

ECOLE CENTRALE MÉDITERRANÉE

—  
UNIVERSITÉ DE MONS

**Architectures logicielles de traitements de données IoT  
médicales, analyse et monitoring**

Thomas Pont - 3A

Rapport Travail de Fin d'Étude

17/04/2023 - 15/09/2023

Tuteur entreprise : Saïd MAHMOUDI  
Tuteur école : François BRUCKER

## Remerciements

Je tiens à remercier toutes les personnes qui ont contribué de près ou de loin au bon déroulé de mon stage.

Ainsi, j'adresse un merci tout particulier à Nicolas BERT qui m'a parlé d'un stage qu'il avait précédemment réalisé au sein du service ILIA, dont les activités m'avaient particulièrement intéressé. Il m'a ainsi mis en relation avec Saïd MAHMOUDI, enseignant chercheur dans ce service, qui a accepté d'être mon tuteur de stage. Je le remercie chaleureusement de m'avoir encadré pendant ces 5 mois. Merci à lui pour ses conseils, ainsi que pour le temps qu'il a accordé pour veiller au bon déroulé du projet.

Merci également à Lilian SOLER, stagiaire et étudiant en informatique en 4ème année de l'école Polytech Grenoble, et à Timothé BRENIER, stagiaire et étudiant en électronique en 4ème année de l'école Polytech Lille, avec qui j'ai collaboré tout au long de mon stage. Travailler avec eux a été très instructif et enrichissant. Nous avons pu partager nos points de vue sur différents sujets, ce qui a nourri ma réflexion sur certains aspects du projet.

Enfin, merci à tous les membres du service ILIA pour leur accueil, leur gentillesse et leur aide tout au long de mon stage.

## Résumé

Dans le cadre de mon travail de fin d'étude pour mon cursus à l'école Centrale Méditerranée, j'ai effectué un stage de 5 mois dans le laboratoire d'informatique de l'Université de Mons en Belgique.

Lors de ce stage, j'ai eu l'opportunité de découvrir le fonctionnement d'un laboratoire de recherche. J'ai travaillé sur la réalisation et le déploiement d'une architecture visant à collecter, traiter et afficher des données médicales. J'ai ainsi participé à la conception d'une API permettant de collecter des données de capteurs médicaux et d'un site permettant de les visualiser facilement.

Durant cette période, j'ai également assisté à des conférences et séminaires donnés par des chercheurs et des doctorants au sein de l'université sur différents sujets autour de l'IoT, l'Intelligence Artificielle et la sécurité des données.

## Abstract

To finish my engineering school, Ecole Centrale Méditerranée, I had to validate an internship, as a part of my formation. Thus, I completed a 5-month internship at the computer science laboratory of the University of Mons in Belgium.

During this internship, I had the opportunity to discover how a research laboratory operates. I worked on the development and deployment of an architecture aimed at collecting, processing, and displaying medical data. I was involved in designing an API for collecting data from medical sensors and a website for easily visualizing them.

Throughout this period, I also attended conferences and seminars conducted by researchers and doctoral students at the university on various topics related to IoT, Artificial Intelligence, and data security.

## Mots clés

Capteur  
Données  
API  
Visualisation  
Sécurité  
Médical

## Glossaire

**API :** Une API (ou Interface de Programmation Applicative) est une interface logicielle permettant de connecter des logiciels et/ou services entre eux pour échanger des données et des fonctionnalités.

**Back-end :** Le back-end ou back désigne la partie serveur d'une application. Cette partie s'occupe du traitement des données et des interactions avec la base de données.

**Base de données :** Une base de données est une collection organisée et structurée de données interconnectées. Elle permet de stocker, gérer et récupérer des informations de manière efficace.

**Blockchain :** La blockchain est une technologie qui permet de stocker des données de manière décentralisée et sécurisée.

