

Rapport d'élève Ingénieur Projet de troisième année

Filière : Sécurité et réseaux

SMARTHOUSE

Présenté par : KHEDHAOURIA Eliès & MARCELET Paul

Responsable Isima : Monsieur Alexandre GUITTON

Date de soutenance : 02/07/2024

Campus des Cézeaux . 1 rue de la Chébarde . TSA 60125 . 63178 Aubière CEDEX

Table des matières

1	Introduction	1
2	Contexte du Projet	2
2.1	Analyse du besoin et définition des objectifs	2
2.2	Organisation de la conception à la création	2
3	Conception et Implémentation	5
3.1	Infrastructure et Environnement de Développement	5
3.1.1	Simulation du serveur et architecture réseau	6
3.1.2	Modélisation et Simulation d'une Maison Connectée	9
3.2	Mise en place d'une API Web	12
3.2.1	Architecture logicielle de l'API et choix technologiques	13
3.2.2	Tests de l'API avec Postman	18
3.3	Surveillance des données avec une interface graphique	23
3.3.1	Architecture logicielle de l'application SmartHouse Monitoring	23
3.3.2	Tests réalisée avec SmartHouse Monitoring	25
4	Résultats et Discussion	27
4.1	Situation à la fin de l'étude	27
4.2	Analyse des résultats obtenus	28
5	Conclusion	29
5.1	Limites et améliorations possibles	30
A	Annexes	31
A.1	Lexique	31
A.2	Bibliographie	31
A.3	Webographie	31

Table des figures

2.1	Diagramme de Gantt Prévisionnel	4
2.2	Diagramme de Gantt réel	4
3.1	Architecture de l'infrastructure fonctionnelle	5
3.2	Sécurisation de mosquitto	7
3.3	Authentification effectuée: Émission et réception effectuée avec succès	7
3.4	Authentification erronée: Émission et réception non fonctionnel	8
3.5	Exemple de données JSON simples à envoyer	9
3.6	Architecture logicielle proposée de notre simulateur de maisons connectées	12
3.7	Diagramme UML du simulateur de maisons connectées	13
3.8	Vérification de la création de la base de données	15
3.9	Association d'une Table Maison de Smarthouse à ses séries de données temporelles collectées	16
3.10	Architecture de SmartHomeAPI	18
3.11	Requête d'enregistrement d'un propriétaire associable à sa maison	19
3.12	Réponse de SmartHomeAPI après enregistrement réussi	19
3.13	Champs à renseigner au sein de HomeSimulation : main.py	19
3.14	Envoie de 2 séries de données au serveur: 172.20.10.5	20
3.15	Requête de récupération de la dernière donnée acquise	20
3.16	Réponse de l'API de la dernière données acquise	21
3.17	Requête de récupération des données acquises par capteur d'humidité	21
3.18	Réponse de l'API des données acquises par capteur d'humidité	22
3.19	Architecture des classes de SmartHouse Monitoring	24
3.20	Interface utilisateur réalisée	25
3.21	Communication entre SmartHouse Monitoring et SmartHouseAPI	26

Résumé

Dans le cadre de notre projet de fin d'études, nous avons conçu et développé un **système de maison intelligente** capable de transmettre **des données en temps réel de manière sécurisée** vers un serveur distant. L'objectif principal est de mettre en place un **système de monitoring avancé**, permettant à un propriétaire de superviser et d'analyser les données générées par ses équipements connectés.

Pour garantir **l'intégrité et la confidentialité des échanges**, nous avons implémenté une **communication sécurisée basée sur des certificats SSL/TLS** et le protocole **MQTTs**, assurant ainsi une transmission chiffrée et authentifiée entre la maison et le serveur.

Les données collectées par les capteurs sont stockées dans une **base de données à série temporelle** (InfluxDB), spécialement optimisée pour le traitement et l'analyse de données en flux continu.

Une **API centralisée**, développée en **Laravel**, a été mise en place afin de :

- **gérer la création des propriétaires** et l'association sécurisée de leurs équipements.
- **automatiser la génération et la signature des certificats** pour garantir une authentification fiable.
- **offrir une interface d'accès aux données**, permettant aux utilisateurs de récupérer **des données en temps réel ou historiques**, selon différents filtres appliqués à la base de données.

Enfin, une **interface graphique interactive**, développée en **Qt**, permet aux utilisateurs de **visualiser les données en temps réel** sous forme de **graphiques dynamiques**. Cette interface interagit directement avec l'API afin de récupérer et d'afficher **les données filtrées**, qu'elles soient en temps réel ou issues d'une période spécifique dans le passé.

Ce projet intègre **des technologies modernes et des concepts avancés en sécurité, IoT, gestion des bases de données et visualisation de données en temps réel**, assurant ainsi une **infrastructure robuste, fiable et évolutive**.

Abstract

As part of our final-year engineering project, we designed and developed a **smart home system** capable of securely transmitting **real-time data** to a remote server. The main objective is to implement an **advanced monitoring system** that allows a homeowner to monitor and analyze data generated by their connected devices.

To ensure **data integrity and confidentiality**, we implemented a **secure communication protocol based on SSL/TLS certificates** and the **MQTTs protocol**, providing encrypted and authenticated communication between the home and the server.

Sensor data is stored in a **time-series database** (InfluxDB), optimized for real-time data processing and analysis.

A **centralized API**, developed in **Laravel**, has been implemented to:

- **Manage the creation of homeowners** and the secure association of their devices.
- **Automate the generation and signing of certificates** to ensure reliable authentication.
- **Provide a data access interface**, allowing users to retrieve **real-time or historical data** based on various filters applied to the database.

Finally, an **interactive graphical interface**, developed in **Qt**, allows users to **visualize real-time data using dynamic graphs**. This interface directly interacts with the API to retrieve and display **filtered data**, whether in real-time or from a specific historical period.

This project integrates **modern technologies and advanced concepts in security, IoT, database management, and real-time data visualization**, ensuring a **robust, reliable, and scalable infrastructure**.

Chapter 1

Introduction

La **domotique** représente aujourd’hui un enjeu majeur dans le domaine des innovations technologiques. Avec l’essor des **maisons connectées** et des **IOTs**, les utilisateurs peuvent désormais **surveiller** et **contrôler** leur domicile à distance, leur assurant ainsi une amélioration significative en termes de **sécurité**, **d’efficacité énergétique** et de **confort**. Cette évolution s’inscrit dans un contexte plus large dans lequel l’automatisation et la connectivité jouent un rôle crucial dans notre quotidien.

Le **monitoring à distance** des équipements d’une maison constitue un axe fondamental de la domotique moderne. Il permet aux propriétaires d’obtenir une **vue globale de l’état de leur habitation en temps réel**. Cela joue un rôle crucial dans plusieurs domaines:

- il permet de **sécuriser** un domicile, permettant par exemple la détection d’intrusion ainsi que la prévention des cambriolages
- il **optimise énergiquement** le domicile, par le biais de l’automatisation des objets connectés, en fonction d’horaires programmés, afin d’optimiser la consommation d’énergie.
- et il permet enfin **un confort et un contrôle à distance**, offrant aux propriétaires la possibilité d’activer ou de désactiver certains dispositifs sans être physiquement présent.

Si ces avancées technologiques offrent des opportunités considérables, elles soulèvent néanmoins une problématique critique: **la sécurisation des dispositifs IoTs et du transfert des données**. Aujourd’hui, de nombreux objets connectés sont déployés avec des failles de sécurité importantes souvent négligées par les fabricants et les utilisateurs. Des outils comme **Shodan**, un moteur de recherche spécialisé dans l’identification des appareils connectés exposés sur Internet, mettent en évidence la vulnérabilité de nombreux systèmes IoTs accessibles sans protection adéquate. Cette situation constitue un risque majeur, rendant possible des cyberattaques capables de compromettre **l’intégrité et la confidentialité** des données échangées.

Ce rapport décrit une architecture, solution à cette problématique en explorant l’une des applications majeures de la domotique: **le monitoring à distance des capteurs d’une maison connectée, ainsi que l’établissement d’une communication sécurisée et authentifiée entre celle-ci et un serveur distant**. L’objectif est de permettre aux utilisateurs de récupérer des **données en temps réel**, issues de capteurs de leur domicile tout en garantissant **une transmission chiffrée** afin de protéger les échanges contre d’éventuelles interceptions malveillantes. C’est à partir de cela que l’on peut définir une problématique à laquelle la solution doit répondre: **Comment développer un système de monitoring en temps réel pour une maison connectée, garantissant la sécurité de la transmission des données tout en renforçant l’authentification des équipements IoTs ?**

Chapter 2

Contexte du Projet

2.1 Analyse du besoin et définition des objectifs

Lorsqu'un résident quitte son domicile, il peut être préoccupé par l'état de sa maison, se demandant si une lumière a bien été éteinte ou si une fenêtre a été correctement fermée. Ces préoccupations sont légitimes, d'autant plus que les statistiques récentes indiquent une augmentation des cambriolages des logements en France. En effet au 30 juin 2024, les forces de sécurité ont enregistré une hausse de 4% des cambriolages de logements sur les douze derniers mois¹. Cette tendance souligne la nécessité de renforcer les dispositifs de sécurité pour protéger les habitations.

Parallèlement, la prolifération des dispositifs IoT dans les foyers pose des défis non négligeables en matière de cybersécurité. Bien que ces technologies offrent des avantages indéniables en termes de confort et d'efficacité énergétique, elles peuvent constituer des points d'entrée pour les cybercriminels si elles ne sont pas correctement sécurisées. Il est ainsi essentiel de garantir que seuls les propriétaires autorisés aient accès à ces dispositifs et que les données transmises soient protégées contre toute altération ou interception malveillante.

Fâche à ces constats, notre projet vise à développer une solution permettant la transmission sécurisée issues de capteurs IoTs d'une maison vers un serveur distant. Cette solution devra assurer l'authentification des dispositifs, garantir l'intégrité ainsi que la confidentialité des données, et ainsi permettre aux propriétaires de surveiller à distance l'état de leur domicile en temps réel.

2.2 Organisation de la conception à la création

Dans le cadre du développement de ce projet, nous avons adopté une approche structurée en plusieurs phases allant de la simulation initiale de l'environnement domotique jusqu'à la mise en place d'une infrastructure de communication sécurisée et fiable.

Phase 1: Simulation de l'environnement domotique et émission des données

Avant de mettre en place l'architecture réseau et serveur, nous avons débuté par la simulation logicielle de la maison connectée, en programmant un environnement permettant la génération de données de divers capteurs (température, humidité, lumière...). Cette simulation développée en **Python**, modélise une maison contenant divers équipements IoTs et capteurs, émettant des séries de données à temps réel. L'objectif principal de cette étape était de tester l'envoi de séries de données, à intervalle régulier, au sein d'un serveur distant (Phase suivante), en utilisant le protocole de communication **MQTT**. À ce

1. source: <https://www.interieur.gouv.fr/actualites/actualites-du-ministere/analyse-conjoncturelle-des-crimes-et-delits-enregistres-par-la-police-et-la-gendarmerie> et <https://mobile.interieur.gouv.fr/Interstats/Actualites/Info-Rapide-n-43-La-delinquance-enregistree-par-la-police-et-la-gendarmerie>

moment là, la transmission s'effectuait sans authentification ni chiffrement, nous permettant ainsi, de valider l'intégralité du transport, la réception au serveur et d'évaluer aussi les performances du protocole.

Phase 2: Mise en place de l'infrastructure serveur

Une fois la simulation fonctionnelle, nous avons déployé une **infrastructure serveur** sous une machine virtuelle **Ubuntu**, utilisant l'hyperviseur **Virtualbox** avec un accès par pont en configuration réseau. Ce serveur assure le rôle de récepteur des données envoyées par la maison connectée.

Afin de permettre la réception ainsi que le stockage des données, nous avons mis en place plusieurs composants essentiels:

- **Mosquitto**: Un broker **MQTT** permettant la gestion des messages entre les maisons connectées et le serveur aux divers topics.
- **InfluxDB**: Une **base de données à séries temporelles**, choisie pour sa capacité à stocker et traiter efficacement des flux de données en temps réel.
- **Telegraf**: Un agent de collecte des données utilisé pour formaliser et structurer les données reçues depuis le **Broker** avant leur insertion dans la base de données.

