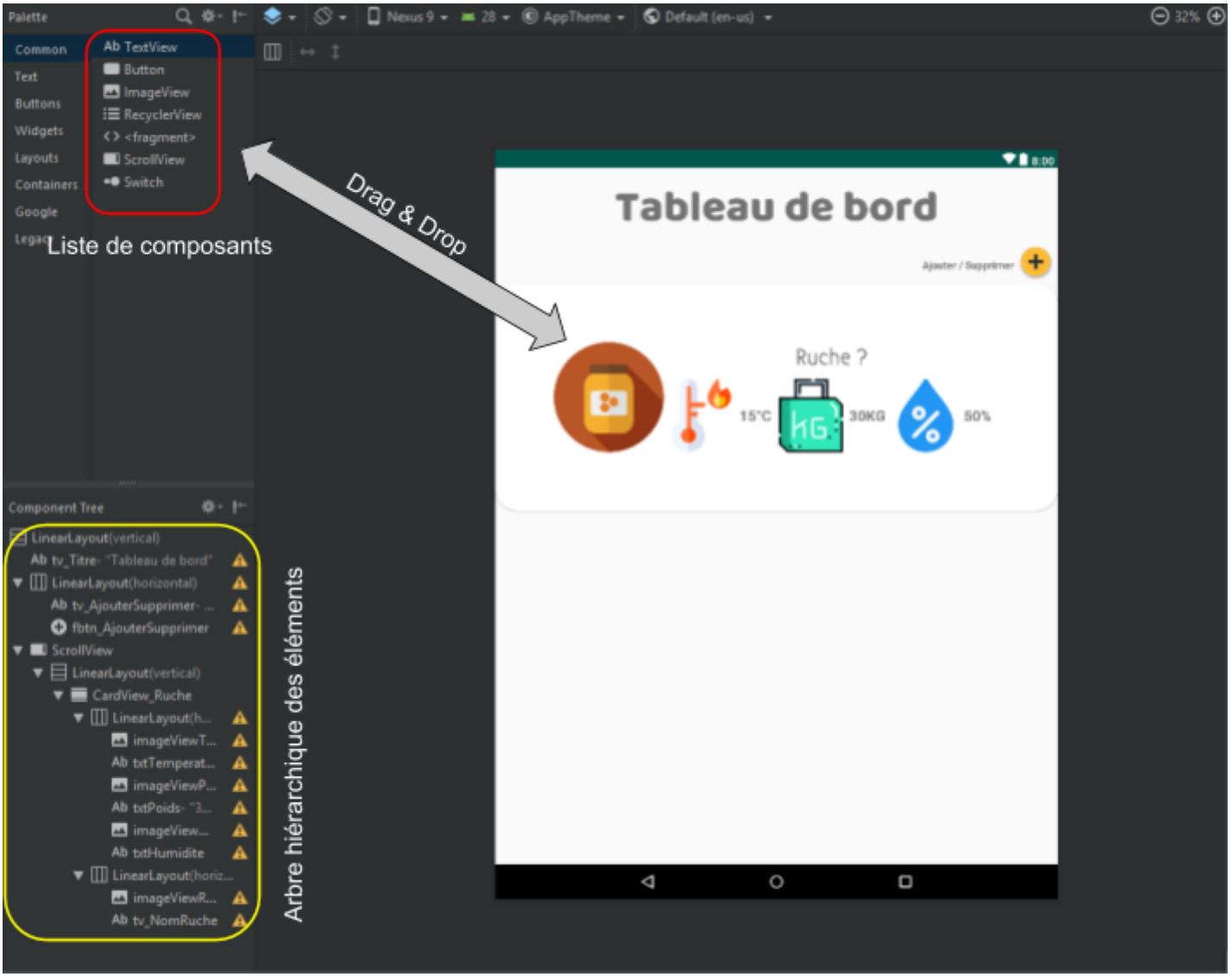


On retrouvera ses méthodes dans la classe de l'activité :

```
public class MainActivity extends AppCompatActivity
{
   @Override
   protected void onCreate(Bundle savedInstanceState)
   {
      ...
   }
   ...
}
```

Modélisation des activités

L'Environnement de Développement Intégré **Android Studio** propose un assistant designer marchant sur du "drag and drop" d'éléments graphique dans la fenêtre de l'activité. Android Studio s'occupe donc de générer le code XML associé. Deux choix sont donc possible l'écriture directement en XML ou en utilisant l'assistant designer.