**Branche (sur GitHub ou GitLab) :** GitHub et GitLab offrent la possibilité de créer des branches distinctes, chacune représentant une version unique du code source en comparaison avec les autres branches. Ceci permet aux développeurs de travailler sur de nouvelles fonctionnalités tout en laissant le code principal inchangé.

**CI/CD :** CI/CD est l'acronyme de Continuous Integration (Intégration Continue) et Continuous Deployment (Déploiement Continu), deux pratiques permettant l'automatisation des processus de livraison de logiciels.

**Conteneur Docker :** Un conteneur Docker est une unité d'exécution autonome qui contient tout ce qui est nécessaire pour exécuter une application. Il permet entre autres la portabilité (les applications peuvent fonctionner quel que soit l'environnement sur lequel le conteneur est lancé) ou une gestion facilitée des dépendances.

**CSS :** Le CSS ou Feuille de style en cascade est un langage informatique qui permet de définir le style des pages HTML. Il permet de contrôler la mise en forme, la présentation et la disposition des éléments sur une page web.

**Dépôt (sur GitHub ou GitLab) :** Sur GitHub ou GitLab, un dépôt est un espace de stockage pour héberger des projets de développement. Il permet de versionner les différentes modifications apportées au code source, ce qui simplifie le travail en équipe.

**Diagramme d'entités :** Un diagramme d'entités est un diagramme permettant de visualiser les liens d'association entre les différentes entités d'une base de données.

**Électrocardiogramme :** Un électrocardiogramme, également abrégé en ECG, est un enregistrement graphique de l'activité électrique du cœur au cours du temps.

**Federated learning :** Le federated learning ou apprentissage fédéré en français est une approche de l'intelligence artificielle qui consiste à entraîner des modèles de manière distribuée sur différents appareils ou serveurs, tout en maintenant les données locales sur ces appareils sans les centraliser. Cela permet notamment de préserver la confidentialité des données et de réduire la quantité de données transférées vers un serveur central.

**Framework :** Un framework, ou infrastructure logicielle en français, est un ensemble d'outils qui facilite le développement de logiciels en fournissant par exemple des structures et composants préconçus pour résoudre les problèmes courants.

**Front-end :** Le front-end ou front désigne la partie visible et interactive d'une application ou d'un site web.

**GitHub, GitLab :** GitHub ou GitLab sont des plateformes de gestion de code facilitant le versionnement ou la collaboration dans les projets. GitLab possède les mêmes fonctionnalités que GitHub mais permet en plus de pouvoir déployer ses propres instances sur ses propres serveurs.

**HTML :** HTML ou HyperText Markup Language est un langage de balisage utilisé pour structurer le contenu et définir la présentation des pages web.

**Intelligence Artificielle :** L'intelligence artificielle, également désignée par l'abréviation IA, désigne la capacité de machines ou de systèmes informatiques à exécuter des tâches qui nécessitent normalement une intelligence humaine, telles que l'apprentissage, la résolution de problèmes, la compréhension du langage naturel ou la prise de décisions, en utilisant des algorithmes et des modèles mathématiques.

**IoT :** L'IoT (Internet of Thing, ou Internet des objets en français) désigne la connectivité et l'interaction entre les objets physiques et les appareils via Internet.

**Issue (sur GitHub ou GitLab) :** Sur GitHub ou GitLab, une issue (ou ticket en français) est une fonctionnalité permettant de tracer les points ouverts, les tâches à traiter sur le projet en cours. Cela donne de la visibilité sur l'avancement du projet et aide à la planification des tâches.

**JavaScript :** Le JavaScript aussi abrégé en JS est un langage de programmation utilisé notamment dans le développement web pour ajouter de l'interactivité et des fonctionnalités dynamiques au site.

**Merge Request :** Une Merge Request (MR) également appelé Pull Request est une fonctionnalité sur GitHub ou GitLab permettant à un développeur de demander la fusion de modification depuis une branche dans une autre branche.