À la fin de cette étape, après configuration des composants, l'infrastructure était fonctionnelle, mais vulnérable : les données envoyées par les capteurs n'étaient pas protégées et n'importe quel utilisateur pouvait intercepter ou publier des messages MQTT sur le serveur, compromettant ainsi l'intégrité du système.

Phase 3: Sécurisation des échanges et authentifications des Maisons

Afin de garantir l'**authenticité des émetteurs** et de protéger les données échangées, nous avons implanté une **authentification basée sur des certificats SSL/TLS** pour le protocole MQTT. Cette sécurisation repose sur l'**utilisation de certificats clients** générés par une autorité de certification interne au serveur, exigeant une **authentification mutuelle** entre la maison et le serveur pour toute communication MQTT ainsi que le **chiffrement des échanges** grâce à TLS qui empêche toute interception des données transmises.

Ce mécanisme permet ainsi de **garantir l'identité des dispositifs connectés** et d'empêcher ainsi toute interception des données par un acteur non autorisé.

Phase 4 : Développement d'une API centralisant l'accès aux données

Afin de faciliter l'accès aux données, d'éviter une exposition directe du broker MQTT et aussi de permettre par la suite la création d'interfaces fonctionnant sur divers plateformes, nous avons développé une **API Rest sous Laravel**, jouant deux rôles importants:

- **Gestion de propriétaires et authentification**: l'API permet la **création de propriétaires** assurant une authentification unique et sécurisée. Un nouvel utilisateur peut en effet s'enregistrer en tant que propriétaire, l'API lui générera ainsi un **token unique**, qui servira **d'identifiant primaire** pour toutes les interactions futures entre les propriétés de l'utilisateur et le système. Elle générera de plus, un **certificat client signé** par l'autorité de certification accompagné d'une clé privée, permettant ainsi une authentification sécurisée lors des échanges avec le broker MQTT.
- **API de relais et récupération des données**: l'API agit également en tant que **relais sécurisé** entre les propriétaires et les données stockées au sein du serveur. L'API est capable d'extraire les séries temporelles stockées au sein de **InfluxDB**, en les filtrant en fonction des critères demandés pour chaque utilisateurs (temps réel, historique...). Ainsi, les utilisateurs

peuvent accéder uniquement aux données qui leur sont destinées, garantissant l'intégrité et la confidentialité des échanges.

Phase 5: Développement d'une interface graphique permettant un affichage concret des données

Afin de permettre une visualisation claire des données issues des capteurs, nous avons eu l'idée de développer une interface graphique utilisant **Qt**. Cette interface permet aux propriétaires d'interagir avec l'**API** et d'accéder aux informations de leur maison de manière ergonomique.

Le but de l'interface QT est de récupérer les données via l'API en appliquant divers critères de filtrage, et placer les résultats affichés sous forme de graphique dynamique, afin de faciliter l'analyse ainsi que la supervision de la maison connectée.

Le choix d'utiliser Qt comme technologie, est liée à plusieurs raisons:

- **Demande élevée en entreprise:** Qt est largement utilisé dans l'industrie pour le développement d'interfaces utilisateur performantes et multiplateformes.
- **Complémentarité avec les connaissances acquises en C++:** jusqu'à présent, le programme de formation avait principalement abordé le C++ de manière théorique. Ce projet était donc une opportunité idéale pour appliquer concrètement ses connaissances en développant une interface interactive et fonctionnelle.

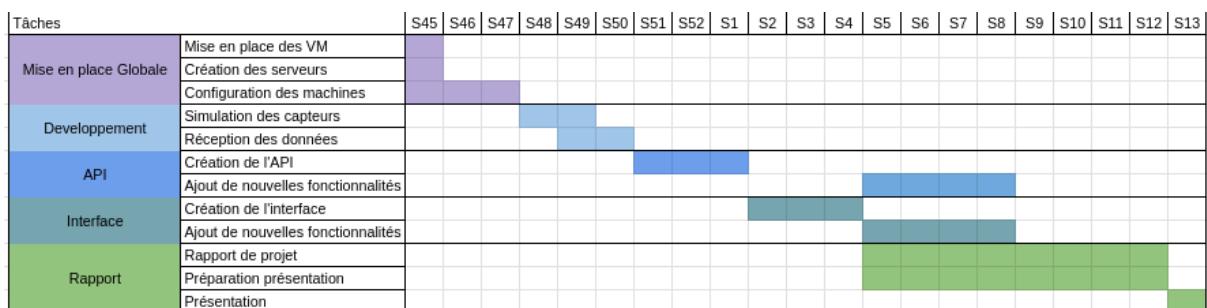


Figure 2.1 – Diagramme de Gantt Prévisionnel



Figure 2.2 – Diagramme de Gantt réel

Chapter 3

Conception et Implémentation

3.1 Infrastructure et Environnement de Développement

Cette section détaille l'ensemble de l'infrastructure mise en place, depuis la simulation logicielle d'une maison connectée jusqu'à l'infrastructure serveur dédiée. Cette dernière assure la **réception**, le **traitement** ainsi que le **stockage** sécurisé des données issues des différents capteurs IoT simulés. Nous présenterons les diverses technologies utilisées au sein de l'infrastructure, ainsi que la façon dont elles ont été intégrées afin de garantir une cohérence globale avec les objectifs initiales du projet.

Ce projet a été réalisé en **local**, sous la forme d'une **simulation globale**. Le serveur utilisé repose sur une machine virtuelle exécutant **Ubuntu Server**, configurée localement via **un accès réseau en mode pont**. Cette configuration permet au serveur d'obtenir une adresse IP dédiée sur le même réseau local que la machine hôte, facilitant ainsi une connectivité réseau directe et simplifiée avec la maison connectée simulée. Dans ce contexte, notre simulation considère que le serveur et l'émetteur sont présents au sein du même réseau local, en l'absence de solution Cloud externe.

Nous verrons que le serveur joue un rôle central au sein de cette infrastructure. En effet, il héberge l'ensemble des services essentiels au fonctionnement de la solution réceptrice: le **broker MQTT**, l'**API REST** développée sous Laravel, ainsi que l'ensemble des bases de données qu'elles soient à **séries temporelles ou relationnelles**.

Concernant l'infrastructure émettrice, afin de simuler les équipements IOTs de la **maison connectée**, nous avons utilisé le **language Python** du fait de la disponibilité étendue des bibliothèques réseau faciles à implémenter, facilitant ainsi la mise en œuvre rapide et fiable de la simulation.

Dans les sous-sections suivantes, nous détaillerons la mise en place précise de chacun des composants techniques essentiels à la réalisation de nos objectifs.

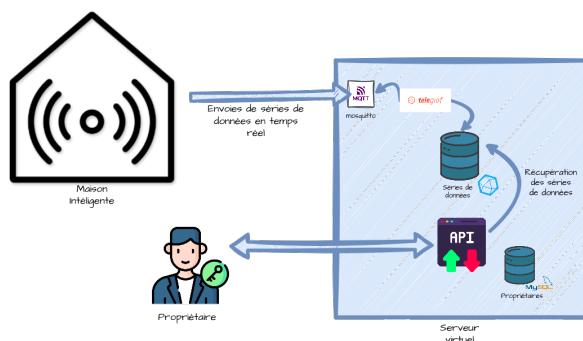


Figure 3.1 – Architecture de l'infrastructure fonctionnelle

Le schéma ci-dessus décrit l'infrastructure globale mis en place au sein de la section.

3.1.1 Simulation du serveur et architecture réseau

Déploiement d'un Broker MQTT sécurisé

Afin de garantir une transmission en temps réel des données IoTs entre la maison connectée et le serveur, nous avons trouver plus judicieux de choisir le protocole **MQTT**.

MQTT est un protocole de messagerie léger basé sur le modèle **publish/subscribe**, initialement conçu pour des environnements où la bande passante est limitée et la latence est faible. MQTT est, dans notre contexte, parfaitement adapté. En effet, le protocole a initialement été conçu afin d'être adapté aux applications IoTs, grâce à sa faible consommation de ressources, sa simplicité d'implémentation et son support pour des communications asynchrones et pouvant être sécurisées par le biais de la technologie **SSL/TLS**.

Il est important de savoir que MQTT repose sur trois éléments fondamentaux:

- **le Broker (serveur de message)**: Element central de l'architecture MQTT agissant comme un relais entre les clients acheminant les messages publiés et les abonnés appropriés. Il en existe un certain nombre, et dans le cadre de notre projet, nous avons choisi d'utiliser **Mosquitto**.
- **les Topics (Sujets de message)**: Il s'agit d'une chaîne hiérarchique permettant d'identifier une catégorie de messages, leur format est composé de "/" permettant de séparer les sous-topics. Dans le cadre de notre projet nous avons justement exploité ce système de chaînes hiérarchiques afin d'organiser d'une manière précise, l'ensemble des types d'**iots** situés dans les différentes salles des divers **maisons** des différents **propriétaires**.
- **les clients (publishers et subscribers)**: MQTT fait la distinction entre deux types de clients:
 - **les publishers (éditeurs)**, ceux qui envoient des messages à un topic spécifique, sans se soucier du destinataire, il s'agit dans le cadre de notre projet, de la **maison connectée** qui envoie des séries de données aux différents topics correspondants.
 - **les subscribers (abonnés)** ceux qui écoutent au sein d'un topic et reçoivent les messages spécifiques correspondants, dans notre cas, il s'agit des **propriétaires** des maisons.

Tel qu'expliqué, nous avons choisis de déployer **Mosquitto** au sein du serveur. Malgré le fait qu'il existe d'autres solutions telles que **HiveMQ**, **EMQX** ou **RabbitMQ MQTT**, pouvant faire office de **broker**, nous avons choisi **Mosquitto**, du fait qu'il est conçu pour être **ultra-léger**, consommer **très peu de mémoire et de CPU même sous forte charge**, et convient aussi bien **aux petits réseaux IoT qu'aux grandes infrastructures**.

De plus, **Mosquitto** permet une installation assez rapide de son service, et la configuration s'effectue via un seul fichier : **mosquitto.conf**. Enfin, il supporte des fonctionnalités de sécurité avancées que nous avons mis en place au sein de notre serveur, telles que le support **TLS/SSL** afin de chiffrer les communications.

Pour sécuriser les connexions et assurer l'authenticité des communications, nous avons implémenté MQTT sécurisé (MQTTs) basé sur SSL/TLS. Nous avons utilisé l'outil **OpenSSL** pour créer une autorité de certification interne **CA** ainsi que pour **générer et signer automatiquement** des certificats clients **client.crt** et leurs clés privées associées **client.key**. Chaque maison simulée dispose donc de trois certificats essentiels pour établir une connexion sécurisée.

Voici ainsi l'architecture de sécurisation de **Mosquitto**:

- la création d'une Autorité de Certification (CA) dédiée pour la **génération et la signature des certificats**.

- l'utilisation de certificats clients signés (certificat client et clé privée) pour chaque maison voulant envoyer des données au sein d'un serveur.
- l'utilisation du protocole MQTTs, permettant une **authentification mutuelle** et un chiffrement systématique des communications, empêchant ainsi toute interception ou injection malveillante de données.

Pour faciliter la compréhension de la sécurisation de Mosquitto, et l'intégration du protocole MQTTs au sein de l'architecture, voici un schéma explicatif:

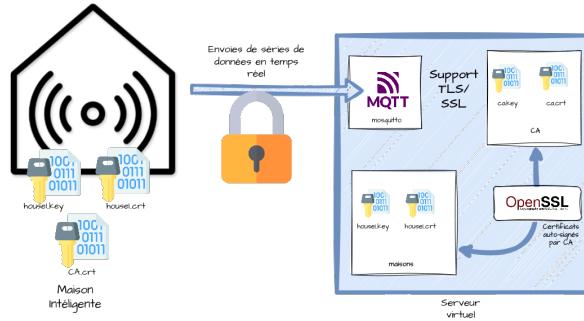


Figure 3.2 – Sécurisation de mosquitto

Afin de vérifier le fonctionnement des certificats ainsi que l'envoi de données au sein du broker mosquitto du serveur virtuel, nous avons à ce moment là testé avec les commandes **mosquitto_pub** côté client et les **logs temps réel côté serveur**¹.

