

WeatherStation :
Systèmes d'exploitation mobiles
Programmation des systèmes mobiles

LOUVEAU Julien, MARTINOT Romain, MENTEN Simon
MERCENIER Louis & STREIGNARD Rémi

SESSION JUIN 2021

Sommaire

1	Architecture globale	1
2	Section IOT	2
2.1	La station météo	2
2.1.1	Conception électronique de la station	3
2.1.2	Composants et montage	4
2.2	Pourquoi une Raspberry Pi ?	5
2.3	Pourquoi MQTT ?	6
2.4	Envoi et réception des données	6
3	Section API	8
3.1	Pourquoi une API ?	8
3.2	Pourquoi Java ?	8
3.3	Pourquoi REST ?	8
3.4	Base de données	9
3.5	Structure des points d'accès	10
4	Développement mobile	12
4.1	Choix de la technologie	12
4.2	Description de l'application	12
4.2.1	Segmentation du code	13
4.2.2	Partie site	14
4.2.3	Partie room	15
4.2.4	Partie sensor et device	16
4.2.5	Partie Event	17
5	Possibilités d'améliorations	18
5.1	Ajout des graphes	18
5.2	Meilleure segmentation du code	18
5.3	Authentification	18
5.4	Notifications	18
5.5	Implémentation concrète du MQTT	19
5.6	Sécurité	19
5.7	QR code	19

Ce projet a comme objectif la conception et la réalisation d'une application mobile dont la tâche est de faciliter le management des différents capteurs et appareils d'une maison connectée. Chaque groupe dispose alors d'un ensemble de capteurs dont les informations doivent être envoyées et centralisées dans un espace commun afin d'être ensuite récupérées et utilisées par l'intégralité des groupes.

Chacune de ces informations doit ensuite être affichée au travers de cette application et diverses fonctionnalités doivent être implémentées sur base de choix personnels effectués par chaque groupe.

Ce projet est disponible au lien suivant : [WeatherStation](#)

1 Architecture globale

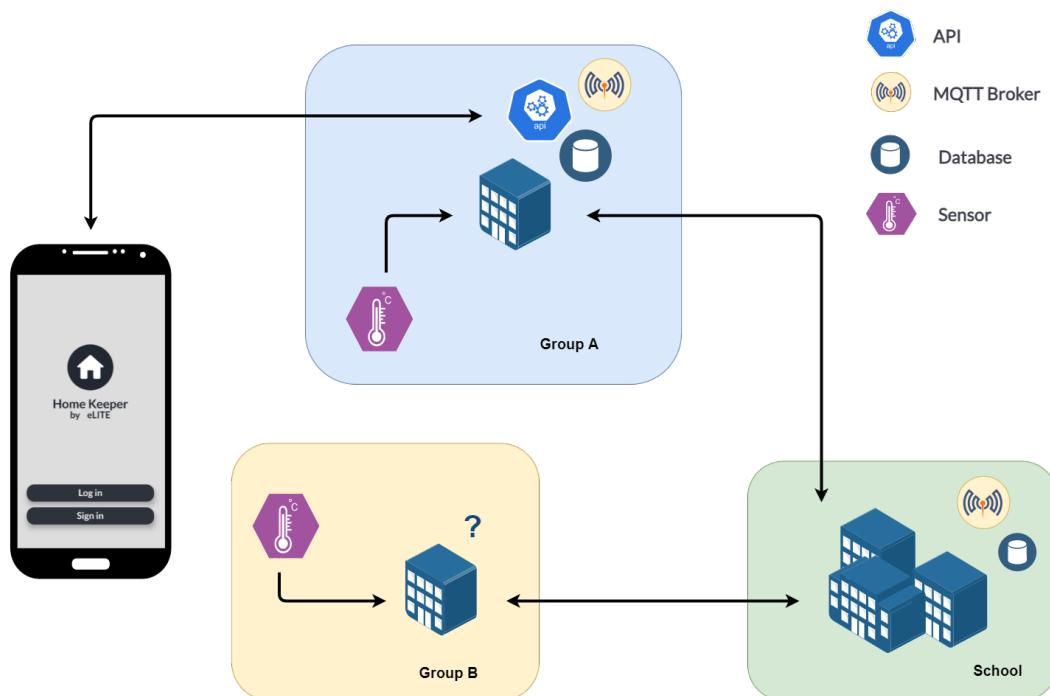


Figure 1: Représentation de l'architecture du projet et de ses services

Afin d'uniformiser et de garantir une application fonctionnelle sans pour autant devoir prendre des mesures de compatibilité avec les services de chaque groupe, une architecture dénuée de toute dépendance envers ces derniers a été implémentée.

Sur l'illustration ci-dessus figurent les différents services agencés lors de l'implémentation du projet.

S'y retrouvent alors; les services propres à l'API, la gestion du MQTT, le service de base de données ainsi que les *sensors*. L'agencement des services des autres groupes n'est en revanche pas connu, ceci étant représenté par le point d'interrogation.

Les sections qui constituent ce rapport s'attellent donc à décrire l'agencement entre ces services, les choix qui les entourent ainsi qu'une analyse propre à chacun.

2 Section IOT

Cette section va reprendre les choix qui ont été faits concernant la partie IOT du projet, ainsi qu'une description du fonctionnement de celle-ci.

2.1 La station météo

Pour notre projet, nous avons créé une station météo qui a la possibilité de renvoyer les données des capteurs à l'application mobile. Cette station météo est composée de 3 capteurs : un capteur de température, d'humidité et de luminosité. Ces capteurs sont pilotés grâce à un micro contrôleur. Les données de la station sont affichées sur un écran LCD. Les données récupérées sur les différents capteurs sont également envoyées sur une Raspberry.

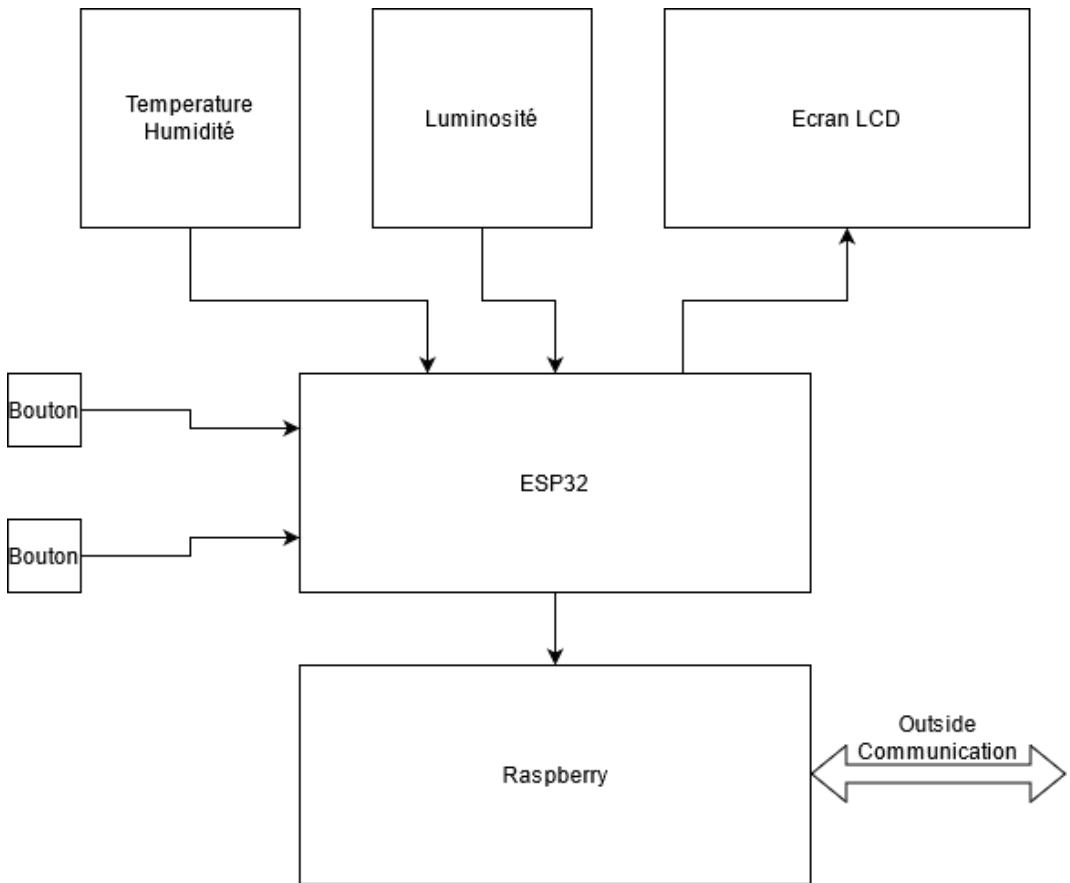


Figure 2: Diagram de l'architecture de la station météo.