**Merger :** Dans le contexte du développement, merger (ou fusionner en français) fait référence à l'action de combiner des branches de code distinctes en une seule branche en intégrant les changements d'une branche dans l'autre.

**Méthodes agiles :** Les méthodes agiles sont des méthodes de gestion de projet qui permettent la flexibilité, la collaboration et l'adaptation.

**Méthode MoSCoW :** La méthode MoSCoW est une technique de priorisation utilisée pour la gestion de projet. Les différentes tâches sont alors classées en quatre catégories de manière à faciliter la planification et la prise de décision :

- Must have (doit avoir) : exigence essentielle à la réalisation du projet ;
- Should have (devrait avoir) : exigence importante mais pas cruciale ;
- Could have (pourrait avoir) : exigence souhaitable mais non essentielle ;
- Won't have (ne devrait pas avoir) : exigence qui ne sera pas incluse dans la version actuelle du projet par manque de pertinence ou de temps

**Release :** En informatique, une release désigne une version d'une application, d'un logiciel ou d'un produit.

**Tableau Kanban :** Un tableau Kanban est un outil visuel de gestion de projet qui permet de suivre les tâches ou les éléments d'un projet au fur et à mesure de leur avancement.

**Protocole MQTT :** Le protocole MQTT (Message Queuing Telemetry Transport) est un protocole de messagerie léger et orienté vers le réseau, conçu pour les applications de l'Internet des objets (IoT) et de la communication machine à machine (M2M).

**Requête HTTP :** Une requête HTTP est un type de message envoyé par un client à un serveur à l'aide du protocole HTTP. Ce dernier est utilisé pour transférer des données entre un navigateur web et un serveur web.

**Serveur :** Dans le domaine de l'informatique, un serveur est un ordinateur ou un système informatique qui fournit des services, des données ou des ressources à d'autres appareils, programmes ou utilisateurs.

**TypeScript :** TypeScript, également désigné par l'abréviation TS, est une extension de JavaScript qui ajoute des fonctionnalités tel que le typage.

**UUID :** UUID ou Universally Unique Identifier est un identifiant au format alphanumérique, permettant d'identifier de manière unique un élément.

# Table des matières

<b>Remerciements .....</b>	<b>2</b>
<b>Résumé .....</b>	<b>3</b>
<b>Abstract .....</b>	<b>3</b>
<b>Mots clés.....</b>	<b>4</b>
<b>Glossaire.....</b>	<b>4</b>
<b>Table des matières.....</b>	<b>7</b>
<b>Introduction .....</b>	<b>8</b>
<b>I. Présentation de l'Université de Mons .....</b>	<b>9</b>
1. L'université de Mons .....	9
2. L'institut de recherche Infortech.....	9
3. Le service ILIA (Informatique Logiciel & IA).....	10
<b>II. Le projet RAMi.....</b>	<b>11</b>
1. Présentation générale.....	11
2. L'architecture .....	12
3. Ma mission.....	13
<b>III. Travail réalisé sur RAMi .....</b>	<b>15</b>
1. Version initiale de l'architecture : connexion à une API externe .....	15
2. Seconde version : insertion d'un capteur de température .....	17
3. Déploiement d'une API et d'un site Internet.....	19
<b>IV. Imperfections rencontrées et solutions apportées .....</b>	<b>29</b>
1. Choix des paramètres sur le graphique et informations sur les valeurs .....	29
2. Envoi par batch .....	34
3. Sécurité .....	37
<b>Conclusion et perspectives .....</b>	<b>41</b>
<b>Bibliographie .....</b>	<b>42</b>
<b>Annexe .....</b>	<b>43</b>

## Introduction

Dans le cadre de ma troisième année à l'Ecole Centrale Méditerranée, je devais réaliser un stage de fin d'étude à l'étranger.

Ayant choisi un cursus orienté informatique et ayant un attrait important pour cette discipline, j'ai recherché un stage dans ce domaine.