Rendu dans l'assistant designer

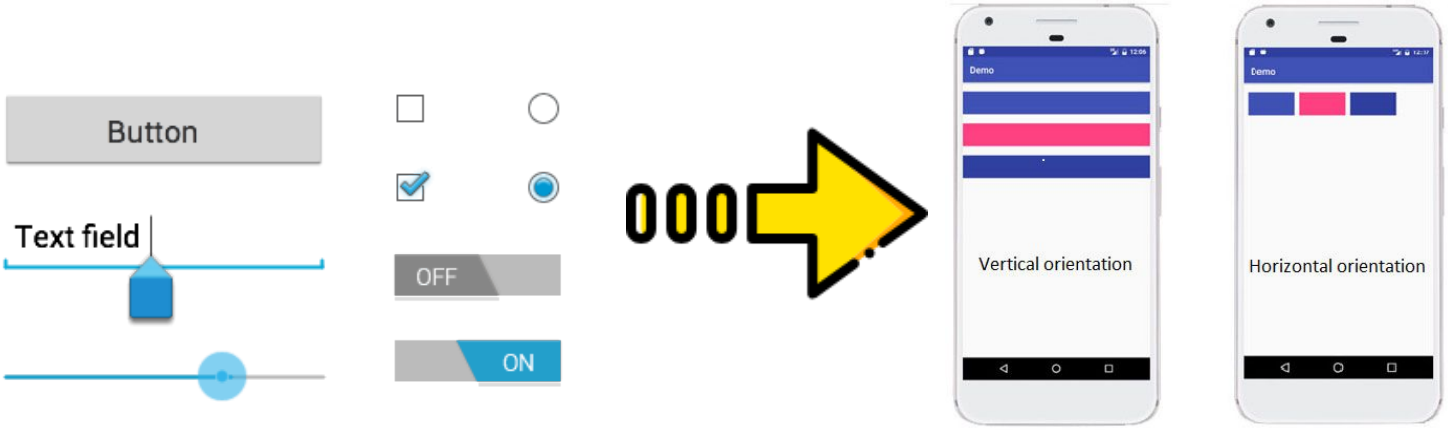
```
<TextView
    android:id="@+id/tv_NomRuche"
    android:layout_width="124dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:fontFamily="@font/bubbler_one"
    android:text="Ruche ?"
    android:textAlignment="center"
    android:textSize="30sp"
    android:textStyle="bold" />
```

Code généré pour le même élément

Chaque élément est déclaré dans le fichier XML avec des balises portant le même nom que le type d'élément ajouté dans l'activité. On a par exemple ici un élément **TextView** délimité par les balises `<TextView .. />`.

On remarque qu'en XML chaque élément est défini par des propriétés chacune personnalisable et agissant sur des caractéristiques de l'élément en question. Chaque élément est défini par un **id** unique ce qui nous permettra par la suite de venir le récupérer dans notre code par la suite. Des options sont personnalisable en fonction de l'éléments graphique en question dans le cas de cette zone de texte appelé **TextView** on remarque la possibilité de changer la police, l'alignement du texte ainsi que sa taille, etc …

Il faut aussi s'occuper de **placer** les éléments graphique à la place désirée, pour cela des layouts sont mis à notre disposition pour **organiser les positionnements**.



Dans le projet je n'ais utilisés qu'un type de layout bien qu'il en existe d'autres, le **LinearLayout**. Le **LinearLayout** permet le positionnement des éléments en son sein sous forme de ligne. Chaque éléments se partage la place disponible en fonctions des autres éléments dans le layout et de la taille de l'appareil sur lequel s'exécute l'application. Une option est particulièrement importante dans le **LinearLayout.**

```
android:orientation="vertical"
```

La propriété **Orientation** permet d'indiquer si les éléments au sein du layout doivent s'organiser selon une orientation verticale ou horizontale.