Ce diagramme représente de manière schématique les différentes interactions entre les différents composants de notre station météo. Notre station météo est donc composée d'un capteur de température et humidité ainsi qu'un capteur de luminosité. Les capteurs renvoient leurs données à l'ESP32. Celui-ci est également équipé d'un écran LCD et de deux boutons. L'écran LCD permet d'afficher les différentes données de température, humidité et luminosité en temps réel. Les deux boutons permettent à l'utilisateur de naviguer entre les différents menus de l'écran LCD.

2.1.1 Conception électronique de la station

Avant de réaliser le montage, nous avons d'abord réalisé un schéma électronique de la station météo. Dans ce schéma, le microcontrôleur n'est pas un ESP32, mais un Arduino. Cependant, les numéros des branchements correspondent aux numéros des pinnes de l'ESP32. Cela veut dire que toutes les étiquettes commençant par la lettre *g* et suivies d'un numéro se brancheront au pin correspondant sur l'ESP32.

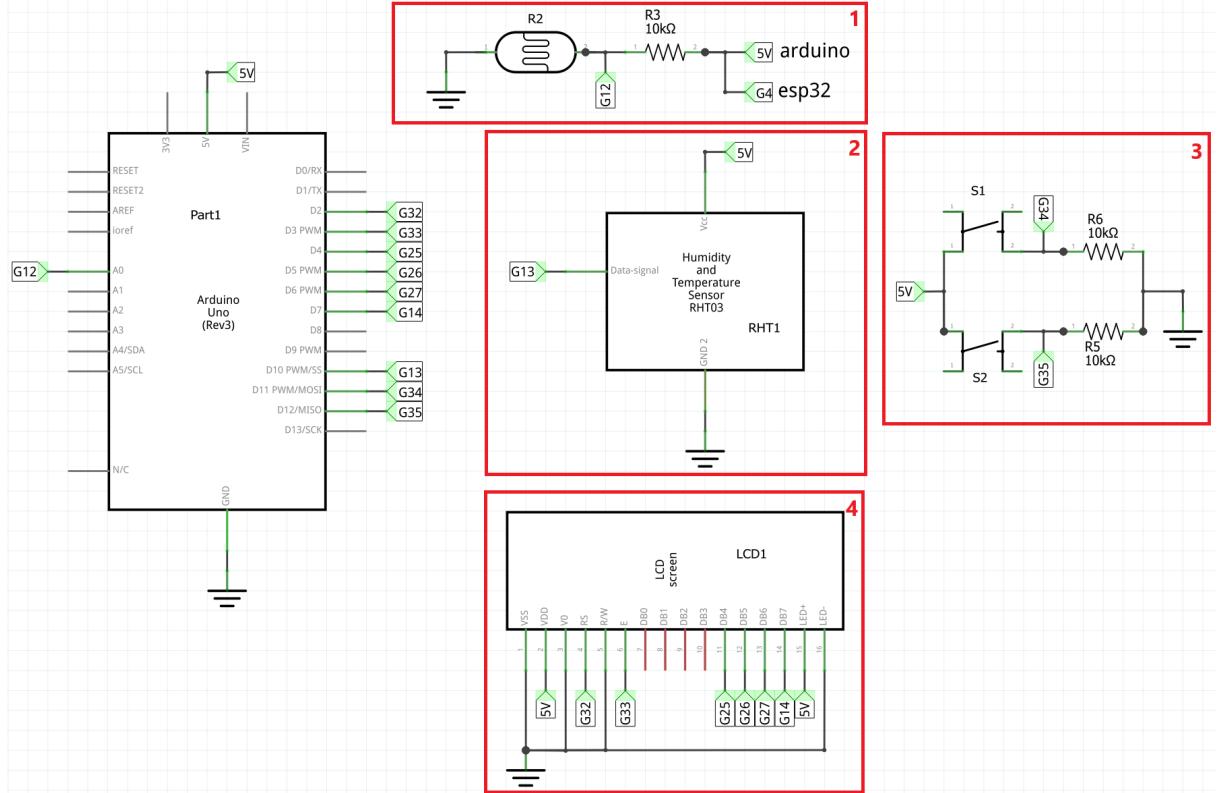


Figure 3: Schéma électronique de la station météo.

L'électronique de la station météo peut être divisée en plusieurs blocs expliqués dans les points suivants :

- **Le bloc numéro 1** permet de mesurer les différences de luminosités. Cela fonctionne grâce à une photorésistance. Une photorésistance est une résistance qui voit sa valeur changer en fonction de la luminosité à laquelle elle est exposée. Pour mesurer cette différence de résistance, on utilise un pont diviseur de tensions. Un pont diviseur de tensions est un montage électronique qui permet de diviser une tension en fonction des valeurs des résistances utilisées.

Sur le schéma, les résistances utilisées sont R2 et R3. R2 a une résistance variable et R3 a une résistance de 10 Khome. Pour mesurer les différences de luminosité, il suffit de mesurer la différence de tension à l'étiquette G12. Pour que le montage fonctionne, il faut lui appliquer une tension de 3.3v (pour ESP32) aux extrémités du montage. Sur le schéma, on peut remarquer que l'alimentation ne se fait pas à travers le Vcc, mais par le pin G4 de ESP32. Cela a été fait après des problèmes de démarrage de ESP32 qui ne démarrait pas quand une tension trop élevée est appliquée au pin G12. Cela permet donc d'allumer le capteur qu'après le démarrage de ESP32.

- **Le bloc numéro 2** permet de mesurer l'humidité et la température de l'air ambiant. Pour le faire fonctionner, il suffit de connecter le GND, le Vcc et le pin qui transmet les données. Dans notre cas, nous l'avons connecté à la pin G13 de ESP32.
- **Le bloc numéro 3** est composé de deux interrupteurs. Pour savoir si l'interrupteur est dans une position fermée ou ouverte, il est nécessaire de le connecter au 3.3V d'un côté et à un pin de ESP32 de l'autre. Dans notre cas, les boutons sont connectés aux pins G34, G35. Il est également nécessaire de connecter les pins G34 et G35 à la masse à travers une résistance. Si les pins G34 et G35 n'étaient pas connectés à la masse, il est possible que des erreurs de lecture se produisent au niveau de ESP32.
- **Le bloc numéro 4** est l'affichage LCD. Cet affichage permet d'afficher du texte envoyé par ESP32. Cet afficheur reçoit ces données sur les pins DB4, DB5, DB6, DB7. Le pin RS permet de sélectionner le registre et le pin E permet activer l'affichage.

2.1.2 Composants et montage

Pour que le système soit fiable et sans trop de câbles, nous avons soudé les différents composants sur un PCB de test. Ce PCB de test est alors tout de suite connecté à la Raspberry via le port GPIO. Cela permet de contenir toute l'électronique sur un seul module.