Ainsi, après génération et auto-signature des certificats, les figures ci-dessous montrent ce qu'il se passe lorsque l'authentification est valide c'est à dire lorsque le **client.key** et le **client.crt** correspondent bien, et lorsque ces derniers sont invalides:

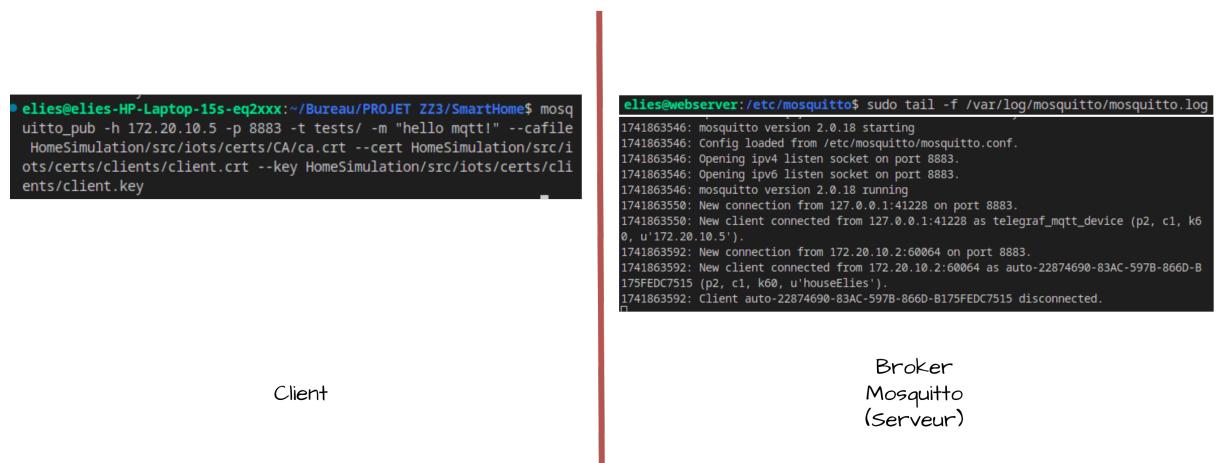


Figure 3.3 – Authentification effectuée: Émission et réception effectuée avec succès

Dans la figure ci-dessus, nous pouvons nous apercevoir (par la capture d'écran du résultat du log instantané du broker) que la connexion de l'émetteur s'est correctement déroulée, et que ce client a été correctement authentifiée par le nom d'utilisateur **houseElies**.

1. Au sein de la configuration de mosquitto, nous avons défini un chemin de log accessible en temps réel via la commande linux `tail -f /var/log/mosquitto/mosquitto.log`

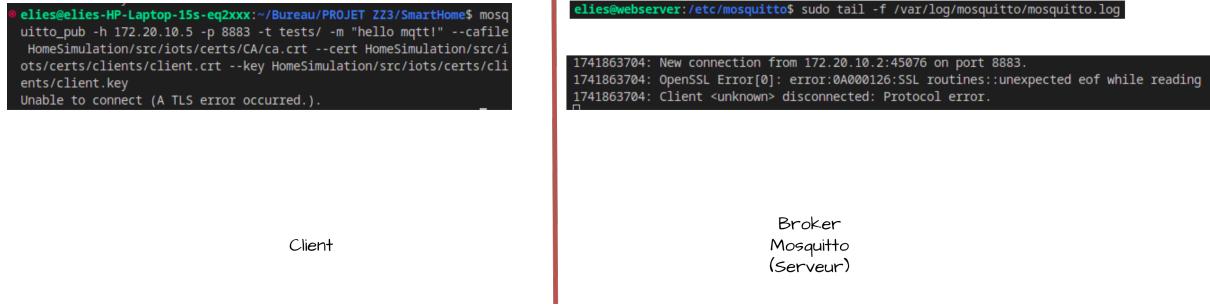


Figure 3.4 – Authentification erronée: Émission et réception non fonctionnel

À la différence de la figure précédente, afin de prouver qu'il est nécessaire de disposer des certificats clients valides, nous avons intentionnellement modifié la clé générée de manière à ce qu'elle soit erronée. Nous pouvons ainsi constater qu'il y a une erreur lors de la connexion de l'emetteur et que ce dernier n'est pas parvenu à publier, ni à être identifié par le broker.

Intégration d'une base de données à séries temporelles

L'émission sécurisée des données depuis une adresse IP distante vers le serveur étant désormais établie, l'étape suivante consiste à assurer le stockage efficace et structuré de ces informations. L'objectif de ce projet, étant non seulement de permettre l'émission de grandes séries données en temps réel, mais aussi de pouvoir interroger et analyser des données collectées à une date ou une période spécifique. C'est pourquoi, nous avons jugé indispensable d'intégrer une base de données adaptées à des contraintes temporelles.

Afin de répondre à ce problème de contraintes temporelles, nous avons utilisé une **base de donnée à séries temporelles (TSDB)**. Une base de donnée à séries temporelles est un type de base de données optimisée pour stocker et interroger des données indexées par le temps. À la différence des bases de données classiques telles que **MySQL** ou **PostgreSQL**, stockant les données sous forme de relation statiques, une **TSDB** permet de gérer des séries de données continues, optimisant les requêtes temporelles et stockant de manière efficace les données horodatées.

Une des plus connues est **InfluxDB**. Développée par **InfluxData**, cette base de données répond parfaitement au problème du fait qu'elle permet l'enregistrement de données évoluant dans le temps et facilite leur analyse. Contrairement à ses concurrents, elle permet une écriture des données rapide, ce qui est essentiel pour le temps réel. De plus, elle compresse les données de manière efficace, permettant ainsi l'optimisation de l'espace de stockage. Enfin, un des avantages clés pour notre projet, est le fait que cette base de données est capable de traiter des millions de points de données par seconde.

Elle dispose de plus d'une **API Web** permettant ainsi un accès distant des données, nous verrons par la suite que cette **API** nous sera essentiels pour accéder aux données transférées.

Cependant, la subtilité, est que **InfluxDB** ne supporte pas nativement le protocole **MQTT**, il est donc nécessaire d'utiliser un composant intermédiaire afin d'assurer la transmission des messages depuis le protocole MQTT vers la base de données.

Pour cela, nous avons intégré **Telegraf**, un agent de collecte de métrique conçu afin d'intéragir avec différents systèmes de monitoring et bases de données.

Nous avons ainsi configuré **Telegraf** pour écouter les messages publiés sur **Mosquitto** et les reformater avant de les insérer directement au sein de la base de données.

Telegraf propose deux formats principaux pour insérer des données au sein d'**InfluxDB**:

- **Line Protocol:** ce format propre à **InfluxDB** optimise le stockage ainsi que la vitesse d'écriture, il s'agit d'une syntaxe représentée sous la forme **measurement,tag=value timestamp**. Bien que très performant, ce format peut être complexe à manipuler et requière une structure rigide des données envoyées.
- **JSON:** ce format est plus lisible et universellement reconnu, facilitant le traitement et la manipulation des données. Il permet une structuration claire des informations envoyées, avec des clés explicites pour chaque valeur de mesure, ce qui facilite le débogage et l'analyse des données stockées.

Ainsi, pour des raisons de clarté et de lisibilité, nous avons opté pour le format **JSON**, cela nous facilitera l'écriture des algorithmes dédiés à l'envoi de données des maisons.

Afin d'effectuer cela, nous avons configuré le fichier **telegraf.conf** pour utiliser le plugin **mqtt_consumer**. Ce plugin va permettre à **Telegraf** de s'abonner à tout les **topics MQTT** qui nous permettront par la suite de récupérer les messages publiés au format **JSON**, afin d'**orchestrer automatiquement** la structuration des données au sein des tables **InfluxDB**, et de faire correspondre **les colonnes aux valeurs JSON** issues des clés correspondantes. Nous verrons plus en détail, dans la **sous-section suivante**, comment nous avons choisi de formaliser les données et ainsi structurer la base de données.

Le schéma récapitulatif **ci-dessous** décrit le fonctionnement de **InfluxDB** et **Telegraf** au sein de l'architecture par le biais d'un exemple simple et récapitulatif des données à envoyer:

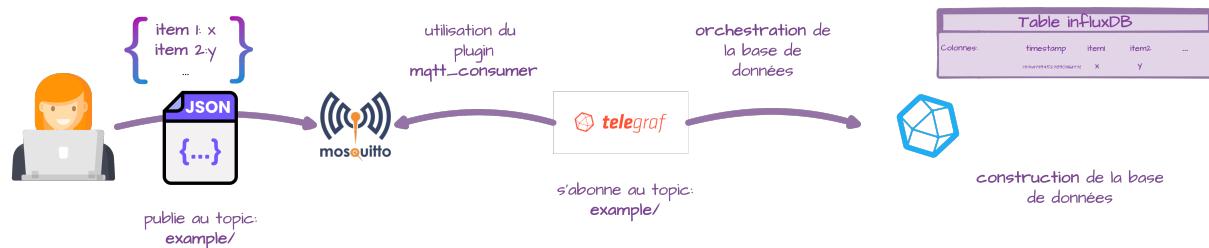


Figure 3.5 – Exemple de données **JSON simples** à envoyer

Ainsi, l'émetteur écrit un **buffer** dans lequel il insère des données formatées en **JSON**, ces données sont envoyées à travers le protocole MQTT à un topic nommé **example/**.

Ces données sont reçues au sein du broker au topic **example**, auquel s'abonne le service **telegraf** dans le but de **traduire** le format JSON reçu, afin d'**orchestrer** et créer au sein d'**InfluxDB** la table correspondante.

3.1.2 Modélisation et Simulation d'une Maison Connectée

Après avoir rendu possible l'émission ainsi que le stockage de manière sécurisée et authentifiée entre un client et un serveur via le protocole **MQTTs**, nous avons entrepris la modélisation d'une maison connectée de manière réaliste. La simulation vise à reproduire les interactions d'une habitation intelligente équipée de capteurs IoT, tout en respectant les contraintes de communication définies précédemment. L'objectif de cette simulation est d'une part de permettre l'émission de séries de données depuis une maison connectée simulée, personnalisable et extensible, utilisant le protocole MQTT, et d'autre part de mettre en place un format d'envoi de données en JSON afin de permettre le remplissage de la base de

données distante tout en identifiant chaque maison.

Structuration et formalisation des données échangées

Dans un premier temps, avant de commencer la conception des maisons simulées, il est nécessaire de réfléchir à la structure des données au format **JSON**, que nous allons mettre en place afin de permettre une gestion efficace des données recues par les IOT's de la maison.

La structuration des données repose sur une approche où chaque message transmit par la maison suit un format JSON préférable. Le but est de garantir une compatibilité optimale avec les systèmes de stockage et de permettre ainsi un affichage correctement structuré au sein de la base de donnée.

Afin de garantir tout cela, il nous a fallut insérer au sein de la donnée transmise divers élément permettant d'identifier la maison émettrice, mais aussi d'identifier quel IOT de la maison est concerné. Voici les clés que nous avons judicieusement choisi afin de garantir un stockage des données optimale au sein de la base de données:

- **token:** Le token est la clé primaire d'une donnée envoyée. En effet, elle permet d'identifier la maison connectée émettrice, c'est sur ce jeton que nous nous baserons afin qu'un propriétaire puisse rechercher sa maison. Nous pouvons voir le token tels que la clé de l'utilisateur, que seul lui dispose et qu'il ne doit partager personne. Nous verrons par la suite que cette clé sera générée par l'API et donnée au propriétaire lorsque ce dernier souhaitera enregistrer sa maison au sein de la base de donnée.
- **house_name:** permet simplement de nommer la maison, il s'agit là aussi d'une clé unique, en effet elle permettra de distinguer les différents maisons au sein des sous-topics de **houses/** (explication détaillée au paragraphe suivant).
- **device_name:** concerne l'**IOT** auquel on a capturée la donnée à envoyer (il peut s'agir d'un capteur, de l'état d'une porte d'une lumière...). Ainsi, contiendra une chaîne JSON auxquels sera stocké sont état en tant qu'information utile au propriétaire.
- **device_location:** permet de localiser l'IOT au sein de la maison, cette information pourra être utile dans le cas où l'on souhaiterait établir une cartographie de la maison au sein de l'interface graphique, et que le propriétaire puisse ainsi avoir une vue globale de sa propriété, avec l'ensemble des capteurs présent à leur localisation spécifiques.
- **type:** Cette clé permettra d'organiser chaque **IOT** en fonction de leur rôle attribué (capteur_lumière...), cela pourra ainsi permettre d'ajouter un filtre en plus dans le cas où on suppose qu'un propriétaire veuille récupérer les données issues de l'ensemble des lampes de sa maison par exemple.
- **values:** Cette clé pourra stocker un tableau contenant une ou plusieurs données, correspondant à l'état d'un **IOT** à un instant donné.