Chaque élément graphique ainsi décrit en XML se voit donc attribué dans le code JAVA un objet du même type afin de pouvoir être manipuler par la suite dans le code.

```
TextView txtPoids;
txtPoids = (TextView) this.findViewById(R.id.txtPoids);
```

*

La méthode findViewById() permet la mise en relation de l'élément XML et de l'objet JAVA, dans l'exemple ci-dessus la mise en relation d'un élément de type TextView (zone de texte).

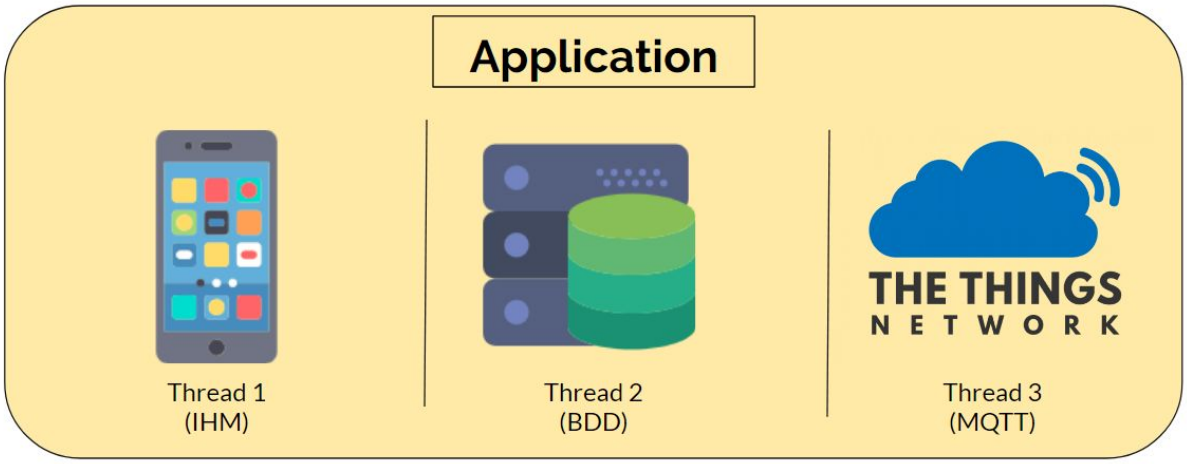
Notion de Thread

Android se réserve le droit de stopper l'exécution d'un processus si son temps de réponse est trop long. Dans le cadre du projet l'enjeu est de garder l'application en fonctionnement même si certaines actions peuvent avoir des temps de réponse long : la connexion à la base de données, requête vers la base de données, réception de données depuis le serveur TTN.

L'utilisation de thread est donc incontournable, en effet elle permet de séparer le programme en plusieurs tâches.

Chaque thread représente un fil d'exécution pour Android.

On aura donc un thread pour l'IHM appelé MainThread ou ThreadUI, un thread pour la récupération de données depuis la base de données et enfin un thread pour la récupération de données depuis le serveur TTN.