Par ailleurs, j'avais réalisé mes stages de fin de première année et de deuxième année dans des entreprises. Or, les activités de recherche m'ont toujours tout particulièrement attiré. C'est pourquoi, j'ai naturellement choisi ce stage au sein du service ILIA de l'UMONS, dans la mesure où il permettait de concilier mes différents centres d'intérêt.

Dans ce rapport, je présenterai la structure qui m'a accueilli, le projet auquel j'ai participé, ainsi que les travaux que j'ai réalisés.

# I. Présentation de l'Université de Mons

## 1. L'université de Mons

L'université de Mons [1] (aussi appelée UMons) est une université belge située dans la province de Hainaut (en Wallonie, proche de la frontière franco/belge). Elle possède des campus à Mons et à Charleroi. Elle a été créée en 2009 et est issue de la fusion de l'université de Mons-Hainaut et de la Faculté polytechnique de Mons.



*Figure 1 : Carte de la Belgique avec les différents sites de l'UMons (source : Google Maps)*

L'UMons est l'une des 6 universités francophones de Belgique. Avec près de 10 000 étudiants, c'est la 4ème en termes de taille. Les trois plus importantes (l'Université catholique de Louvain, l'Université libre de Bruxelles et l'Université de Liège) accueillent chacune plus de 25 000 étudiants, tandis que les deux plus petites (l'Université de Namur et l'Université Saint-Louis - Bruxelles) en accueillent au maximum 7 000.

L'UMons propose plus de 150 formations, et ses étudiants sont répartis dans 7 facultés et 3 écoles. Les domaines d'apprentissage sont très divers, allant de l'architecture à la linguistique en passant par la chimie ou encore l'urbanisme.

L'UMons est également très engagée dans le domaine de la recherche. Elle accueille près de 1 000 chercheurs et 450 doctorants dans 10 instituts de recherche, regroupant une centaine de services.

Lors de mon stage, j'étais rattaché à l'institut de recherche Infortech.

## 2. L'institut de recherche Infortech

Infortech [2] est l'institut de recherche en Technologie de l'Information et Sciences de l'Informatique de l'UMons. Il est composé d'environ 120 chercheurs et 35 doctorants, répartis dans 14 services spécialisés autour de la collecte et de transmission de données, du Big Data

et du Cloud computing, de l'Intelligence Artificielle (IA), du machine learning et du deep learning, du génie logiciel et algorithmique, du traitement du signal et enfin des technologies de l'éducation.

Les projets menés par cet institut se veulent interdisciplinaires et collaboratifs avec d'autres organismes pour être menés à l'échelle provinciale, régionale, fédérale, européenne ou internationale.

Lors de mon stage, j'ai travaillé dans le service ILIA (Informatique Logiciel & IA) de cet institut.

### **3. Le service ILIA (Informatique Logiciel & IA)**

Le service ILIA [3] est installé dans les locaux de la faculté polytechnique de Mons, construits en 1545 pour le collège de Houdain et rattachés en 1837 à cette faculté.



*Figure 2 : Site de Houdain, faculté polytechnique de Mons (Source : Wikipédia)*

Ce service d'Infortech est spécialisé dans l'Intelligence Artificielle, la vision par ordinateur, la gestion des données dans le domaine de l'Internet des Objets, le Cloud et l'Edge Computing. Au sein de cette structure, différents projets sont menés, tels que :

- WALLeSmart : plateforme qui collecte des données (météorologiques, état des sols, ...) pour optimiser la rentabilité des agriculteurs de Wallonie;
- RAMi : plateforme qui collecte des données médicales pour faciliter la prise en charge des patients.

C'est sur ce second projet, RAMi, que mon stage a porté.

## **II. Le projet RAMi**

Comme énoncé précédemment, j'ai travaillé lors de mon stage sur le projet RAMi (Real-Time Architecture for the Monitoring of elderly patients thanks to the Internet of Medical Things - architecture en temps réel pour l'acquisition de données des personnes âgées grâce à l'internet des objets médicaux).