Ainsi, chaque série de données envoyées par la maison est constituée d'une ou plusieurs données au format JSON composées des clés spécifiées.

Voici un exemple de donnée envoyé par une maison à un instant donné composé uniquement d'un capteur de lumière:

```
1  {
2      "house_name": "eliesHouse",
3      "house_token": "Wdp4HCPNvC3wHnvSfSwmVknhLuHtjUkoIzG6zPno",
4      "device_name": "lamp_kit",
5      "device_id": 1,
6      "device_location": "kitchen",
7      "type": "light_sensor",
```

```

8         "values": { "state": 0 }
9     }
10

```

Disposition des Topics Un autre élément permettant la structuration des données émises par les maisons ainsi que l'organisation de ces dernières est la disposition des topics. En effet, il est important de distinguer les différents **sous-topics** du topic racine **houses**, contenant l'ensemble des maisons, qui elles-mêmes incluent dans des sous-topics les différents types d'IoT.

L'organisation des **topics** suit la hiérarchie suivante :

- **houses/** : Racine des topics, regroupant toutes les maisons connectées.
- **houses/<house_name>/** : Chaque maison possède un sous-topic dédié identifié par son nom unique.
- **houses/<house_name>/<device_type>/** : Le but est de garantir une catégorisation des équipements par type (ex: TYPE_LIGHT, TYPE_HUMIDITY...).
- **houses/<house_name>/<device_type>/<device_name>** : Le dernier niveau représente un équipement précis (par exemple lampe, capteur de température).

Voici un exemple de topics auquel l'information de l'état de la lampe de la maison est publié: **houses/eliesHouse/TYPE_LIGHT/lamp_kit**

Cette organisation permet une classification logique et structurée des messages publiés, facilitant ainsi la récupération et le traitement des données par les abonnés spécifiques aux topics concernés.

Afin de garantir cette organisation, nous avons configuré avant de passer à la création du simulateur configures telegraf. Le plugin **mqtt_consumer** permet de hiérarchiser les topics auxquels seront envoyées les données et d'établir des structures précises.

Conception de l'architecture logicielle de la simulation

La conception de l'architecture logicielle de la simulation repose sur une structure modulaire extensible et permettant d'intégrer plusieurs maisons connectées, entièrement personnalisable, pouvant contenir un très grand nombre d'appareils **IOTs**. Nous avons choisi de développer cette architecture en Python en raison de sa flexibilité et du fait qu'il contient des bibliothèques permettant l'implémentation des communications réseau via le protocole MQTT de manière simple et rapide.

Chaque maison connectée est modélisée sous forme d'une classe Python encapsulant l'ensemble des capteurs et des dispositifs présents dans l'habitation. Ces équipements sont eux-mêmes représentés par des classes spécifiques dérivées d'une classe abstraite **Device**, permettant d'uniformiser la gestion des données et des envois MQTT.

Principales composantes de l'architecture logicielle Voici les fonctionnalités que nous avons implémenté au sein de l'architecture logiciel:

- **House** : Classe représentant une **maison connectée**. Chaque instance est associée à un token unique et gère un ensemble d'équipements IoT.

- **Device** : Classe abstraite représentant un dispositif connecté (capteurs, actionneurs). Les classes spécifiques (ex: **LightSensor**, **HumiditySensor**) en héritent.
- **Main**: Cette classe agit comme un point central au sein de cette simulation, en effet, elle initialise les maisons connectées ainsi que leur capteurs, elle orchestre l'émission des données en fonction d'un intervalle défini et elle établit et maintient la connexion **MQTT** pour chaque maison.

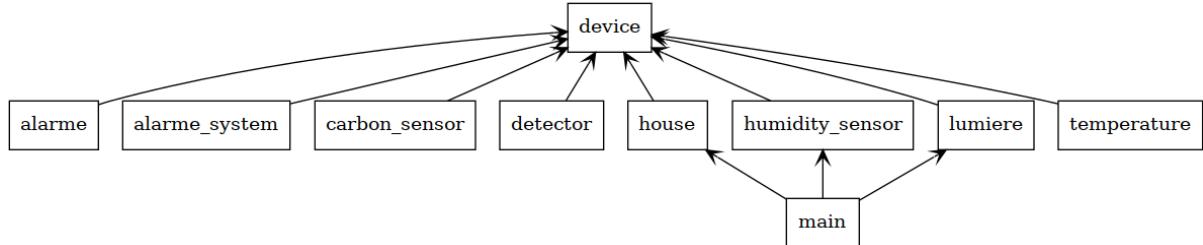


Figure 3.6 – Architecture logicielle proposée de notre simulateur de maisons connectées

Envies des données JSON à chaque classe IoT

Implémentation du Protocole MQTT Afin d'implémenter le protocole **MQTT** nous avons utilisé la bibliothèque **paho-mqtt**, qui est une implémentation légère du protocole en Python. Cette bibliothèque permet la gestion complète des communications entre les clients **MQTT** et le **broker**, en facilitant la publication ainsi que la souscription aux topics définis.

Nous avons dans un premier temps vérifié le fonctionnement de manière non sécurisé (sans SSL/TLS), afin de vérifier si l'émission des données au format conçu fonctionnait tels que prévu, avant d'ajouter les **certificats SSL/TLS** afin de permettre l'implémentation du protocole MQTTS.

Implémentation du protocole MQTTS La bibliothèque **paho-mqtt** que nous avons utilisée supporte l'intégration de certificats **SSL/TLS**. Afin d'intégrer ces certificats, la bibliothèque dispose de l'objet **mqtt.Client()** contenant une méthode **tls_set(ca_certs, certfile, keyfile)**. Cette méthode prend en paramètre le chemin du certificat de l'autorité de certification, du propriétaire ainsi que la clé privée de ce dernier.

En appelant ensuite la méthode **publish()** de la classe **mqtt.Client**, il nous sera possible d'envoyer les données en utilisant ces certificats et ainsi garantir une authentification sécurisée.

Cette implémentation garantit une transmission fiable et sécurisée des données IoT simulées, tout en respectant les bonnes pratiques d'authentification et de chiffrement des communications réseau.

Diagramme UML décrivant l'architecture des classes du simulateur Afin de résumer l'architecture des classes du simulateur, voici le diagramme UML:

3.2 Mise en place d'une API Web

Afin d'automatiser l'**authentification des utilisateurs**, d'accéder aux **données** et de permettre par la suite la **possibilité de création d'applications multiplateformes**, nous avons mis au point une **API Rest**. Cette **API Rest** est l'élément principal de l'architecture. En effet, elle constitue le cœur de l'application, jouant un rôle central dans la gestion des données acquises issues des maisons connectées. Elle permet notamment :

- L'**authentification** et l'**identification** des maisons connectées.

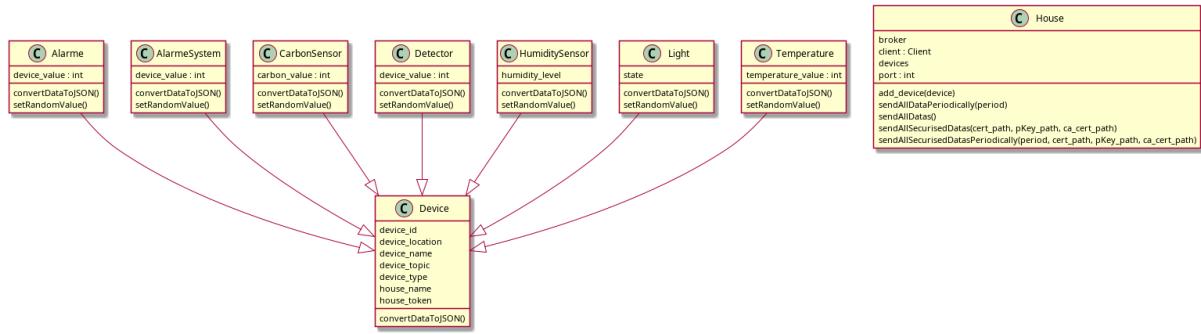


Figure 3.7 – Diagramme UML du simulateur de maisons connectées

- L'automatisation de la **gestion des certificats SSL/TLS** afin de garantir l'authentification des maisons connectées, et de la fiabilité des données envoyées par ces dernières.
- L'exposition d'une interface permettant aux applications clientes (bureau, mobile, web...) d'interagir avec le système.

3.2.1 Architecture logicielle de l'API et choix technologiques

Afin de mettre en place l'**API Rest**, que nous avons nommé **SmartHome API**, nous avons choisis d'utiliser le framework **Laravel** et d'ainsi développer l'application en PHP.

Choix du framework Laravel

Laravel est un frameworl PHP moderne et puissant, basé sur une architecture **MVC**, permettant de structurer et d'organiser de manière efficace le développement d'applications web.

Dans le cadre de notre projet, l'utilisation de ce framework présente de nombreux avantages, notamment en matière de **sécurité** et de **gestion des bases de données**.

Les fonctionnalités clés de Laravel qui nous ont convaincus à l'utiliser afin de développer l'API sont:

- **La sécurité avancée:** Laravel propose nativement une protection contre les attaques courantes tels que les **injection SQL**, les **failles XSS** et les attaques **CSRF**, qui consistent à falsifier les requêtes intersites.
- **ORM Eloquent:** Un système de gestion de base de donnée en optimisé permettant de manipuler des données en utilisant des notions simple de Partie orientée objets, plutôt qu'avec des requêtes SQL classiques. Il permet d'effectuer des opérations **CRUD** sur la base de données.
- **Migrations de bases de données:** Laravel permet de gérer la structure des base de données par le biais d'un système de versionnement. L'objectif est de faciliter la création, la mise à jour ainsi que la synchronisation des bases de données entre les environnements de développement et de production.
- **Gestion des routes et des contrôleurs:** Laravel offre un système de routage facile à comprendre, permettant d'associer de manière simple les requêtes aux méthodes des contrôleurs

En plus des fonctionnalités citées, **Laravel** dispose d'une communauté importante ainsi que d'une documentation très complète, nous ayant permit d'effectuer ce projet pour le mieux.

Déploiement de Laravel au sein du serveur Virtuel

Afin de pouvoir développer l'API, nous avons dans un premier temps déployé le Framework **Laravel**, au sein de notre serveur virtuel.

Ce processus de déploiement s'est déroulé en plusieurs étapes.

Installation des dépendances requises Avant d'installer **Laravel**, nous avons dans une première partie fait en sorte de rendre l'environnement de notre serveur, compatibles avec ses exigences.

Nous avons tout d'abord installer **PHP7.4**, qui est le language supporté par Laravel. Nous avons choisis cette version du fait qu'il est largement compatible avec **Laravel**, et bénéfice encore de mises à jour de sécurité.

Ensuite, nous avons installé **pdo**, une interface qui permet d'interagor avec les bases de données SQL tels que **MySQL**, qui est par ailleurs la base de donnée que nous avons utilisés afin de stocker les informations relatifs à l'identification des maisons au sein de l'**API** et d'**InfluxDB**.

Nous avons installé **Apache2**, un serveur web qui nous permettra d'exécuter Laravel, en raison de sa compatibilité native avec PHP, et de sa facilité de configuration.