Base de données

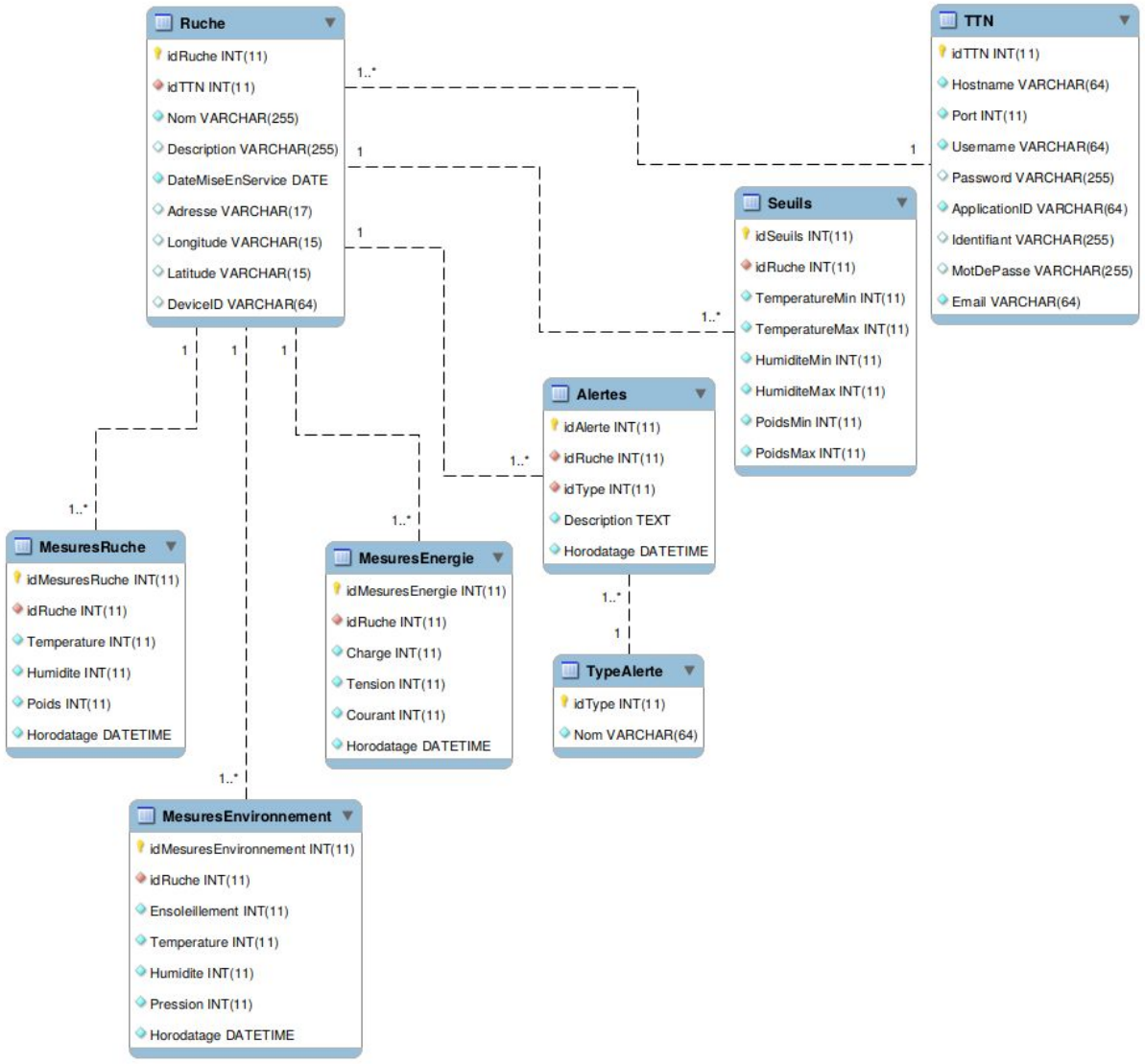


Schéma de base de données

La base de données est composé de différentes tables reliées entres elles par des relations de clé primaire et clé étrangère. La clé qui servira ici de relation est le champ **idRuche**.

Le système de gestion de base de données utilisé est MySQL, pour récupérer les données on exécutera donc des requêtes SQL.

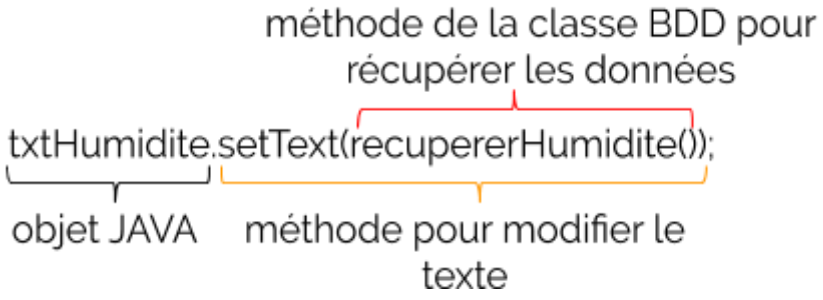
```
SELECT * FROM MesuresEnvironnement
WHERE idRuche = '1'
```