Dans cette partie, le contexte et les enjeux du projet RAMi, ainsi que les objectifs du stage seront présentés.

### **1. Présentation générale**

De nos jours, on observe un vieillissement de la population [4] ainsi qu'une forte volonté des personnes âgées de rester indépendantes. En revanche, en raison de leur âge [5], il est essentiel de pouvoir vérifier régulièrement leurs constantes. Afin d'éviter des déplacements et/ou une prise en charge dans un système de santé onéreux, il est nécessaire de pouvoir suivre ces patients depuis leur domicile ainsi que pouvoir détecter au plus tôt leurs potentiels problèmes de santé.

Ainsi, depuis ces dernières années, on observe un fort développement de l'IoMT (Internet of Medical Thing). Il s'agit d'une sous-branche de l'IoT (Internet of Thing) qui se concentre sur le domaine de la santé. Plus précisément, l'IoMT définit l'ensemble des dispositifs médicaux connectés à Internet, tels que les capteurs, les moniteurs de santé, les applications mobiles de santé, les implants, les wearables, etc. L'objectif de l'IoMT est de permettre une surveillance précise et en temps réel de la santé des patients, de faciliter le diagnostic et le traitement des maladies, afin d'améliorer la qualité des soins de santé en général [6].

De tels dispositifs engendrent également des contraintes. En effet, comme ces dispositifs collectent des données sensibles sur la santé des individus, il est essentiel de prendre des mesures pour assurer la sécurité et la confidentialité de ces informations. Par ailleurs, il est nécessaire que le mode de transmission des données soit fiable et que le traitement de celles-ci soit fait en temps réel pour ne pas mettre en péril la vie des patients.

RAMi vise à répondre à toutes ces problématiques. Il s'agit d'une architecture temps réel pour le suivi des patients âgés grâce à l'Internet des Objets Médicaux. Cette architecture permet d'effectuer un suivi des patients âgés. Elle intègre un système d'alerte utilisant l'analyse des données en temps réel à l'aide d'algorithmes de Machine Learning pour alerter en cas de problème chez un patient. Par ailleurs, elle répond à toutes les contraintes pesant sur le domaine en respectant la vie privée et la confidentialité des données, notamment en sécurisant les données grâce à la blockchain.

## 2. L'architecture

Un premier article [7] décrivant l'architecture proposée par RAMi ainsi que ses avantages par rapport aux autres solutions déjà existantes a été publié en septembre 2022.

### a. Architecture générale

L'architecture proposée par RAMi est composée de différentes couches : une couche *Things* où les données médicales sont récupérées à partir des capteurs, et deux couches *Fog* et *Cloud* où les données sont traitées et analysées.

Les données sont récupérées depuis les capteurs dans la couche *Things*. Ces données sont transmises à la couche *Fog* via un protocole spécifique nécessitant peu d'énergie tel que le bluetooth ou LoRaWAN. Lorsqu'elles arrivent dans la couche *Fog*, elles sont traitées puis temporairement stockées avant d'être transmises à la couche *Cloud*.

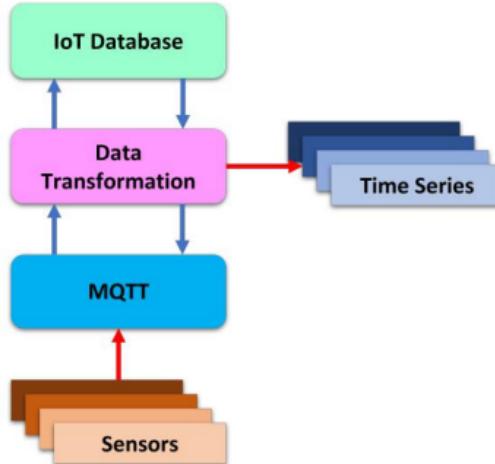


Figure 3 : Schéma du fonctionnement de la partie Fog de RAMi (source : article sur RAMi - annexe 6)