Enfin, afin de faciliter l'installation ainsi que les mises à jour des packages **Laravel**, nous avons installer **Composer**, un gestionnaire de dépendance PHP, couramment utilisé en complément de **Laravel**.

Après avoir installés ces dépendances, nous avons installer **Laravel** et mit au point le projet constituant l'**API**, que nous avons appelé **SmartHomeAPI**.

Configuration de SmartHomeAPI La configuration de l'**API** représente une étape fondamental à son fonctionnement. En effet, elle permet de définir quels services nous allons utiliser, quels ports nous allons choisir...

Afin de faire cela, nous avons configuré le fichier **.env**, un fichier caché de **Laravel** permettant la configuration ainsi que la dzfinition des variables d'environnements spécifiques à l'application. Il permet de spécifier les paramètres d'accès à la base de données, le nom de l'application...

Voici un des extraits de notre fichier de configuration **.env**, contenant les informations permettant de lier chaque services essentiels au projet à son port d'utilisation au sein du serveur:

```
1      # On spécifie les informations principales directement liée à l'API ainsi qu'à
2      # son accessibilité
3      APP_NAME=SmartHomeAPI
4      APP_ENV=local
5      APP_KEY=base64:mrAfKToAj/ycaELKT04bx4Zl9Lq/5de3EQ2xTK5LWak=
6      APP_DEBUG=true
7      APP_TIMEZONE=UTC
8      APP_URL=http://172.20.10.2:8000
9      ...
10     # On spécifie l'utilisation de mySql et des informations relatives au sein du
11     # serveur
12     DB_CONNECTION=mysql
13     DB_HOST=127.0.0.1
14     DB_PORT=3306
15     DB_DATABASE=intelighouseDB
16     DB_USERNAME=elies
17     DB_PASSWORD=admin
```

Configuration d'Apache2 **Apache2** est un serveur web open-source, que nous avons utiliser afin d'héberger notre API.

Il permet de traiter des requêtes HTTP et de servir les différentes pages aux clients connectés.

Ainsi, **Apache 2** va assurer la communication entre l'utilisateur souhaitant avoir accès aux données de

sa maison et le serveur virtuelle.

La configuration d'Apache2 est essentielle à la mise en place de notre projet **Laravel**, afin de permettre un accès sécurisé et contrôlé à l'API.

Dans un premier temps, nous avons mis en place un **VirtualHost**, une directive de configuration d'Apache2 qui permet d'héberger plusieurs sites web ou applications au sein d'un même serveur, en définissant des règles spécifiques à chaque hôte. Dans notre cas, nous avons créé un fichier de configuration dédié nommé **intelihouse.conf** afin d'exposer notre API SmartHome sur un port spécifique que nous avons attribué: **Le port 8000**.

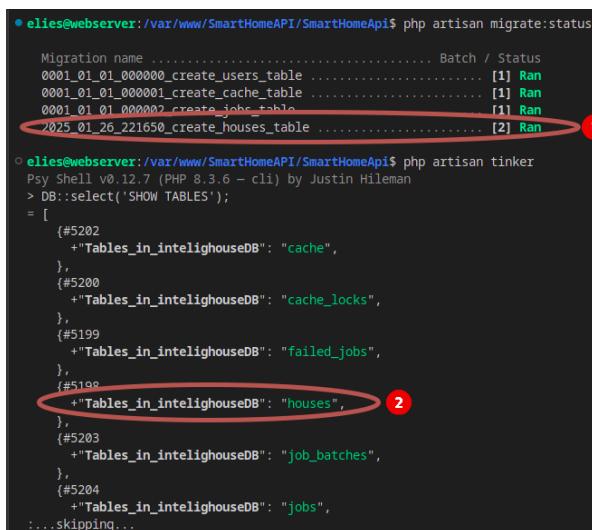
Le fichier de configuration **intelihouse.conf** a été créé au sein du répertoire **/etc/apache2/sites-available/** avec le contenu suivant :

```
1 <VirtualHost *:8000>
2   ServerName 172.20.10.5
3   DocumentRoot /var/www/SmartHomeAPI/SmartHomeApi/public
4
5   <Directory /var/www/SmartHomeAPI/SmartHomeApi/public>
6     AllowOverride All
7     Require all granted
8   </Directory>
9 </VirtualHost>
10
```

La configuration ci-dessus, spécifie ainsi que l'application **Laravel** écoutera sur le **port 8000**, et que le **VirtualHost** sera associée à l'adresse IP **172.20.10.5**. Ainsi, l'API sera accessible via l'url: **http://172.20.10.5:8000/**, depuis le client.

Enfin, on associe le **VirtualHost** au répertoire dans lequel se situe l'application Laravel, et attribuons à apache2 les droits nécessaires afin d'activer les fichiers **.htaccess** et d'y définir des règles.

Configuration de la base de données Nous avons aussi à ce moment là mis en place la base de données contenant les **intelihouseDB**, dans laquelle nous avons généré depuis **Laravel**, les migrations permettant la création de la table **houses** qui stockera les maison, identifiée par un nom de **maison**, un **mots de passe** ainsi qu'un **token**.



The screenshot shows a terminal session on a server named 'elies@webserver'. It displays two commands: 'php artisan migrate:status' and 'php artisan tinker'. The 'migrate:status' command lists database migrations with their batch numbers and statuses. The 'tinker' command shows the results of a database query to list tables in the 'intelihouseDB'. Two specific entries are circled with red circles and numbered: '2025_01_26_221650_create_houses_table' (labeled 1) and 'Tables_in_intelihouseDB: "houses"' (labeled 2).

```
• elies@webserver:/var/www/SmartHomeAPI/SmartHomeApi$ php artisan migrate:status
Migration name ..... Batch / Status
0001_01_000000_create_users_table ..... [1] Ran
0001_01_000001_create_cache_table ..... [1] Ran
0001_01_000002_create_jobs_table ..... [1] Ran
2025_01_26_221650_create_houses_table ..... [2] Ran 1

• elies@webserver:/var/www/SmartHomeAPI/SmartHomeApi$ php artisan tinker
Psy Shell v0.12.7 (PHP 8.3.6 - cli) by Justin Hileman
> DB::select('SHOW TABLES');
= [
  {#5202
    +"Tables_in_intelihouseDB": "cache",
  },
  {#5200
    +"Tables_in_intelihouseDB": "cache_locks",
  },
  {#5199
    +"Tables_in_intelihouseDB": "failed_jobs",
  },
  {#5198
    +"Tables_in_intelihouseDB": "houses", 2
  },
  {#5203
    +"Tables_in_intelihouseDB": "job_batches",
  },
  {#5204
    +"Tables_in_intelihouseDB": "jobs",
  },
  ....skipping...
```

Figure 3.8 – Vérification de la création de la **base de données**

On constate ainsi en analysant l'image ci-dessus deux points:

- La commande `php artisan migrate.status` permet de ici de vérifier le statut quand aux migrations déployées au sein du projet, nous pouvons voir que la migration `create_house_table` a bien été créée.
- La commande `DB::select('SHOW TABLES')` permet de vérifier quels tables sont contenues en base, ce qui nous permet ainsi de constater que la table `houses` issues de la migration `create_houses_tables` a bien été créée au sein de `intelighouseDB`.

Architecture de l'API

SmartHomeAPI repose sur une architecture suivant l'architecture MVC définie par le framework **Laravel**, faisant ainsi d'elle une architecture modulaire et extensible.

L'architecture permet de gérer les maisons connectés de leur authentication, ainsi que la réception des données issues des maisons authentifiées.

L'API suit le modèle **RESTful**, permettant ainsi une séparation claire des responsabilités ainsi qu'une compatibilité avec divers applications clientes (*cf. section 3.3 Surveillance des données avec une interface graphique*).

Structure globale de l'API L'API est structurée en plusieurs composants:

- **Les routes:** Elles permettent de définir les **points d'entrée** de l'API, c'est à dire les différents URL par lesquelles les clients vont effectuer leurs requêtes de manière à accéder au service voulu. Ces points d'entrées ont été mis en place au sein du fichier `routes/api.php`.
- **Les contrôleurs:** Ils permettent d'acheminer les commandes des modèles. Au sein de l'**API**, deux contrôleurs sont centraux: Il y a tout d'abord **HouseController**, qui permet de gérer la création de maison, l'authentification, et il y a **InfluxDBController**, qui permet de gérer la récupération des données des divers maisons, authentifiées par token, via l'API **InfluxDB** interne au serveur.
- **Le Modèle:** Le modèle dans notre cas contient la représentation spécifiques des données spécifiques d'une maison stockée au sein de la base de données **MySQL**.
- **Base de données** L'API accède à deux bases de données au sein du serveur virtuel: La base de donnée liée au séries de données de chaque maison, avec l'accès se fait par le biais de l'API **InfluxDB**, ainsi que la base de données MySQL contenant les utilisateurs. Les deux bases de données correspondent leurs informations par le biais d'une clé appelée **token**, généré lorsque l'utilisateur effectue une requête de création de maison à **SmartHomeAPI**.

Structure des bases de données La figure ci-dessous décrit la structure de la table **House** associée à sa table **intelighouse**, situées au sein de bases de données de type différents.

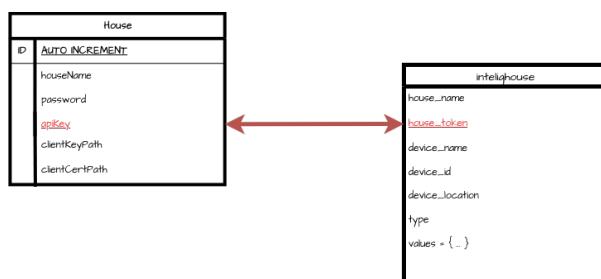


Figure 3.9 – Association d'une Table Maison de Smarthouse à ses séries de données temporelles collectées

La table **House**, situé au sein de la base de donnée **MySQL**, structurée à partie du modèle **House.php**, contient les informations liées à l'authentification d'une **maison connectée** au sein de l'architecture globale du serveur:

- **item** est identifiant unique s'auto-incrémentant au sein de la base à chaque création de maison de la part d'utilisateur.
- **houseName** est nom unique de maisons au sein de la base de données relationnel.
- **password** est le mots de passe haché en **bcrypt** du propriétaire de la maison, lors de son authentification.
- **apiKey** est une clé API unique généré par l'API, elle permet d'identifier les maison au sein des deux bases de données.
- **clientKeyPath** et **clientCertPath** constituent les chemins des certificats client au sein du serveur.

Les valeurs de la tables **intelighouse** ont été décrites à la fin de la partie **Structuration et formalisation des données échangées**, la seul spécificité est le fait que **house_token** constitue une forme de **clé étrangère** liée à la valeur de **apiKey** situé au sein de la base de données **MySQL**.

Fonctionnement des routes La spécification des routes est contenue au sein du fichier **routes/api.php**. Ce fichier contient un **groupe de routes** préfixé **house**, permettant la mise en œuvre des action à réaliser par l'api en fonction des requêtes émises par l'utilisateur.

Au sein de notre API, cinq routes ont étées mise en œuvre permettant le fonctionnement voulu de l'**API** conforme aux besoins nécessaire à l'architecture.

Route	Méthode	Fonction associée	Description
/api/house/create	POST	HouseController@create()	Création d'une maison avec génération des certificats SSL et d'une clé API en retour JSON
/api/house/data	GET	InfluxDBController@ getAllData()	Récupération de toutes les données associées à une maison
/api/house/latestData	GET	InfluxDBController@ getLatestData()	Récupération des dernières données enregistrées
/api/house/dataBySensor	GET	InfluxDBController@ getDataBySensor()	Récupération des données pour un type de capteur spécifique
/api/house/dataAverage	GET	InfluxDBController@ getAveragebySensor()	Moyenne des données d'un capteur sur une période donnée

Table 3.1 – Routes de l'API et leurs descriptions

Résumé de l'architecture L'API suit une architecture MVC et permet l'authentification sécurisée des maisons ainsi que la récupération de leurs données en temps réel. Lorsqu'un propriétaire enregistre sa maison, l'API génère un token unique et des certificats SSL, stockant les informations dans une base MySQL. Pour accéder aux données IoT, le client envoie des requêtes à l'**InfluxDBController**, qui interroge InfluxDB via une API dédiée et retourne les mesures correspondant au token. Cette architecture garantit une gestion efficace des maisons connectées et de leur données, ainsi qu'une automatisation du placement des certificats, et clés privé au sein du dossier **cert** du simulateur.