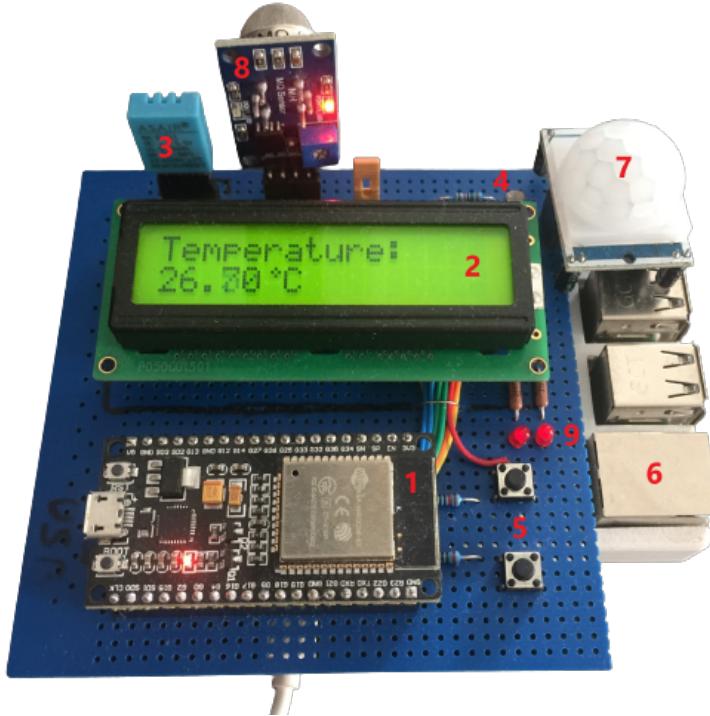


Figure 4: Photo de la station météo.

- **Le composant 1** est l'ESP32. L'ESP32 est un microcontrôleur qui dans notre projet permet de gérer les différents capteurs, bouton et écran. Il permet également de faire une interface entre tous les capteurs et la Raspberry.
- **Le composant 2** est l'écran LCD. Il est connecté à l'ESP32 et permet d'afficher différentes données obtenues par les différents capteurs.

- **Le composant 3** est le capteur de température et d'humidité. Les données de ce capteur sont traitées par l'ESP32.
- **Le composant 4** est le capteur de luminosité. Les données de ce capteur sont traitées par l'ESP32.
- **Le composant 5** consiste en deux boutons. Ils sont connectés à l'ESP32 et permettent de naviguer dans les différents menus de l'écran LCD.
- **Le composant 6** est la Raspberry. Elle permet de récupérer les différentes données de l'ESP32 et de les transmettre via le protocole MQTT.
- **Le composant 7** est un capteur de mouvement. Ce capteur ne fait pas partie de la station météo, mais nous permet d'avoir une source de donnée différente pour notre projet. Ce capteur est également connecté à l'ESP32 qui renvoie les données du capteur à la Raspberry.
- **Le composant 8** est un capteur de qualité de l'air. Ce capteur ne fait pas partie de la station météo, mais nous permet d'avoir une source de donnée différente pour notre projet. Ce capteur est également connecté à l'ESP32 qui renvoie les données du capteur à la Raspberry.
- **Le composant 9** est constitué de deux LED. Ces deux LED sont directement connectées à la Raspberry. Elles permettent de simuler une action que la Raspberry devrait effectuer comme ouvrir ou fermer un volet. Elle fonctionne à travers un petit serveur web Flask. Pour allumer ou éteindre une LED il suffit d'envoyer une requête "PUT" sur la Raspberry à l'URL suivant : <http://0.0.0.0:5000/device/{id}> avec un json contenant le statut *true* ou *false* en fonction de si on veut allumer ou éteindre la LED.

```
{
    "activate": true
}
```

2.2 Pourquoi une Raspberry Pi ?

Nous avons décidé de joindre une Raspberry Pi 3 à notre projet pour faciliter la portabilité du projet. En effet, tant qu'une connexion internet et du courant (secteur ou batterie) sont disponibles, le projet peut être déployé où l'on veut. De plus, l'utilisation de la RPI permet le formatage, l'envoi et la réception de données nécessaires au projet. Celle-ci sert donc de "relais" entre notre ESP32 et notre API.

Cependant, il existe des contraintes au déploiement du projet. En effet, pour que celui-ci soit opérationnel, il est nécessaire que la RPI intègre plusieurs paquets utilitaires. Comme Java et Python, ceux-ci sont nécessaires au fonctionnement de l'API et des scripts de communications. Il est aussi nécessaire d'avoir une base de données MySQL pour récolter les différentes valeurs des capteurs.

Pour pallier ce problème de dépendances, on pourrait envisager la création d'un script qui s'occuperait de ces différentes exigences lors du premier lancement du projet.

2.3 Pourquoi MQTT ?

Après une réunion avec les autres groupes d'élèves, nous sommes arrivés à un consensus, utiliser le protocole de communication MQTT. Celui-ci sert à publier les différentes valeurs des capteurs du projet, ainsi qu'à recevoir les valeurs des autres groupes.

Ce protocole permet un échange de données très rapide, ainsi que des mises à jour quasi instantanées. Il nous donne donc la possibilité de ne pas devoir systématiquement demander à la base de données de l'école pour les dernières valeurs.

Il est important de noter que le MQTT, bien que très intéressant pour un petit projet, commence à s'essouffler lorsqu'un nombre important de clients l'utilise. En effet, la capacité d'un réseau MQTT à soutenir une charge de travail est directement proportionnelle aux ressources du broker. Le rôle de celui-ci étant de conserver la connexion avec tous ses clients.

2.4 Envoi et réception des données

Envoi :

- Lecture des données produites par les capteurs de l'ESP32 sur la RPI à l'aide d'un script Python.
- Publication de ces données sur le réseau MQTT via le même script Python. Le script utilise les commandes du paquet Mosquitto. Celui-ci permet d'envoyer et de recevoir des données en devenant un broker MQTT.

Le schéma ci-dessous présente les différentes interactions entre les services du projet lors de l'envoi des données.

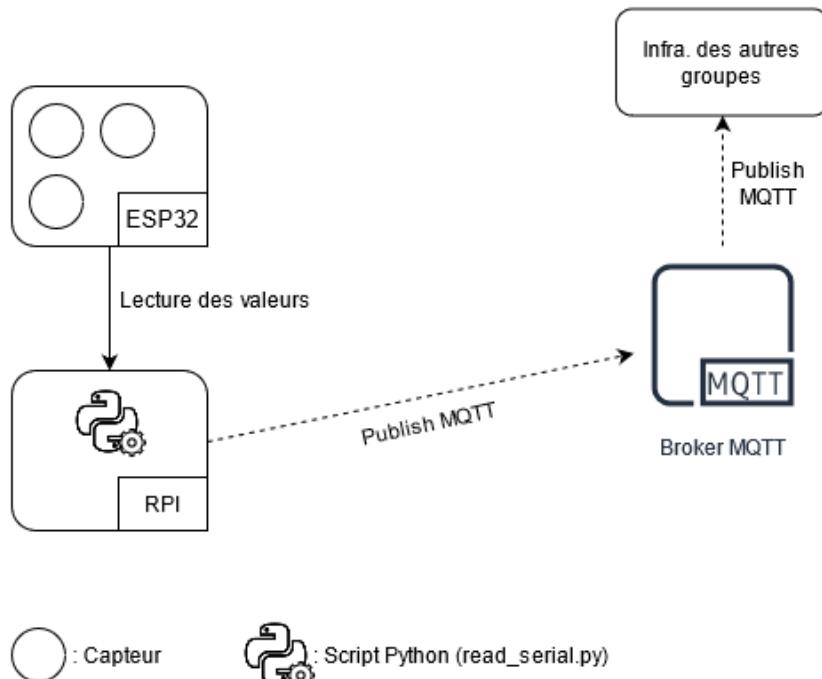


Figure 5: Schéma de l'envoi de données

Réception :

- Un script Python récupèrent les données publiées par les autres groupes via le broker MQTT de l'école.
- Ensuite, ce qui vient d'être récupéré est envoyé sur la base de données via un appel à l'API.