Les données arrivent dans la couche *Cloud* via un broker MQTT. Il s'agit d'un serveur intermédiaire qui facilite la communication entre des dispositifs ou des applications utilisant le protocole MQTT. Dès lors, ces données vont pouvoir être analysées grâce à un algorithme de machine learning en temps réel. Parallèlement à cela, elles sont stockées dans une base de données où des infirmiers ou médecins peuvent les annoter pour indiquer si à l'aide des données, l'IA a bien su détecter si un patient était malade ou non. Pour faciliter la future recherche des données et leur affichage, les données sont converties en objets et stockées. Enfin, une page affiche les données et un système de cache est mis en place de manière à optimiser cet affichage.

Pour assurer la couche *Cloud* de RAMi, différents logiciels ont été retenus : Eclipse Mosquitto [8], PostgreSQL [9], Apache Camel [10], Apache Druid [11], Redis [12], MinIO [13], Elasticsearch [14], Apache Superset [15], Apache Spark [16] et Apache Kafka [17].

L'organisation de cette couche est résumée sur la figure 4 ci-dessous.

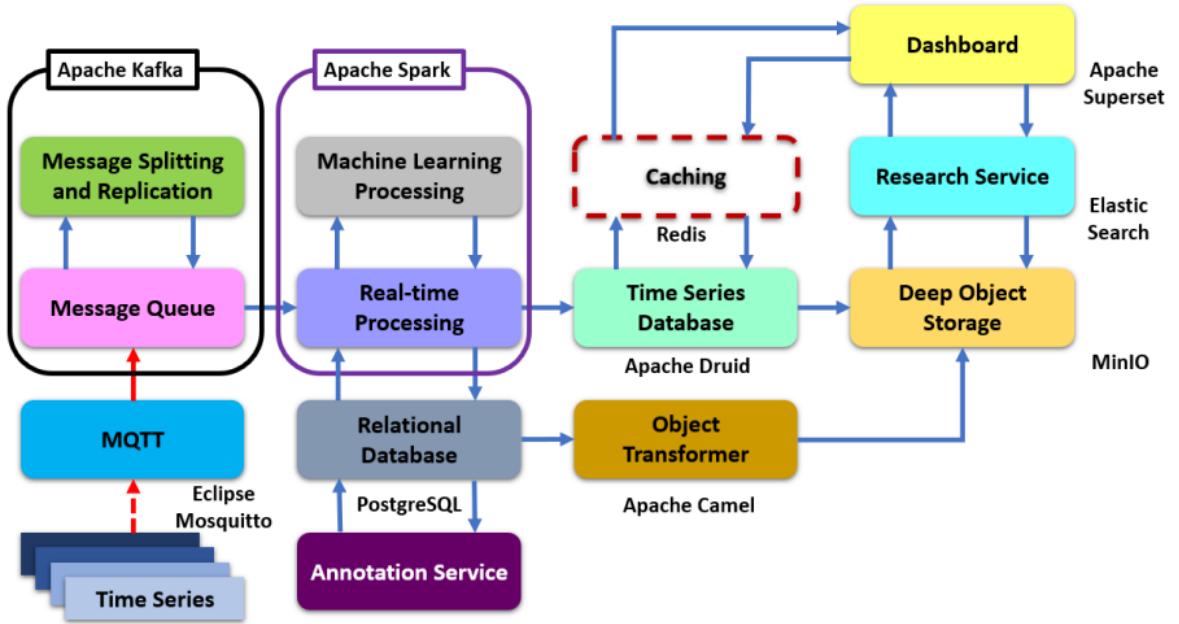


Figure 4 : Schéma des différents logiciels de la partie Cloud de RAMi (source : article sur RAMi - annexe 6)

## b. Avantages

Un des premiers avantages de RAMi est qu'il n'utilise que des logiciels open source. Cela lui permet donc de ne pas être dépendant d'un éditeur de logiciels ou d'un code propriétaire.