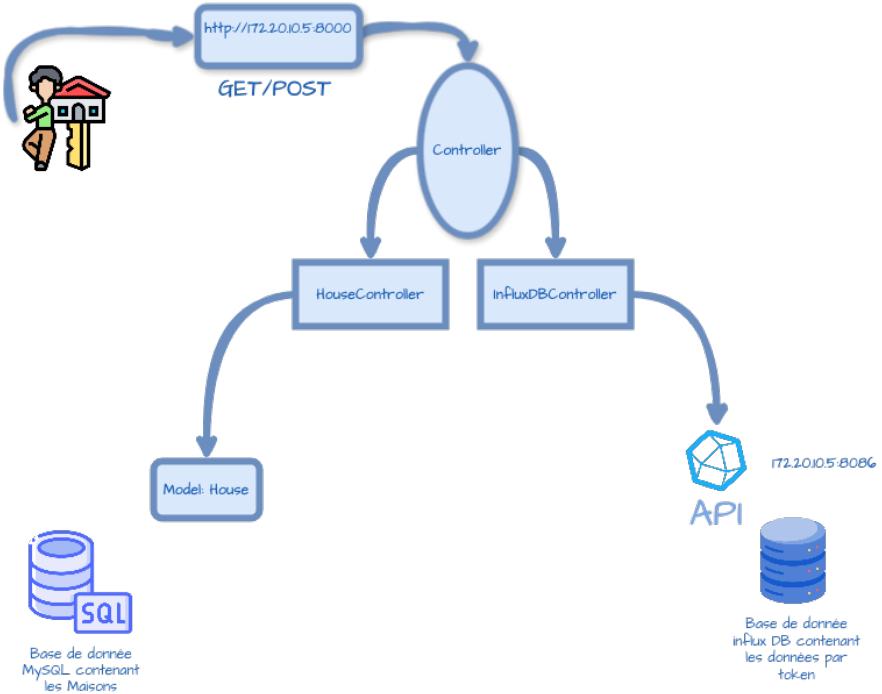


Figure 3.10 – Architecture de SmartHomeAPI

3.2.2 Tests de l'API avec Postman

Dans cette sous-section, nous allons tester l'**API REST** à l'aide de **Postman**, un outil permettant d'envoyer **des requêtes HTTP** et d'**analyser les réponses du serveur**. **Postman** facilite le développement, le test et la validation des API en simulant des appels client sans nécessiter d'interface graphique spécifique. Il nous a été particulièrement utile lors du développement de l'API.

Le tests de l'API est constitué de 3 phases :

- **Créer un propriétaire** et associer son token à la maison connectée simulée, et **récupérer les certificats** nécessaires à l'authentification auprès du broker MQTT.
- **Simuler** l'envoi de données depuis la maison connectée.
- **Vérifier** que l'API accède correctement aux données stockées dans InfluxDB.

Création d'un Propriétaire La première requête consiste à associer un propriétaire à une maison connectée. Pour ce faire, il enregistre sa maison au sein de la base de données du serveur via l'API REST. En réponse, l'**API** retourne plusieurs éléments **essentiels à l'identification et à la sécurisation des échanges** :

- **Le Token:** Un identifiant unique attribué à la maison. Il agit comme une clé d'authentification, connue uniquement du propriétaire et stockée dans la maison connectée. Ce token est utilisé pour accéder aux données des capteurs et garantir que seul le propriétaire peut consulter l'état de son domicile.
- **Le certificat d'autorité (CA.crt):** Nécessaire pour toute communication sécurisée avec le broker MQTT, ce certificat permet de **vérifier l'authenticité du serveur et d'établir une connexion sécurisée**.
- **clientKey:** Il s'agit de la clé privée propre à la maison connectée, générée lors de son enregistrement. Elle est essentielle pour établir une connexion sécurisée et authentifiée avec le broker MQTT, garantissant que **seule cette maison spécifique est autorisée à publier des messages**.

- **clientCrt**: Ce certificat est associé à la clé privée et permet d'authentifier la maison auprès du broker MQTT. Il assure que seules les maisons disposant d'un certificat valide pourront être reconnues et communiquer avec le serveur, empêchant ainsi toute tentative d'usurpation d'identité.

```
http://172.20.10.5:8000/api/house/create

POST http://172.20.10.5:8000/api/house/create

Params Authorization Headers [10] Body Pre-request Script Tests Settings
radio none form-data x-www-form-urlencoded raw binary JSON

1 houseName: "houseName",
2         "password": "User1234",
3         "password_confirmation": "User1234!"
```

Figure 3.11 – Requête d'enregistrement d'un propriétaire associable à sa maison

Figure 3.12 – Réponse de **SmartHomeAPI** après enregistrement réussi

Association d'un propriétaire à sa maison et envoie des données Une fois le propriétaire enregistré via l'**API REST**, la maison connectée récupère les informations d'authentification nécessaires pour communiquer de manière sécurisée avec le broker MQTT. Les certificats ainsi que les clés sont stockés au sein du répertoire `certs/` de **HomeSimulation**, garantissant que chaque maison dispose de son **ensemble de certificats** et de sa clé privée.

```
# IP ou nom de domaine du Broker
# PORT port par défaut 1883 (pour le tests (attention pas sécurisé))
BROKER = '172.20.10.5'
# PORT    = 1883
PORT = 8883

# Authentification de la maison, et association de celle-ci avec
# Les informations du Propriétaire récupéré depuis SmartHomeAPI.
ca_cert      = './certs/CA/ca.crt'
client_cert  = './certs/clients/client.crt'
client_key   = './certs/clients/client.key'
house_token  = "1CMaPGPNA0ZH8wwkWHcHrdQtHpv1fipAnQoIMLYL"
house_name   = "housename"
```

Figure 3.13 – Champs à renseigner au sein de **HomeSimulation** : main.py

Ainsi, les données sont envoyées à **Mosquitto**, et stockés dans **InfluxDB**.

```

elies@elies-HP-Laptop-15s-eq2xxx:~/Bureau/PROJET ZZ3/SmartHome/HomeSimulation/src/iots$ python3 main.py
Publishing to houses/housename/TYPE_LIGHT/lamp_kit: {"house_name": "housename",
"house_token": "1CMaPGPNA0ZH8wwkWWhcHrdQtHpv1fipAnQoIMLYL", "device_name": "lamp_kit", "device_id": 1, "device_location": "kitchen", "type": "light_sensor", "values": {"state": 0}}
Publishing to houses/housename/TYPE_HUMIDITY/humid_sens: {"house_name": "housename",
"house_token": "1CMaPGPNA0ZH8wwkWWhcHrdQtHpv1fipAnQoIMLYL", "device_name": "humid_sens", "device_id": 1, "device_location": "kitchen", "type": "humidity_sensor", "values": {"humidity_level": 0.12}}
Publishing to houses/housename/TYPE_LIGHT/lamp_kit: {"house_name": "housename",
"house_token": "1CMaPGPNA0ZH8wwkWWhcHrdQtHpv1fipAnQoIMLYL", "device_name": "lamp_kit", "device_id": 1, "device_location": "kitchen", "type": "light_sensor", "values": {"state": 0}}
Publishing to houses/housename/TYPE_HUMIDITY/humid_sens: {"house_name": "housename",
"house_token": "1CMaPGPNA0ZH8wwkWWhcHrdQtHpv1fipAnQoIMLYL", "device_name": "humid_sens", "device_id": 1, "device_location": "kitchen", "type": "humidity_sensor", "values": {"humidity_level": 0.12}}
Publishing to houses/housename/TYPE_LIGHT/lamp_kit: {"house_name": "housename",
"house_token": "1CMaPGPNA0ZH8wwkWWhcHrdQtHpv1fipAnQoIMLYL", "device_name": "lamp_kit", "device_id": 1, "device_location": "kitchen", "type": "light_sensor", "values": {"state": 0}}
Publishing to houses/housename/TYPE_HUMIDITY/humid_sens: {"house_name": "housename",
"house_token": "1CMaPGPNA0ZH8wwkWWhcHrdQtHpv1fipAnQoIMLYL", "device_name": "humid_sens", "device_id": 1, "device_location": "kitchen", "type": "humidity_sensor", "values": {"humidity_level": 0.12}}

```

Figure 3.14 – Envoie de 2 séries de données au serveur: **172.20.10.5**

Accès aux données des capteurs via SmartHouseAPI à temps réel Une fois les données envoyées et stockées dans InfluxDB, il est nécessaire de vérifier leur accessibilité via SmartHouseAPI. Par le biais du **token unique attribué lors de l'enregistrement de la maison**, seul le propriétaire peut récupérer les données associées à ses capteurs.

L'accès aux données s'effectue en envoyant une requête **GET** à l'API, en fournissant le **token** en paramètre. L'API interroge alors **InfluxDB** pour extraire les séries de données correspondant à la maison demandée.

Récupération de la dernière donnée acquise



Figure 3.15 – Requête de récupération de la dernière donnée acquise

```

"statement_id": 0,
"series": [
{
  "name": "mqtt_consumer",
  "columns": [
    "time",
    "device_id",
    "device_location",
    "device_name",
    "host",
    "house_name",
    "house_token",
    "topic_houses",
    "type",
    "values_humidity_level",
    "values_state"
  ],
  "values": [
    [
      "2025-03-20T12:19:00.891271277Z",
      "1",
      "kitchen",
      "humid_sens",
      "webservice",
      "housename",
      "1CMaPGPNA0ZH8wwkWHeHrdQtHpv1fipAnQo1MLYL",
      "houses/housename/TYPE_HUMIDITY/humid_sens",
      "humidity_sensor",
      0.12,
      null
    ]
  ]
}
]

```

Figure 3.16 – Réponse de l'API de la dernière données acquise

Récupération de données filtrées en fonction du capteur d'humidité

GET http://172.20.10.5:8000/api/house/dataBySensor?token=Wdp4HCPNvC3wHnvSfSwmVknhLuHtJUkoIzG6zPho&sensorType=humidity_sensor

Figure 3.17 – Requête de récupération des données acquises par capteur d'humidité

```

"statement_id": 0,
"series": [
  {
    "name": "mett_consumes",
    "columns": [
      "time",
      "device_id",
      "device_location",
      "device_name",
      "host",
      "house_name",
      "house_token",
      "topic_houses",
      "type",
      "values_humidity_level",
      "values_state"
    ],
    "values": [
      {
        "time": "2025-03-20T12:18:55.8920000538Z",
        "device_id": "1",
        "device_location": "kitchen",
        "device_name": "humidity_sens",
        "host": "webserver",
        "house_name": "1CMAcPGPNA0ZB8mekWhChrdq0tHpv1ifpAnQo1MLYL",
        "house_token": "houses/housename/TYPE_HUMIDITY/humid_sens",
        "type": "humidity_sensor",
        "values_humidity_level": 0.12,
        "values_state": null
      },
      {
        "time": "2025-03-20T12:18:55.892027038Z",
        "device_id": "1",
        "device_location": "kitchen",
        "device_name": "humidity_sens",
        "host": "webserver",
        "house_name": "1CMAcPGPNA0ZB8mekWhChrdq0tHpv1ifpAnQo1MLYL",
        "house_token": "houses/housename/TYPE_HUMIDITY/humid_sens",
        "type": "humidity_sensor",
        "values_humidity_level": 0.12,
        "values_state": null
      },
      {
        "time": "2025-03-20T12:19:00.891271277Z",
        "device_id": "1",
        "device_location": "kitchen",
        "device_name": "humidity_sens",
        "host": "webserver",
        "house_name": "1CMAcPGPNA0ZB8mekWhChrdq0tHpv1ifpAnQo1MLYL",
        "house_token": "houses/housename/TYPE_HUMIDITY/humid_sens",
        "type": "humidity_sensor",
        "values_humidity_level": 0.12,
        "values_state": null
      }
    ]
  }
]

```

Figure 3.18 – Réponse de l’API des données acquises par capteur d’humidité

Synthèse des Tests Les tests effectués confirment le bon fonctionnement de SmartHouseAPI, démontrant que l’architecture mise en place répond aux exigences définies. L’authentification des maisons via les certificats SSL/TLS et les tokens uniques s’est révélée efficace, garantissant un accès sécurisé aux données.