Le schéma ci-dessous présente les différentes interactions entre les services du projet lors de la réception des données.

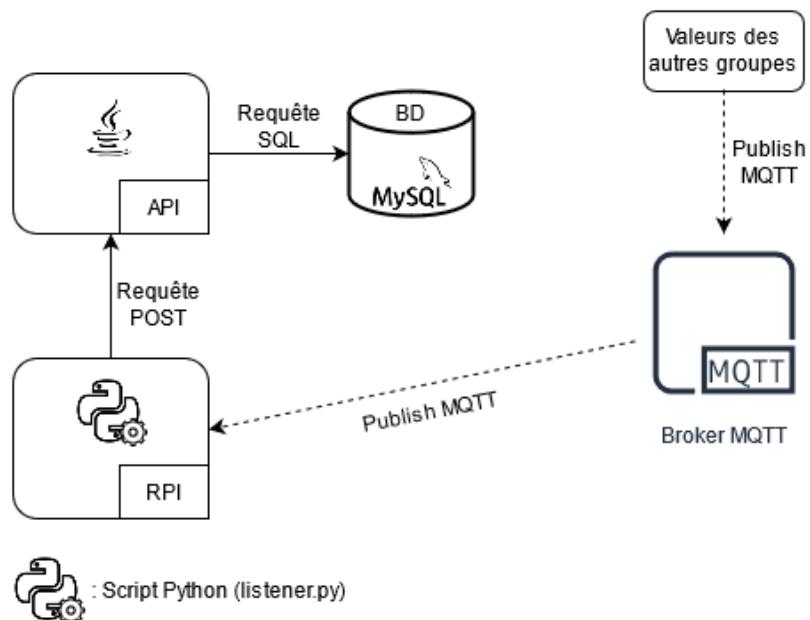


Figure 6: Schéma de la réception de données

3 Section API

Cette section explique les raisons pour lesquelles nous avons décidé de réaliser une API, les choix technologiques la concernant, des exemples sur son utilisation, ainsi que des explications sur la structure de la base de données qu'elle manipule.

3.1 Pourquoi une API ?

Nous avons décidé de placer une API entre le client et la base de données plutôt que de faire communiquer directement ces derniers entre eux. Cela nous permet de modéliser les données selon nos préférences, indépendamment de la base de données commune entre les groupes. De plus, des contrôles supplémentaires peuvent être effectués sur les données avant de les manipuler. L'évolutivité globale est également améliorée puisque le client ne dépend plus du type de la base de données utilisée.

3.2 Pourquoi Java ?

Nous avons choisi de réaliser le serveur en Java. L'avantage d'utiliser ce langage est le mécanisme mis en place par l'utilisation de la Java Virtual Machine. Il permet d'obtenir une application indépendante de la plateforme tout en étant plus rapide que d'autres langages interprétés.

3.3 Pourquoi REST ?

Nous avons également décidé d'implémenter l'architecture REST. Les contraintes suivantes sont donc respectées :

- Séparation des responsabilités entre le client et le serveur.
- Le client ne sait pas s'il est connecté directement au serveur final.
- L'état de la session de communication n'est pas stocké sur le serveur. Chaque appel contient les informations nécessaires à la résolution de la requête.
- L'interface est uniforme. Les données sont représentées sous la forme de ressources et sont accessibles en manipulant une URL et le type de la requête HTTP.

Le respect de ces différentes contraintes fournit les avantages suivants :

- Grande évolutivité grâce à la séparation entre le client et le serveur.
- L'absence d'états permet de répartir la gestion des requêtes sur plusieurs serveurs.
- L'utilisation de formats standards pour la communication assure la compatibilité dans le temps.

3.4 Base de données

Au démarrage de l'application, une connexion avec le serveur MySQL est établie. La base de données est créée si elle n'existe pas.

Cette dernière contient les tables suivantes :

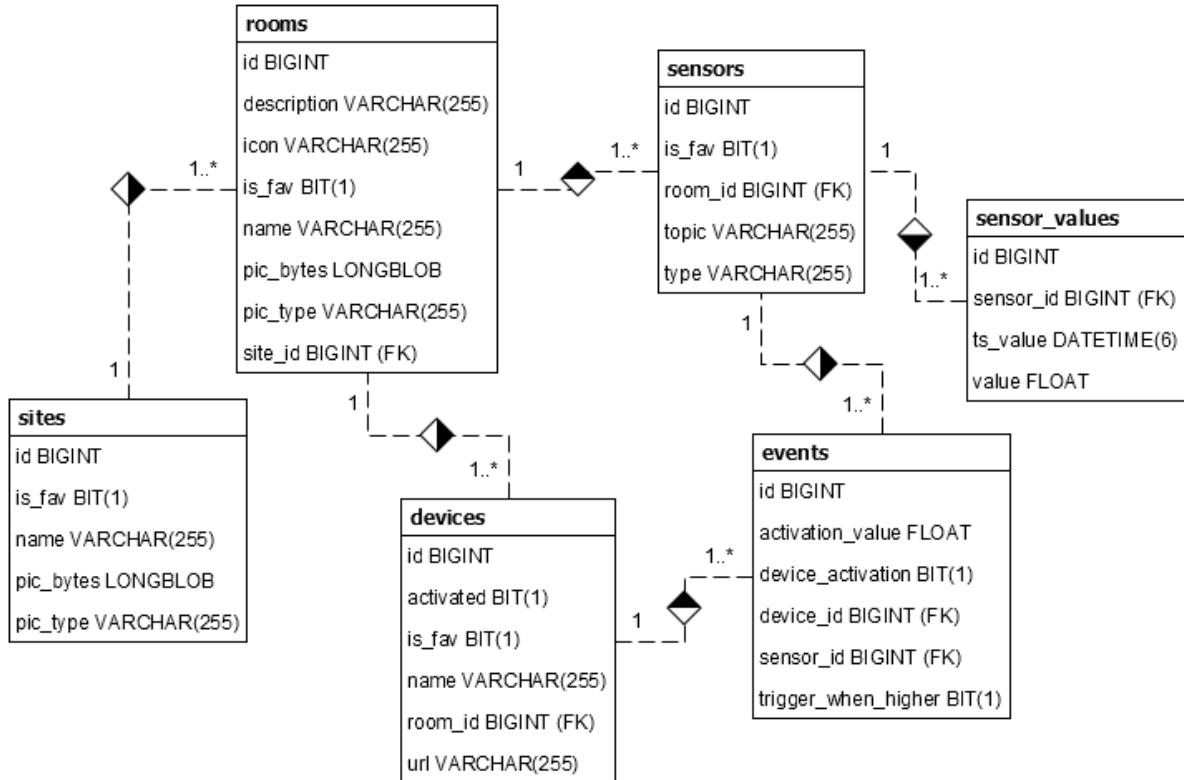


Figure 7: Schéma de la base de données

- La table **sites** contient la liste des sites existants. Un site est composé d'un identifiant, d'un nom, d'un booléen qui indique s'il est classé en tant que favori et d'un tableau de bytes représentant l'image associée.
- La table **rooms** contient la liste des pièces existantes. Une pièce est composée d'un identifiant, d'un nom, d'une description, d'un booléen qui indique si elle est classée en tant que favori, d'un tableau de bytes représentant l'image associée et de l'identifiant du site dans lequel elle se situe.
- La table **sensors** contient la liste des capteurs. Un capteur est composé d'un identifiant, d'une description, d'un booléen qui indique s'il est classé en tant que favori, d'une indication sur le type des valeurs associées et de l'identifiant de la pièce dans laquelle il se situe.
- La table **sensor_values** contient la liste des différentes valeurs reçues par les capteurs. Un tuple de cette table est composé d'un identifiant, de la date à laquelle l'information a été reçue par le serveur, de la valeur produite par le capteur et l'identifiant de ce dernier.