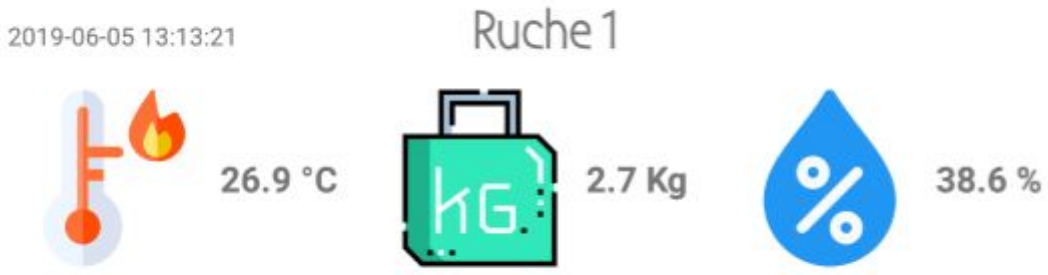
```
ORDER BY Horodatage DESC LIMIT 5;
```

Les mots clés SELECT et FROM constitue la base de la requête, ici une requête de récupération de donnée. Le mot clé SELECT indique les champs à récupérer ici l'étoile désigne tous et le mot clé FROM désigne dans quelle table les données doivent être récupéré.

Les mots clés en orange sont optionnel et permette de rajouter des conditions lors de la récupération de données, ici on tri les données de manière descendante par rapport au champs Horodatage et on prends donc les 5 derniers résultats.

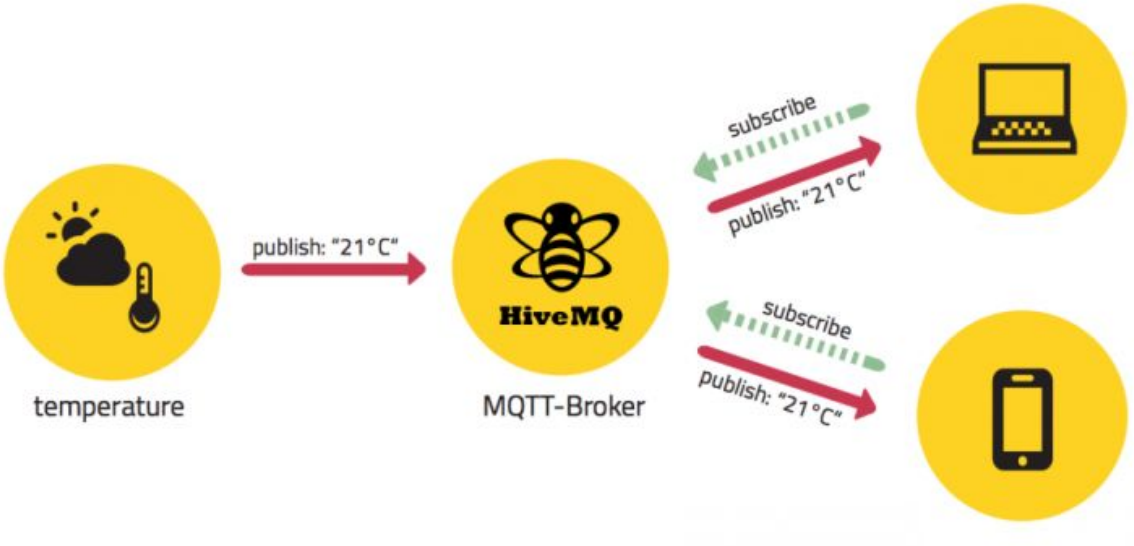
Les données sont donc affichés dans l'IHM après récupération, grâce aux méthodes d'objets définis auparavant.