Les requêtes envoyées via Postman ont permis de valider les différentes fonctionnalités de l’API, notamment la création d’un propriétaire, l’association des certificats, et l’accès aux données stockées dans InfluxDB. L’API a correctement filtré et renvoyé les informations des capteurs en fonction des permissions définies, confirmant ainsi son rôle central dans la gestion et la sécurisation des échanges.

Ces résultats démontrent que l’API fonctionne comme prévu, offrant un accès structuré, fiable et sécurisé aux données des maisons connectées.

3.3 Surveillance des données avec une interface graphique

L'intégration d'une interface graphique est un élément clé dans la gestion et la surveillance d'un système de maison connectée. Une interface utilisateur intuitive permet aux propriétaires de visualiser en temps réel l'état de leurs capteurs, d'interpréter rapidement les données collectées et d'interagir avec le système. Sans cette couche visuelle, l'exploitation des informations se limiterait à des requêtes API et des réponses JSON, ce qui serait peu convivial pour un utilisateur non technique.

Pour le développement de cette interface, nous avons choisi **QT**, un framework puissant et énormément utilisé pour la création d'applications multiplateformes en C++. QT offre plusieurs avantages qui en font un outil performant pour notre projet :

- **Simplicité de gestion de la mémoire:** Contrairement à un programme C++ classique où il est nécessaire de gérer la création ainsi que la destruction des objets, QT réduit les risques de fuites de mémoire en assurant une gestion optimisée des ressources en mémoire.
- **Documentation et support communautaire:** QT bénéficie d'une documentation complète et largement accessible, ce qui facilite la prise en main et l'implémentation des divers fonctionnalités fournies par le framework.
- **Compatibilité multiplateforme :** Une application développée avec QT peut être exécutée sur Windows, Linux, macOS et même certaines plateformes embarquées, offrant ainsi une praticité accrue pour le déploiement.
- **Intégration avec l'API REST :** QT permet une gestion efficace des requêtes réseau, ce qui permet de récupérer les données en temps réel depuis SmartHouseAPI, de les traiter et de les afficher sous forme de graphiques interactifs pour une meilleure surveillance des données issus des capteurs.

Grâce à **QT**, nous avons pu développer un début d'interface permettant de récupérer et parser les données issues de l'API. Bien que l'affichage graphique n'ait pas encore été finalisé, cette première implémentation pose les bases d'un système de monitoring, ouvrant la voie à des améliorations futures telles que l'ajout de tableaux de bord dynamiques et d'un contrôle interactif des équipements connectés.

3.3.1 Architecture logicielle de l'application SmartHouse Monitoring

Structure de SmartHouse Monitoring Tout d'abord, il est essentiel de comprendre la structure d'un projet Qt en C++. Un projet Qt repose sur un fichier .pro, qui définit sa configuration, les bibliothèques utilisées ainsi que les paramètres de compilation. Il est composé de fichiers source et en-têtes, qui contiennent respectivement l'implémentation ainsi que la définition des classes. L'interface graphique peut être conçue à l'aide de fichiers .ui, générés via Qt Designer, mais dans notre cas, nous avons préféré structurer l'interface directement depuis le code, en créant des classes dédiées afin d'avoir un contrôle plus précis sur l'affichage. Enfin, un projet peut inclure des fichiers de ressources (.qrc), permettant d'intégrer des éléments tels que des icônes et images nécessaires à l'affichage.

Ainsi, le développement de **SmartHouse Monitoring** s'est structuré autour de cette architecture en commençant par la configuration du projet QT via le fichier .pro. Cette étape a permis d'intégrer les bibliothèques essentielles au bon fonctionnement de l'application. Tout d'abord, l'ajout des bibli-

thèques réseau (**QtNetwork**) a été nécessaire afin de permettre l'envoi et la réception de requêtes HTTP, assurant ainsi la communication avec **SmartHouseAPI** pour récupérer les données des capteurs en temps réel. Ensuite, l'utilisation des **widgets** (**QtWidgets**) a facilité la conception de l'interface utilisateur, en offrant des composants interactifs tels que **fenêtres**, **boutons** et **champs de saisie**, rendant l'application **ergonomique** et **intuitive**. Enfin, l'intégration des bibliothèques **graphiques** (**QtCharts**) dans le but d'afficher des **courbes** et des **graphiques dynamiques**, afin d'offrir une visualisation **claire** et **efficace** des données capturées par les capteurs IoT.

Explication des différentes classes de SmartHouse Monitoring Afin de faciliter la compréhension, le schéma ci-dessous décrit l'architecture de **SmartHouse Monitoring**:

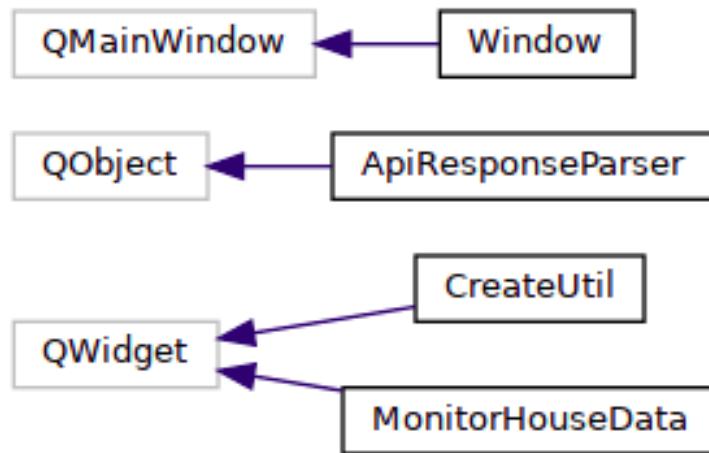


Figure 3.19 – Architecture des classes de **SmartHouse Monitoring**

L'architecture de **SmartHouse Monitoring** repose sur une conception **orientée objet** permettant de gérer les interactions entre l'**interface graphique** et l'**API REST**. Le fichier **main.cpp** constitue le **point d'entrée** de l'application. Il initialise une instance de la classe **Window**, qui est une extension de **QMainWindow**, et la **rend visible à l'exécution**. Ce choix permet d'avoir une interface principale centralisée, capable de gérer et de contenir différents **widgets** en fonction des besoins de l'utilisateur. Ces widgets constituent **des onglets** permettant à l'utilisateur d'aller vers les différents services du logiciel.

L'un des composants clés du projet est la classe **MonitorHouseData**, qui hérite de **QWidget** et sert **d'interface dédiée à la récupération et l'affichage des données issues de l'API**. Cette classe définit plusieurs éléments, notamment **un champ de texte** permettant à l'utilisateur de saisir son **token d'authentification**, des **cases à cocher** afin de **filtrer les données** (données récentes ou historiques), ainsi qu'un **bouton d'envoi** qui déclenche **une requête HTTP**. La communication avec l'**API** est réalisée via **QNetworkAccessManager**, qui permet **d'envoyer des requêtes et de gérer leurs réponses de manière asynchrone**. Une fois les données récupérées, elles sont affichées et interprétées **via un graphique interactif**, rendu possible grâce à l'utilisation de **QtCharts**.

La classe **CreateUtil**, également héritée de **QWidget**, a pour rôle **de gérer l'inscription d'un propriétaire et la création d'une maison connectée**. Elle intègre des **champs de saisie** pour le **nom de la maison** et le **mot de passe**, ainsi qu'un **bouton pour envoyer les informations à l'API REST**. Cette approche permet d'associer chaque **maison** à un **propriétaire unique** et d'obtenir en retour un **token** et des **certificats d'authentification**.

Enfin, l'interprétation des réponses **JSON** envoyées par l'**API** est assurée par la classe **ApiResponseParser**, qui hérite de **QObject**. Son rôle est de convertir les **chaînes JSON** en **objets exploitables** grâce aux classes **QJsonDocument** et **QJsonObject**. Le choix des bibliothèques **QT** natives, comme **QtNetwork** pour les requêtes **HTTP**, **QJson** pour la **manipulation des objets JSON** et **QtCharts** pour l'**affichage des données**, permettra ainsi de répondre au besoin global de **SmartHouse Monitoring**, à savoir afficher les données en temps réels par le biais de graphes interactifs, en communiquant de manière asynchrone à des périodes définies avec **Smarthouse API**.

3.3.2 Tests réalisée avec SmartHouse Monitoring

Structure interface graphique L'interface graphique de **SmartHouse Monitoring** se compose de deux onglets distincts, permettant de gérer et de surveiller les maisons connectées facilement. Le premier onglet, intitulé **Créer une Maison**, est permis d'enregistrer une nouvelle maison au sein du système. Il comporte plusieurs champs de saisie permettant à l'utilisateur d'entrer un nom de maison, un mot de passe, ainsi qu'une confirmation du mot de passe. Une fois ces informations renseignées, un bouton situé au bas de l'interface permet de valider l'enregistrement et de récupérer les données d'authentification associées: le token ainsi que les certificats. Un champ **Résultat** affiche la réponse **JSON** de la requête.

Le second onglet, **Surveiller une maison**, permet à l'utilisateur de suivre en temps réel les données des capteurs de sa maison connectée. Un champ est présent pour entrer le token d'authentification récupéré lors de l'enregistrement. Ensuite, plusieurs options sont disponibles pour filtrer le type de données à récupérer, notamment les données les plus récentes ou les données historiques. Une fois la requête validée en appuyant sur le bouton **État du capteur**, les informations sont récupérées depuis l'**API** et affichées sous forme d'un graphique dynamique, permettant une visualisation claire et détaillée des valeurs mesurées par les capteurs.

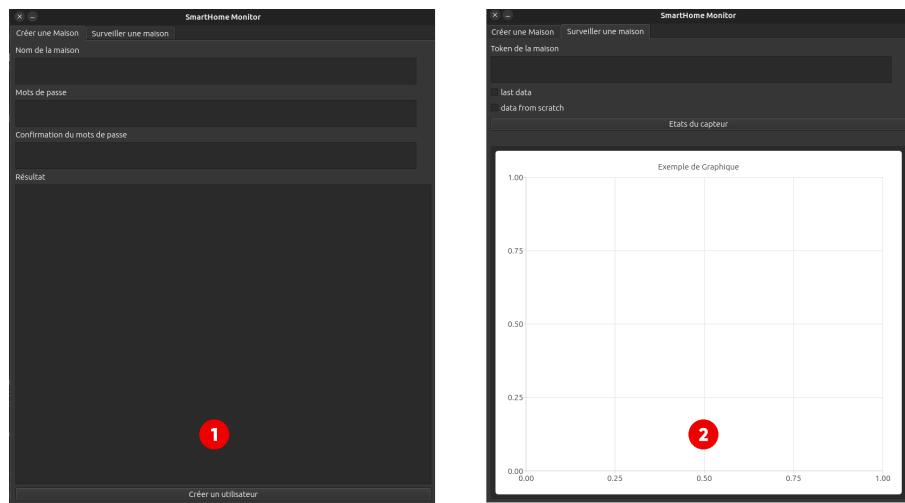


Figure 3.20 – Interface utilisateur réalisée

Une première communication avec l'API L'image ci-dessous illustre un pas majeur dans le développement de **SmartHouse Monitoring**. En effet, nous sommes parvenus à établir une communication entre l'interface Qt et l'**API REST**. Lorsqu'un utilisateur saisit le token de la maison et sélectionne une option de récupération des données, une requête **HTTP** est envoyée à l'**API**. La

réponse obtenue, visible dans la console, est un **JSON** contenant les données des capteurs associés à la maison connectée.