- La table **devices** contient la liste des appareils connectés. Un appareil connecté est composé d'un identifiant, d'un nom, d'un booléen qui indique s'il est activé ou non, d'un booléen qui indique s'il est classé en tant que favori, de l'identifiant de la pièce dans laquelle il se situe et si disponible, l'adresse utilisable pour contacter l'appareil depuis le serveur.
- La table **events** contient la liste des liaisons entre les capteurs et les appareils connectés. Un événement est composé d'un identifiant, de l'identifiant du capteur concerné, de l'identifiant de l'appareil concerné, de l'action à effectuer sur l'appareil (activer / désactiver), de la valeur qui sert de référence pour savoir si l'événement doit être déclenché ou non et d'un booléen qui indique si ce déclenchement doit être réalisé lorsque la valeur reçue se situe au-dessus ou en dessous de la valeur de référence.

3.5 Structure des points d'accès

Pour communiquer avec le serveur, le client doit construire et envoyer une requête HTTP. Celle-ci doit contenir le nom de la méthode (GET, PUT, POST, DELETE) et l'URL de la destination. Des informations supplémentaires peuvent être placées dans le header et dans le body lorsque nécessaire.

Le serveur met à disposition des points d'accès pour manipuler les ressources. Le tableau suivant contient des exemples de la structure des adresses pour la manipulation des capteurs :

GET	/sensor	Obtenir les informations de tous les capteurs
GET	/sensor/{id}	Obtenir les informations du capteur qui possède l'identifiant {id}
POST	/sensor	Créer un nouveau capteur
PUT	/sensor/{id}	Modifier le capteur qui possède l'identifiant {id}
DELETE	/sensor/{id}	Supprimer le capteur qui possède l'identifiant {id}

La structure de ces cinq points d'accès est uniforme entre les différents types de ressources. Des accès supplémentaires peuvent être créés selon les besoins. Par exemple :

GET	/sensor/unassigned	Obtenir les capteurs qui ne sont pas assignés à une pièce
POST	/sensor/{id}/value	Enregistrer une valeur pour le capteur qui possède l'identifiant {id}
GET	/sensor/{id}/value	Obtenir la dernière valeur pour le capteur qui possède l'identifiant {id}

Pour faciliter la communication des accès disponibles aux membres du groupe, l'outil Swagger a été installé sur le serveur. Lorsque ce dernier est lancé, des informations sont disponibles sur l'adresse `/swagger-ui.html`.

The screenshot shows the Swagger UI interface. At the top, there is a header for the **device-controller** endpoint, which is described as a Device Controller. Below this, a list of API endpoints is displayed, each with a method (GET, POST, PUT, DELETE) and a path. The endpoints are color-coded: GET /device getAllDevices (blue), POST /device newDevice (green), GET /device/{id} getDeviceById (light blue), PUT /device/{id} updateDevice (orange), DELETE /device/{id} deleteDevice (red), and GET /device/unassigned getUnassignedDevice (light blue). Below this section, there are links to other controllers: **event-controller** (Event Controller), **room-controller** (Room Controller), **sensor-controller** (Sensor Controller), and **site-controller** (Site Controller), each with a right-pointing arrow. At the bottom, there is a link to **Models** with a right-pointing arrow.

Figure 8: Représentation de Swagger

Cette page met à disposition la liste de tous les points d'accès, ainsi que des précisions sur la structure des informations à manipuler et un outil permettant d'exécuter les requêtes.

4 Développement mobile

Cette section présente et décrit les choix effectués lors de la réalisation de l'application, une description globale des fonctionnalités qui y figurent ainsi qu'une description de la conception de son interface graphique.

4.1 Choix de la technologie

Lors des cours théoriques donnés par Mr.Dutrieux, Android et React Native ont été introduits. Cependant, de par des questions d'affinités, d'apprentissage et de dépendances envers les programmes de développement, le choix du langage s'est penché du côté de Flutter.

Flutter est une technologie grandissante développée par Google et qui, par opposition à ses concurrents, permet une portabilité maximale; à la fois sur iOS, Android, web et en bêta pour les plateformes bureau.

De plus, ce dernier propose diverses fonctionnalités garantissant un développement d'application plus aisé. Parmi ces fonctionnalités, on retrouve la possibilité d'effectuer une recompilation rapide de l'application ce qui offre un certain confort de développement.

En comparaison au développement Android, Flutter présente un gros désavantage. En effet, ce dernier ne contient pas d'outils qui permettent la création d'interfaces utilisateurs par simple *drag and drop*. Ce qui, dans le cas présent, représente une charge de travail supplémentaire pour le développeur.

Enfin, la jeunesse de Flutter représente un autre désavantage non négligeable. En effet, par rapport aux alternatives, il reste un SDK en début de vie, ce qui explique son manque de fonctionnalités. Il faut tout de même noter que la communauté est, elle, très active et offre pléthore de packages afin de combler les éventuelles faiblesses initiales du SDK.

4.2 Description de l'application

Cette section s'attelle à décrire la structure de l'application, les choix inhérents à chacune de ses fonctionnalités ainsi que les possibles améliorations auxquelles elle pourrait être sujette.

Pour rappel, les fonctionnalités attendues pour cette application sont les suivantes :

- La réception des données propres à chaque station météo.
- L'affichage de chacune de ces données.
- L'ajout de deux sources de données supplémentaires.
- L'ajout de deux interactions avec les appareils connectés.
- La possibilité de manager l'ensemble des appareils connectés liés à l'application.

De plus, afin d'offrir un plus grand spectre applicatif à l'utilisateur, et potentiellement professionnaliser l'application, le groupe a fait le choix d'ajouter une couche de personnalisation supplémentaire à l'application en y introduisant le management des sites. Ces sites représentent alors un bâtiment unique et donc un ensemble de pièces.

4.2.1 Segmentation du code

Pour faciliter le travail en groupe et la relecture du code, une structure claire a été pensée dès le début du développement. Cette structure figure sur le schéma ci-dessous et se compose comme suit :

- **components** (à la racine) : contient les widgets utilisables et utilisés dans toute l'application. Ils sont donc génériques.
- **models** : contient les classes modèles utilisées dans toute l'application.
- **styles** : contient les styles de l'application (couleurs, styles de texte, ...).
- **screens** : contient un dossier pour chaque écran de l'application. Chacun de ceux-ci comporte un fichier pour l'écran dans son entièreté, qui, lui, fait appel aux divers composants issus du sous-dossier *components*. Il y a aussi un fichier *strings.dart* pour lister à un seul endroit les différents textes de chaque page.
- **services.dart** : contient les différentes méthodes permettant d'interagir avec l'API.
- **utils.dart** : contient différentes méthodes utilitaires.
- **main.dart** : contient la méthode principale et les routes permettant d'accéder aux différents écrans.