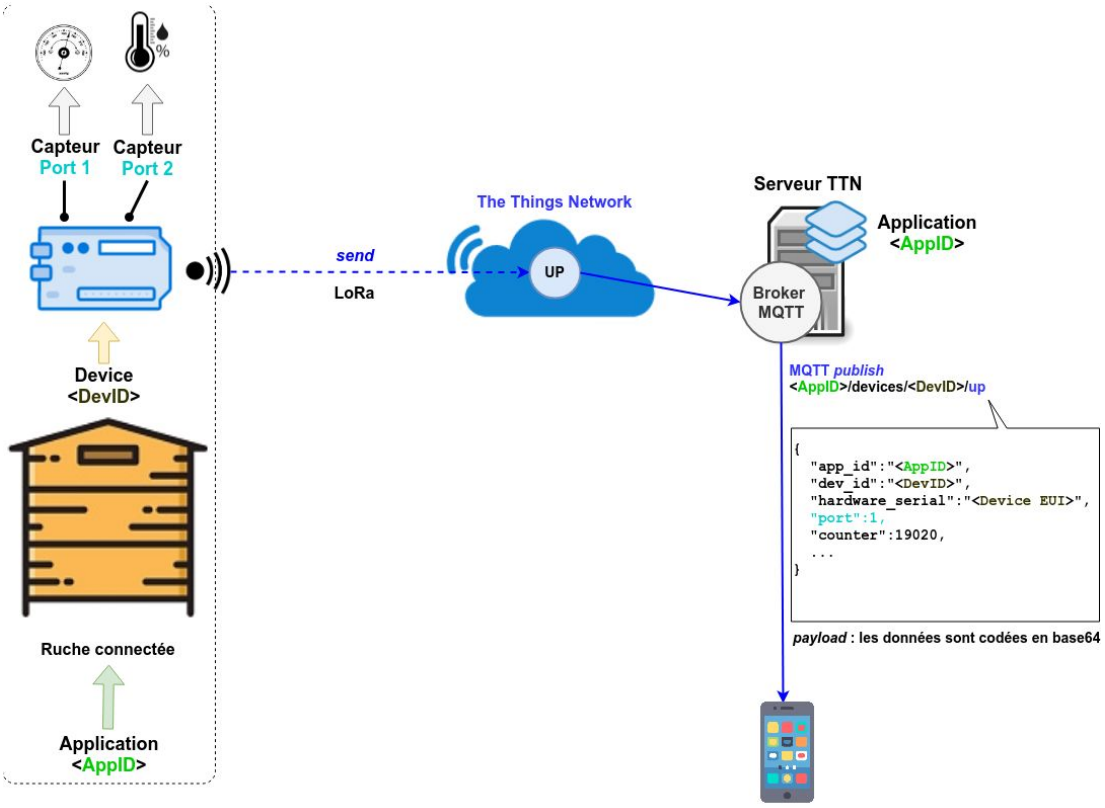


TTN / MQTT

TTN désigne The Things Network, il s'agit d'un réseau communautaire permettant l'échange de données avec des objets connectés. Le protocole de communication utilisé par ce réseau est le protocole MQTT, et fonctionne avec un système d'abonnement (suscribe) et publication (publish) de données sur un sujet (topic).



Les clients s'abonnent (subscribe) au sujet (topic) qu'il les intéresse dans l'exemple ci-dessus le sujet température.



L'AppID désigne l'identifiant du projet donc un ensemble de ruches pour l'apiculteur.

Le DevID désigne la carte microcontrôleur et ses capteurs dans une ruche.

Ensuite les numéros de ports correspondent finalement à des branchements de capteurs. La carte microcontrôleur envoie les données en LoRa (Base64) au serveur TTN qui lui s'occupera de les décoder pour qu'elles soient directement exploitables par tous les clients abonnées.

Le MQTT-Borker se situant dans le serveur TTN s'occupera par la suite à chaque données envoyées de les retransmettre en temps réel au client s'étant abonné au topic correspondant.