Toutefois, ces données sont **encore brutes et non exploitées visuellement**. L'étape suivante consiste donc à **parser ce JSON** afin d'extraire les informations utiles, telles que les **valeurs des capteurs** et leurs **horodatages**, dans le but de les afficher sous forme de **graphique dynamique** dans l'**onglet de surveillance des capteurs**. Cette dernière phase permettrait une visualisation **claire** de l'**évolution des données en temps réel**, rendant ainsi l'interface **fonctionnelle et efficace** pour le suivi d'une maison connectée.

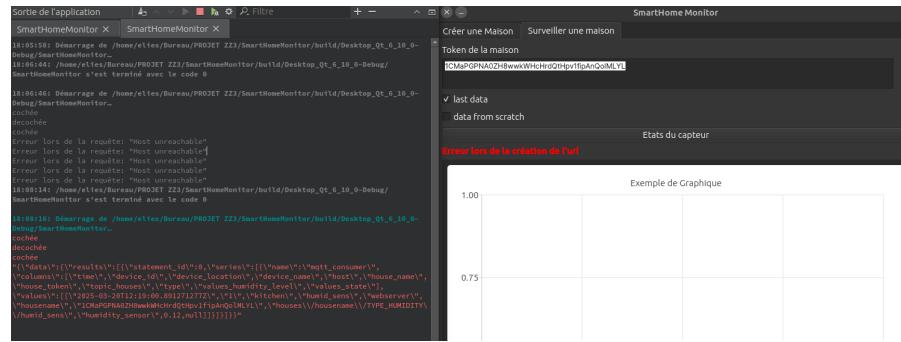


Figure 3.21 – Communication entre **SmartHouse Monitoring** et **SmartHouseAPI**

Chapter 4

Résultats et Discussion

À l'issue de ce projet, nous avons pu mettre en place un système de surveillance des données issues de maisons connectée, sécurisé. L'architecture globale intègre plusieurs technologies, assurant l'**émission** et la **réception** des données en temps réel via le protocole MQTTs, ainsi que leur **stockage** et leur **affichage**.

4.1 Situation à la fin de l'étude

En l'absence de dispositifs physiques, nous avons développé **HomeSimulation**, un **simulateur en Python**, exécuté sur l'une de nos machines, afin de reproduire le comportement de capteurs IoT. Cette simulation permet de **créer une ou plusieurs maisons virtuelles** capables d'émettre, via le protocole MQTTs, des séries de données générées aléatoirement, **simulant ainsi les mesures des capteurs connectés**.

L'architecture de la simulation a été conçue pour être **extensible**, facilitant l'ajout de nouveaux dispositifs et ne limitant ni le **nombre de capteurs** présents dans une maison ni la **fréquence d'envoi** des données.

Cette approche permet de **tester** et **valider** l'ensemble du système dans des conditions **proches d'une utilisation réelle**, permettant ainsi la simulation divers scénarios d'exploitation sans nécessiter de matériel physique.

La transmission des données générées par **HomeSimulation** vers la base de données est assurée via **Mosquitto**, un **broker MQTT** reconnu pour sa **légèreté** et sa **fiabilité**. Afin de garantir la sécurité des échanges, nous avons mis en place une authentification basée sur **SSL/TLS**, assurant ainsi que **seules les maisons autorisées peuvent communiquer avec le serveur**. Ce protocole permet de **chiffrer les données en transit**, les protégeant contre **toute interception ou modification malveillante**. Cette sécurisation est essentielle dans le cadre d'un système de maison connectée, où la **confidentialité** et l'**intégrité** des informations transmises sont essentiels.

SmartHomeAPI, l'**API REST** que nous avons développée en PHP avec le framework **Laravel** constitue le **cœur de notre architecture**. Elle permet d'**authentifier** les propriétaires des maisons connectées, de **générer des certificats SSL** et de récupérer les **données stockées via InfluxDB**, en utilisant un token d'authentification, unique à chaque utilisateur. L'**API** a été conçue de manière à être totalement **indépendante** de l'interface utilisateur, ce qui la rend **utilisable** avec tout type d'interface future – qu'il s'agisse d'une **application web**, d'une **application Android** ou **iOS**. Cette modularité

garantit que l'API n'aura pas besoin d'être modifiée en cas d'évolution des supports ou des interfaces, assurant ainsi une grande **pérennité** et **évolutivité** pour l'ensemble du système.

Pour l'interface utilisateur, nous avons opté pour **Qt**, une technologie très utilisée en entreprise et en industrie, nous permettant ainsi **d'approfondir nos compétences en développement C++**, et de l'utiliser dans un but plus tangible. Ce choix nous a offert une approche plus concrète du développement d'interfaces graphiques et de leur interaction avec une API REST. Grâce à Qt, nous avons pu **récupérer et parser** les données issues de la maison connectée, établissant ainsi une **base solide pour la visualisation des informations en temps réel**. Toutefois, la dernière étape, qui consistait à **afficher ces données sous forme de graphiques dynamiques et interactifs**, n'a pas pu être finalisée par manque de temps. Malgré cela, cette phase nous a permis de mieux appréhender l'intégration de Qt avec notre **API REST**, en posant les **fondations nécessaires pour d'éventuelles améliorations**. À terme, cette interface pourrait être enrichie avec des fonctionnalités d'analyse visuelle avancées, permettant une surveillance optimisée des équipements connectés.

4.2 Analyse des résultats obtenus

Lors de l'analyse des résultats, nous avons mené une **série de tests** visant à **optimiser la structuration des topics MQTT** et à **déterminer le format de données le plus adapté à la simulation des capteurs**.

Dans un premier temps, nous avons réalisé **des tests de transmission de données en clair**, afin d'analyser la **fluidité** et la **fiabilité** des échanges. Par la suite, nous avons sécurisé la communication en **activant MQTTs** et en **intégrant des certificats SSL**, garantissant ainsi l'**authentification** et le **chiffrement des échanges**.

Ces expérimentations nous ont permis d'évaluer la **robustesse** du protocole et d'**analyser l'impact de la sécurisation sur les performances globales du système**, notamment en termes de **latence**.

Les tests effectués ont confirmé plusieurs points forts de notre système :

- **Sécurisation des données** : La transmission est sécurisée grâce à l'utilisation de MQTTs combiné aux **certificats SSL/TLS**, garantissant ainsi un **chiffrement complet** et **protégeant les échanges contre toute tentative d'interception**.
- **Efficacité de l'API REST** : L'**API REST**, remplit efficacement son rôle d'intermédiaire en **filtrant** et en fournissant les données requises par les maisons connectées, assurant ainsi une communication **fluide** entre les différents composants du système.
- **Intégration avec InfluxDB** : La base de données InfluxDB permet une **récupération rapide** et optimisée des **données historiques** ainsi que des **données en temps réel**, ce qui est essentiel pour une **réponse réactive de l'API**.
- **Authentification par certificat** : Le mécanisme d'authentification par certificat, des maisons renforce la sécurité en garantissant que seules les maisons **préalablement enregistrées peuvent communiquer avec le serveur**, limitant ainsi les risques d'intrusion.

Chapter 5

Conclusion

Ce projet nous a permis d'explorer et de mettre en œuvre plusieurs technologies essentielles à la domotique et à l'Internet des objets (IoT). Dès le début, nous avons conçu un système de communication sécurisé permettant l'échange en temps réel des données entre une maison connectée et un serveur central. Cette communication constitue la base indispensable pour assurer un suivi fiable des équipements et optimiser leur gestion.

La sécurisation des échanges a été une priorité majeure, avec l'intégration de MQTTS et de certificats SSL/TLS garantissant l'authenticité des émetteurs et l'intégrité des données transmises. De plus, le choix de InfluxDB pour le stockage des données temporelles a permis d'assurer une récupération rapide et efficace des informations, tout en facilitant l'analyse des historiques.

L'API REST développée sous Laravel joue un rôle clé en assurant une interface sécurisée et modulaire entre les équipements IoT et les systèmes clients. Son architecture indépendante de toute interface utilisateur permet une compatibilité future avec divers supports, qu'il s'agisse d'applications web, Android ou iOS.

Enfin, bien que l'interface utilisateur en Qt ne soit pas totalement finalisée, les bases sont bien établies. Nous avons pu récupérer et parser les données depuis l'API, ouvrant la voie à une intégration future de visualisations graphiques dynamiques. Cette expérience nous a permis d'approfondir nos connaissances en C++, tout en renforçant notre compréhension des interactions entre une interface graphique et une API REST.

Réponse à la problématique *Comment développer un système de monitoring en temps réel pour une maison connectée, garantissant la sécurité de la transmission des données tout en renforçant l'authentification des équipements IoT ?*

Ce projet apporte une **réponse concrète et fonctionnelle à cette problématique**. Grâce à l'utilisation de **MQTTS**, nous avons mis en place un **canal de communication sécurisé garantissant l'authenticité des équipements et la confidentialité des échanges** par le biais de certificats **SSL/TLS**, limitant l'accès aux seules maisons enregistrées.

De plus, l'**API REST** centralisée permet de **filtrer** et récupérer les **données IoT**, tout en offrant une architecture extensible et adaptable. Enfin, l'implémentation de **InfluxDB** permet de gérer efficacement les données en **temps réel**, garantissant une **réactivité optimale du système**.

En conclusion, notre projet répond aux exigences de **sécurité** et de **surveillance en temps réel** pour une maison connectée, tout en offrant une base évolutive permettant d'enrichir et d'améliorer ses

fonctionnalités dans le futur.

5.1 Limites et améliorations possibles

Malgré des bases solides, il est toujours possible **d'envisager des améliorations afin d'approcher encore davantage notre projet de la réalité et de renforcer sa sécurité**. En effet, bien que notre système dispose d'**un premier niveau de sécurité complet**, nous aurions pu aller plus loin dans la réflexion sécuritaire, notamment en développant notre API pour **qu'elle fonctionne en HTTPS plutôt qu'en HTTP**. Cela aurait permis d'assurer une transmission entièrement **chiffrée** des requêtes, garantissant ainsi une protection accrue des données en transit et une meilleure adaptation aux exigences de sécurité du monde réel.

Sur le plan de l'interface, des améliorations auraient également pu être apportées **pour renforcer la connexion utilisateur**. Actuellement, l'authentification repose sur un **mécanisme à facteur unique (1FA)** utilisant uniquement un **token**. La mise en place d'un **second facteur d'authentification**, ou encore **l'implémentation d'un système reposant sur OAuth2**, aurait permis d'offrir une sécurité renforcée pour **les comptes utilisateurs et de réduire le risque d'accès non autorisé**.

Concernant l'interface graphique, nous avons choisi de ne pas utiliser d'outil déjà existant, tel que **Grafana**, afin de nous **perfectionner dans la compréhension et l'utilisation de Qt**, et d'utiliser le language C++ à **des fins plus concrètes**. Cette décision nous a demandé un effort supplémentaire, puisque nous avons dû développer nous-mêmes une interface qui, bien qu'elle ne soit pas finalisée, démontre déjà notre capacité à **récupérer et parser** les données via l'**API REST**. Une alternative aurait été d'implémenter d'abord **Grafana** pour disposer rapidement d'un **outil de visualisation efficace**, puis **d'envisager par la suite soit de l'utiliser tel quel, soit de développer un outil personnalisé plus complet et intuitif pour afficher les données en temps réel**. Cela nous aurait permis d'obtenir dès le départ un aperçu précis des données utilisateur, même si dans notre cas il s'agissait de données simulées.

Enfin, pour se rapprocher encore plus d'un environnement de production, il aurait été intéressant de déployer **notre projet sur un serveur réellement distant**, par exemple en utilisant des services de CLOUDS tels qu'**Azure** ou **AWS**. Ce déploiement aurait permis de mettre en réseau notre application et de la rendre accessible depuis **n'importe quel appareil possédant un compte utilisateur, ouvrant ainsi la voie à une utilisation à plus grande échelle**.

Toutefois, en dépit de ces limites mineures et corrigables, ce projet a été une expérience enrichissante, permettant **d'explorer et d'approfondir** des concepts essentiels en **réseau, sécurité et développement IoT**.

Appendix A

Annexes

A.1 Lexique

A.2 Bibliographie

A.3 Webographie

Bibliography