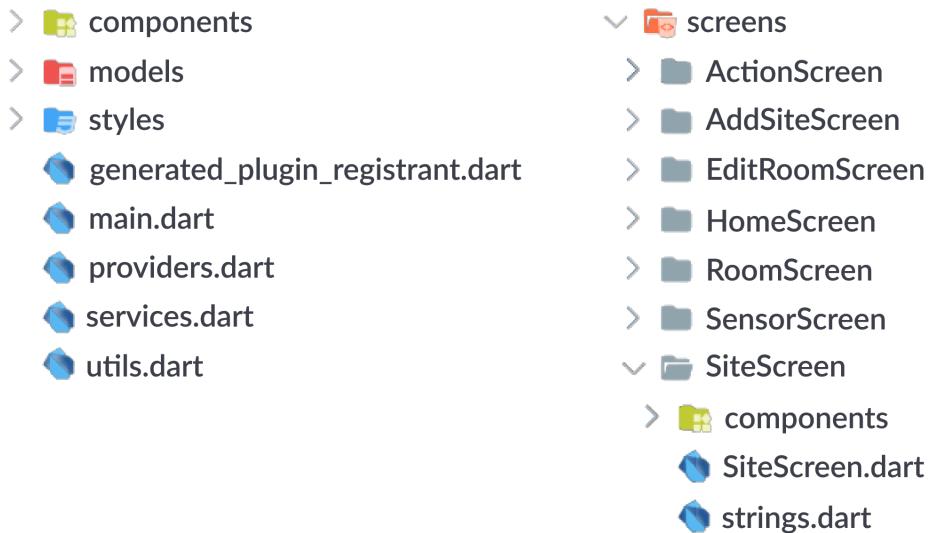


Figure 9: Structure globale de l'application

Cette structure a donc permis de segmenter le code en plusieurs fichiers. Une autre découpe tout aussi importante est celle relative à la logique et à l'interface utilisateur (UI).

Bien qu'un effort ait été produit, cette séparation n'est pas complète. Le code pourrait donc être amélioré en séparant correctement ces deux couches pour une meilleure maintenabilité, tout en rendant chacune de ces parties plus indépendantes l'une de l'autre.

Les sous-sections suivantes présentent individuellement les parties de l'application d'une manière hiérarchique en commençant par les sites, les pièces et finalement les *sensors* et *devices*.

4.2.2 Partie site

Cette partie de l'application présente les différents sites ajoutés par l'utilisateur. Ceux-ci sont triés automatiquement grâce à leur statut de favori, celui-ci étant modifiable grâce à l'icône propre à chaque site. Il est possible de trier les sites sur base de leur nom grâce à la barre de recherche prévue à cet effet. De plus, le nombre de pièces associées à chacun de ces sites est visible sur leur carte respective et l'utilisateur a la possibilité de rafraîchir la page en *swiping* vers le bas.

Grâce au menu se trouvant sur le haut de l'écran, l'utilisateur a la possibilité de soit créer un nouveau site, soit de passer en mode édition, ce qui permet d'en modifier un préexistant ou de le supprimer.

Sur l'écran d'édition des sites, les attributs propres à ceux-ci peuvent être modifiés. Parmi ces derniers, on retrouve notamment le nom du site, son illustration ainsi que l'ensemble des pièces qui le constituent. Il y a possibilité d'ajouter mais aussi de supprimer des pièces associées au site. Une confirmation est demandée à l'utilisateur dans le cas d'une suppression de pièce initialement présente lors de l'édition.

L'ensemble de ces fonctionnalités permet à l'utilisateur de faire du full CRUD (*Create, Read, Update, Delete*) sur les sites.

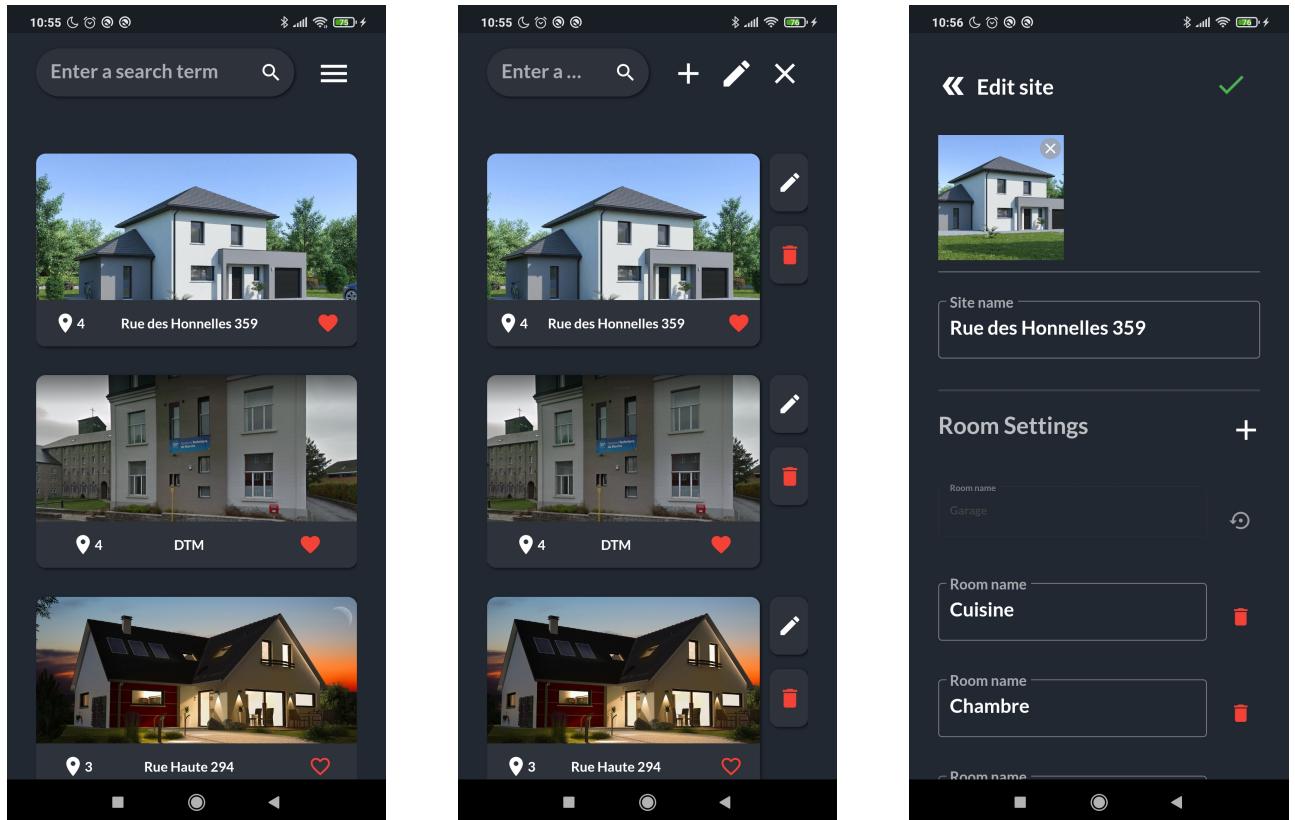


Figure 10: Illustration de la partie *Site* de l'application

4.2.3 Partie room

Cette partie de l'application présente les multiples pièces ajoutées par l'utilisateur. Celles-ci présentent des fonctionnalités similaires à celles propres aux sites. De plus, chacune des cartes présente une icône représentant la pièce, le nombre de sensors et devices connectés ainsi qu'un espace pour une éventuelle description. L'utilisateur a également la possibilité de *swipe* horizontalement pour passer facilement d'un site à l'autre sans pour autant devoir repasser par la page précédente. À cet effet, un indicateur est présent en haut à droite de l'écran et permet de se situer dans les différents sites.

Quant au menu d'édition, celui-ci offre la possibilité de modifier le nom de la pièce, son illustration, sa description et son icône ainsi que l'ensemble des *sensors* et *devices* qui la constituent. Une légende présente sur la page grâce à un bouton d'information permet de déterminer l'appartenance des appareils à une pièce donnée.

Le full CRUD est donc également disponible, même si une partie de celui-ci (*Create et Delete*) est disponible grâce au formulaire de modification du site.