Afin de les récupérer en JAVA, on utilise les dépendances Android Service pour pouvoir utiliser les Threads, et MQTT Paho pour disposer de classe permettant la réception de message MQTT et de signaux quand les messages sont réceptionnés (MessageArrived).

A chaque fois qu'un message est réceptionné la méthode MessageArrived de la classe MQTT Paho s'exécute et on récupere les données au format JSON.

```
{
        "app_id":"mes_ruches",
        "dev_id":"ruche_1",
        "hardware_serial":"0004A30B00203CF8",
        "port":3,
        "counter":19909,
        "payload_raw":"CHoWRA==",
        "payload_fields":{"humidite":57,"temperature":21.7},
        "metadata":{"time":"2019-05-03T08:22:39.727568713Z",
         "frequency":867.5,
         "modulation":"LORA",
         "data_rate":"SF7BW125",
         "airtime":51456000,
         "coding_rate":"4/5",

"gateways":[{"gtw_id":"btssn-lasalle-84","gtw_trusted":true,"timestamp":3252318292,"time":"2019
-05-03T08:22:39Z","channel":5,"rssi":-34,"snr":7.25,"rf_chain":0,"latitude":43.948326,"longitude":
4.8169594,"location_source":"registry"}]}
}
```


Réception de données au format JSON

Les champs entourés sont les champs exploités pour l'affichage dans l'IHM.

Le **dev_id** permet de savoir de quelle ruche il s'agit, le **port** permet de savoir de quelle type de donnée il s'agit, le **payload_fields** permet de récupérer les valeurs des capteurs, et les **metadata** sont les informations relatives à la trame comme le jour et l'heure de l'envoi de celle-ci.

L'affichage dans l'IHM se fait comme pour la base de données avec la méthode **setText** des objets de l'IHM.


```
{"app_id":"mes_ruches","dev_id":"ruche_1","hardware_serial":"0004A30B00203CF8","port":4,"counter":
30106,"payload_raw":"CmQPCgP1","payload_fields":{"humidite":38.5,"pression":1013,"temperature":26.6},"metadata":
{"time":"2019-06-05T13:13:33.298106134Z","frequency":867.9,"modulation":"LORA","data_rate":"SF7BW125","airtime":
51456000,"coding_rate":"4/5","gateways":[{"gtw_id":"btssn-lasalle-84","gtw_trusted":true,"timestamp":
4133839180,"time":"2019-06-05T13:13:33Z","channel":7,"rssi":-78,"snr":10.25,"rf_chain":0,"latitude":43.948326,"longitude":
4.8169594,"location_source":"registry"}]}}
```

Tests de validations

Désignation	Démarche à suivre	Résultat obtenu	Fonctionnel	Remarques
Lire les données	Cliquer sur l'écran d'accueil	Visualisation des données	Oui	
Paramétrer une nouvelle ruche	Cliquer sur le bouton "Ajouter une ruche" sur la page Tableau de Bord	Page de paramétrage d'une nouvelle ruche	Oui	
Supprimer une ruche	Cliquer sur le bouton "Supprimer une ruche" sur la page Tableau de Bord	Suppression d'une ruche dans la base de données	Oui	
Gérer les alertes	Cliquer sur le bouton "Gérer les alertes" sur la page de Tableau de Bord	Modifications des seuils d'alertes	Non	

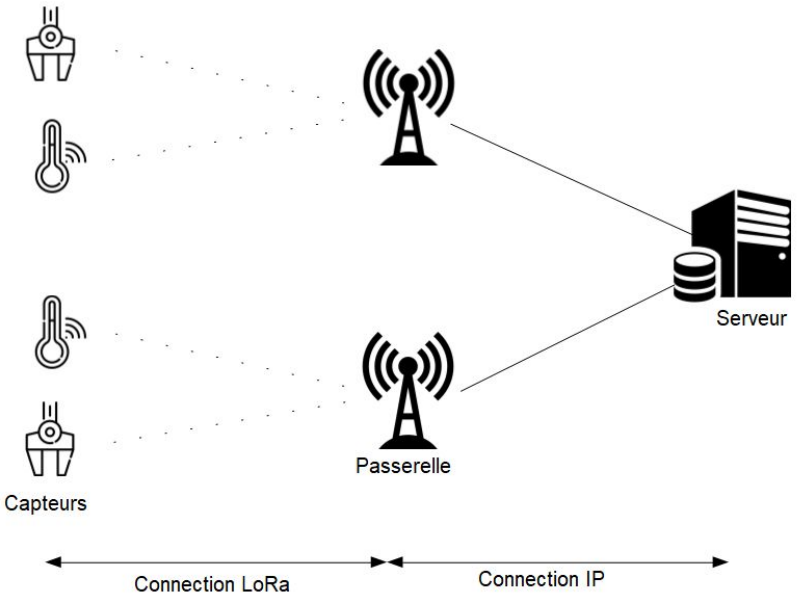
Partie physique

LoRaWAN est un protocole de télécommunication permettant la communication à bas débit, par radio, d'objets à faible consommation électrique communiquant selon la technologie LoRa et connectés à l'Internet via des passerelles, participant ainsi à l'Internet des objets.

Ce protocole est utilisé dans le cadre des villes intelligentes, le monitoring industriel ou encore l'agriculture.

Le protocole LoRaWAN sur la couche physique LoRa permet de connecter des capteurs ou des objets nécessitant une longue autonomie de batterie (comptée en années), dans un volume (taille d'une boîte d'allumettes ou d'un paquet de cigarettes) et un coût réduits.

LoRaWAN est l'acronyme de Long Range Wide-area network que l'on peut traduire par « réseau étendu à longue portée ».

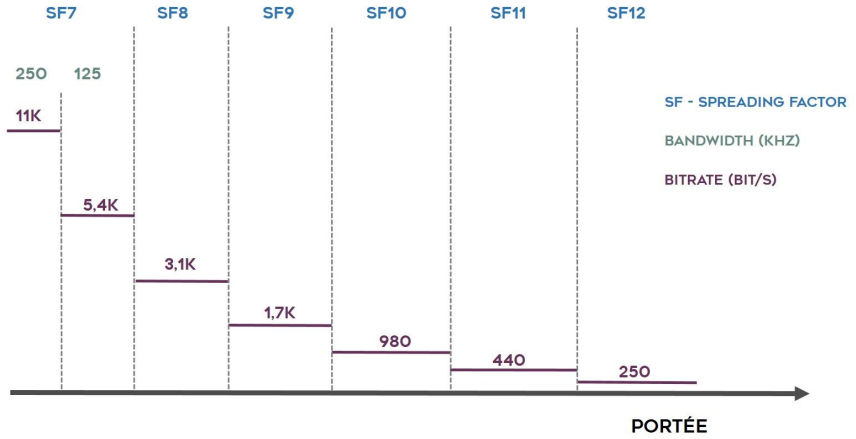


L'utilisation de fréquences libres impose de respecter un temps d'occupation maximum du canal radio. LoRa permet de fixer les principaux paramètres radio à l'aide du paramètre Data Rate. Le Data Rate est défini par un chiffre de 0 à 15 et fixe le type de modulation, le *spreading factor* ainsi que la bande passante utilisée.

Data Rate (DR)	Modulation	Spreading Factor (SF)	Bande Passante	Débit Physique (bit/s)
0	LoRa	SF12	125 kHz	250
1	LoRa	SF11	125 kHz	440
2	LoRa	SF10	125 kHz	980
3	LoRa	SF9	125 kHz	1 760
4	LoRa	SF8	125 kHz	3 125
5	LoRa	SF7	125 kHz	5 470
6	LoRa	SF7	250 kHz	11 000
7	FSCK	50kbit/s		50 000
8	Réservé pour utilisation future			

Data Rates pour la bande 863-70 MHz

L'étalement du signal augmente sa portée, au détriment du débit car il est transmis sur une plus longue période. Au détriment également de l'autonomie de l'équipement car la communication radio est énergivore ! Par conséquent une communication plus longue implique une consommation d'énergie plus importante.



Annexe

Diagramme de classes complet