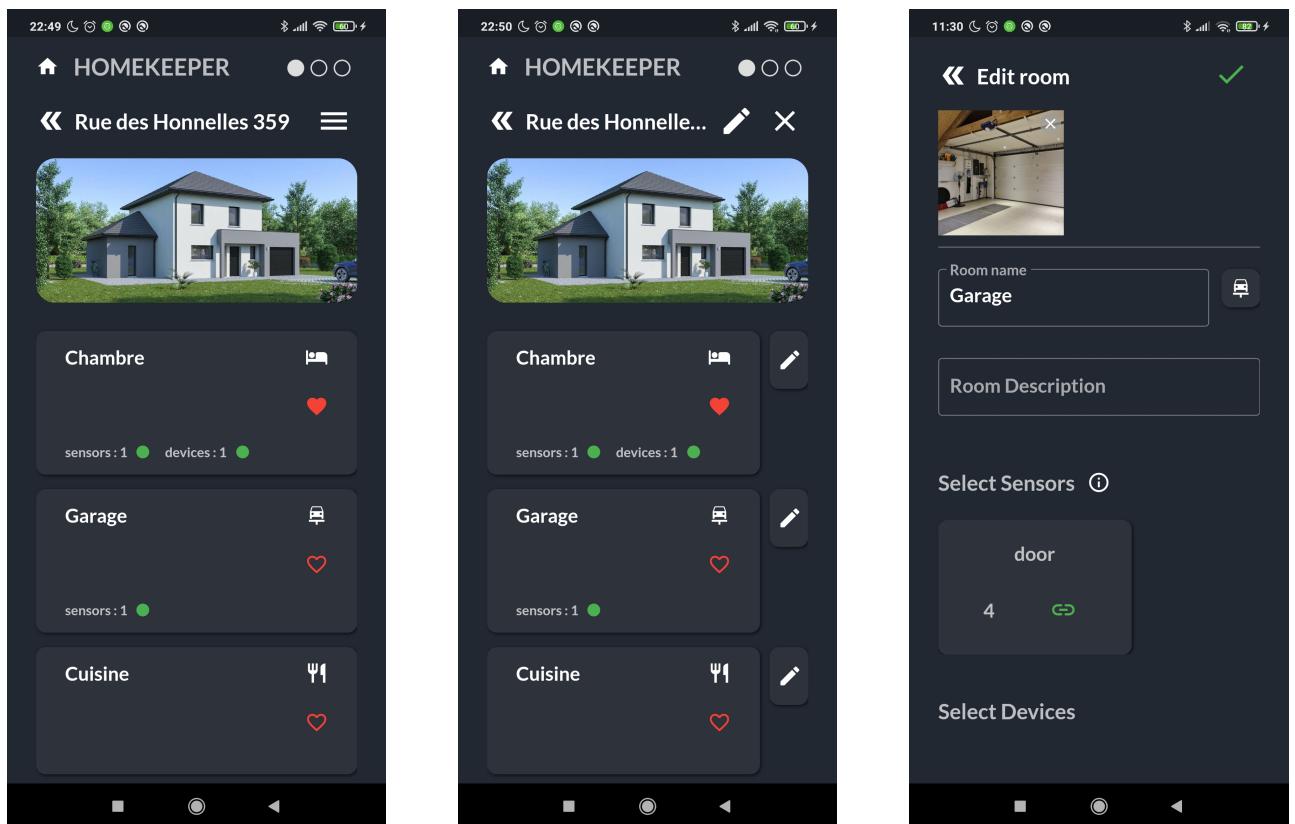


Figure 11: Illustration de la partie *Room* de l'application

4.2.4 Partie sensor et device

Cette partie de l'application présente les divers *sensors et devices* d'une pièce choisie par l'utilisateur. Le *topic* et la dernière valeur reçue sont visibles pour chaque *sensor*. Il est possible de cliquer sur la carte d'un sensor pour afficher les différents événements liés à ce dernier.

Quant aux *devices*, c'est leur nom et id qui sont affichés. De plus, il est possible d'allumer ou éteindre un *device* spécifique grâce au bouton prévu à cet effet.

Similairement aux pages précédentes, la fonctionnalité de *swipe* est également disponible et permet cette fois-ci de naviguer parmi les multiples pièces du site sélectionné. De plus, grâce au bouton d'informations situé à gauche du menu, un *pop-up* d'informations est disponible afin de guider les utilisateurs.

Quant au bouton alarme présent dans le menu, celui-ci permet d'accéder à la page de création des événements. Celle-ci sera détaillée dans la section suivante.

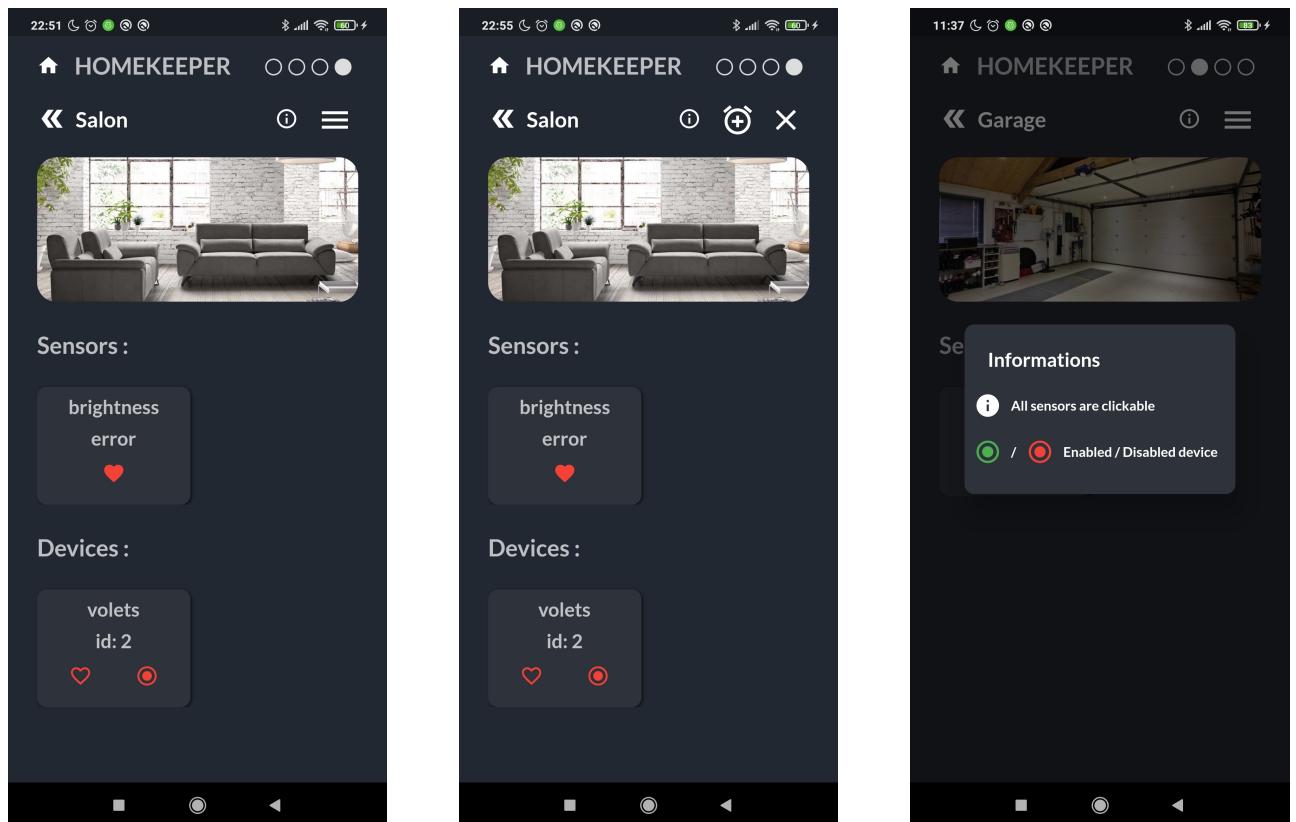


Figure 12: Illustration de la partie *Sensor et Device* de l'application

4.2.5 Partie Event

Sur les illustrations ci-dessous, la première page représente le formulaire de création des événements. Celui-ci propose à l'utilisateur de sélectionner un *sensor* ainsi qu'une valeur associée à celui-ci qui déterminera l'activation de l'événement. Il y a aussi possibilité de sélectionner un *device* et une action à réaliser dans le cas où l'événement se produit.

La seconde page, quant à elle, affiche les différents événements liés au *sensor* sélectionné. L'utilisateur peut également retirer un *event* particulier grâce à l'icône de suppression.

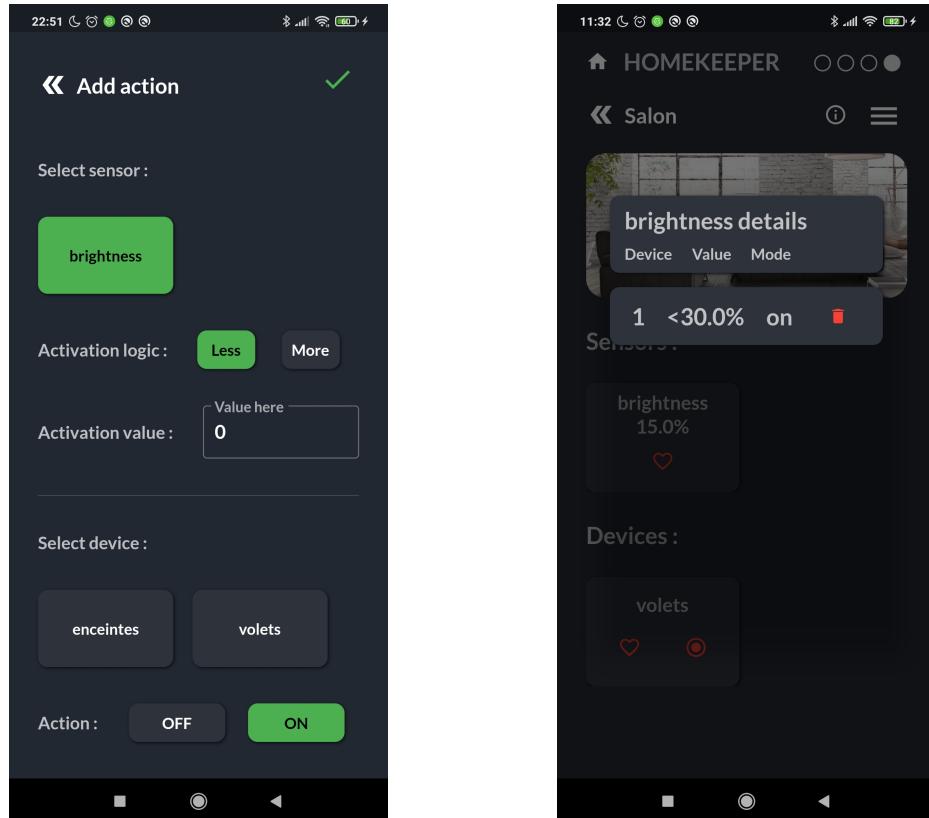


Figure 13: Illustration de la partie *Event* de l'application

5 Possibilités d'améliorations

Diverses fonctionnalités ont initialement été pensées pour être ajoutées au projet. Cependant, par faute de temps et de moyens, celles-ci n'ont pas pu voir le jour.

Les sections suivantes présentent une liste non exhaustive de ces fonctionnalités.

5.1 Ajout des graphes

Il a été pensé initialement de permettre à l'utilisateur de pouvoir visualiser un historique des données de certains *sensors* sous forme de graphes. C'est notamment une des idées qui a principalement justifié la mise en place d'une base de données en local. Cette fonctionnalité a malheureusement dû être abandonnée par manque de temps. Elle n'est cependant pas à écarter. L'ajout d'outils de visualisation tel que celui-ci peut apporter un certain confort d'utilisation et représente une réelle plus-value au projet.

5.2 Meilleure segmentation du code

Comme soulevé plus haut, la segmentation du code de l'application mobile n'est pas optimale. De ce fait, si l'application vient à grandir et à prendre de l'ampleur, il vaudra mieux *refactor* le code pour effectuer complètement cette séparation. Encore peu habitué au développement sous Flutter, l'ensemble des bonnes pratiques à suivre pour permettre d'effectuer cette séparation de manière simple et efficace n'est pas encore complètement acquis. Un *refactor* complet du code est nécessaire pour garantir une application durable et maintenable.

5.3 Authentification

L'application fonctionne actuellement sans mécanisme d'authentification. Il serait intéressant de l'ajouter afin de renforcer la sécurité globale. En outre, cela permettrait également de mettre en place une personnalisation de l'application propre à chaque utilisateur. Pour la réalisation de cette fonctionnalité, il est nécessaire d'effectuer des modifications dans l'API et dans la base de données.

5.4 Notifications

Le projet devait initialement être accompagné de notifications dont le rôle était d'avertir l'utilisateur lors de l'activation d'un événement ou d'une action sur un device. Cependant, l'implémentation de ces notifications de type push nécessite l'ajout et la configuration d'un serveur Firebase. Cette configuration étant relativement compliquée dû à la présence d'une API interne, cette idée a finalement été écartée, faute de temps. De même, le groupe a jugé que l'ajout des notifications non push dans l'application n'apportaient pas une plus-value suffisante pour justifier leur développement.

5.5 Implémentation concrète du MQTT

Une potentielle amélioration pour le futur de l'application serait d'intégrer une implémentation concrète du MQTT. En effet, ce dernier propose tout un ensemble de fonctionnalité, notamment la récupération en direct des données de l'ensemble des capteurs connectés. Mais aussi l'ajout de filtres ainsi que la création de topics privés afin de fournir une meilleure gestion des données envoyées et reçues pour chaque utilisateur.

L'utilisateur aurait ensuite la possibilité de passer du mode initial (avec traitement par la base de données) au mode MQTT directement depuis l'application et en adéquation avec ses préférences.

Ainsi, pour implémenter une telle fonctionnalité, il faudrait que l'application mobile intègre un client MQTT qui écouterait en permanence le broker pour les mises à jour des senseurs.

5.6 Sécurité

Il existe plusieurs améliorations potentielles pour la sécurité du projet :

- Pour le serveur web Flask il faudrait que les communications soient chiffrées en HTTPS. De plus, il peut être nécessaire d'implémenter une authentification afin que seules les personnes autorisées aient le droit d'interagir avec les différents objets.
- En ce qui concerne le MQTT, il pourrait bénéficier d'un chiffrement de sa communication avec TLS, ainsi que d'une vérification des utilisateurs. En effet, MQTT supporte l'ajout d'un certificat électronique et la gestion des clients sur le broker. Pour implémenter ces fonctionnalités, il faudrait modifier la configuration du broker pour qu'il utilise un certificat et qu'il conserve les informations liées aux utilisateurs.
- Comme dans tout projet nécessitant une base de données, il est important de sécuriser les accès à cette dernière. Pour ce faire, il faudrait utiliser une politique de mots de passe fort et limiter les interactions que peuvent avoir les différents utilisateurs avec les tables. Par exemple, seul le compte "API" aurait le droit d'effectuer les opérations CRUD.

5.7 QR code

Une des principales contraintes du projet concernant la partie IOT, et plus précisément les *sensors et devices*, est le manque d'un moyen d'ajouter dynamiquement ces derniers au système. A cette fin, des QR codes contenant les informations de chaque appareil ainsi que les informations de liaison avec les divers composants du projet pourraient être ultérieurement intégrés. Parmi ces informations se trouveraient l'identifiant du capteur, le type de retour et l'url où se dernier peut être contacté.

Ainsi, une fois le QR code scanné, l'appareil serait directement lié à l'ensemble du projet.