De plus, chaque brique logicielle peut être remplacée si un nouveau logiciel plus performant ou plus adapté émerge. Et, chaque logiciel étant indépendant, il peut être remplacé par un autre si l'on souhaite adapter l'architecture de RAMi en fonction des demandes.

De plus, l'architecture RAMi est tolérante aux pannes. Par exemple, les données sont temporairement stockées dans la couche *Fog* afin de prévenir toutes pertes de données en cas de problème de transmission avec la couche *Cloud*.

Par ailleurs, l'entièreté des logiciels est dockerisée de manière à pouvoir déployer RAMi facilement, quel que soit l'environnement.

Enfin, RAMi offre une sécurité sur les données grâce à la blockchain. Celle-ci est déployée de manière à authentifier toutes les données et donc assurer leur valeur et vérifier qui a le droit d'y accéder ou non.

## 3. Ma mission

Au début de mon stage, l'architecture présentée ci-dessous était restée purement théorique et n'avait pas encore été développée. L'objectif de mon stage était de commencer à la mettre en œuvre. Dans la mesure où il y a beaucoup de logiciels et de microservices à

développer, l'idée est de procéder étape par étape, en construisant tout d'abord un modèle qui fonctionne avec les éléments essentiels, puis en ajoutant progressivement des éléments et des fonctionnalités. Dans la suite de ce rapport, je présenterai les différentes versions de l'architecture sur lesquelles j'ai travaillé, ainsi que les réflexions qui ont été menées.

### **III. Travail réalisé sur RAMi**

Dans le cadre de mon stage, j'ai essentiellement travaillé sur les sujets suivants :

- Mise en oeuvre d'une première version fonctionnelle simplifiée de RAMi
- Mise en oeuvre d'une seconde version, intégrant l'utilisation de capteurs
- Développement d'une API et d'un site web pour collecter et afficher des données envoyées par des capteurs

Pour les deux premiers points, j'ai travaillé seul sous la supervision de mon tuteur. Pour le développement de l'API et du site, j'ai travaillé avec deux autres stagiaires : Lilian SOLER pour la partie informatique et Timothé BRENIER pour la partie électronique (envoi des données du capteur à l'API).

#### **1. Version initiale de l'architecture : connexion à une API externe**

Avec la première architecture (cf. figure 5), l'objectif était de collecter les données d'une API ouverte, de les traiter et de les afficher sur un site afin que les utilisateurs puissent facilement les visualiser, sans avoir à se connecter à l'API, mais en les consultant directement depuis le site.

Pour disposer de premières données, j'ai utilisé l'API ouverte OpenWeather [18] qui permet d'obtenir les données météorologiques du monde entier. Il est simple de s'y connecter et elle est gratuite pour l'usage limité que j'en ai. Je me suis limité à la collecte de données d'une dizaine de villes en Allemagne, en Belgique et aux Pays-Bas. Parmi les informations fournies par OpenWeather, j'ai choisi d'afficher le nom de la ville, le pays, une courte description de la météo, la température ainsi que l'heure et le jour de la mesure.

Pour la phase de collecte et de traitement des données, j'ai construit un conteneur Docker qui contient une instance d'Apache Kafka [17] et de ZooKeeper [19].

Apache Kafka est une plateforme de streaming de données open source largement utilisée pour la gestion des flux de données en temps réel. Elle est conçue pour être hautement évolutif, fiable et tolérante aux pannes. Elle permet également aux applications de publier, de souscrire et de traiter des flux continus de données à grande échelle.

ZooKeeper, également développé par Apache, est un service de coordination distribuée utilisé par Kafka pour gérer l'état de la plateforme et la configuration. Il assure la coordination des différents composants d'un cluster Kafka.

En parallèle, j'ai codé un script que j'ai lancé dans un autre conteneur Docker. Celui-ci se connecte à mon instance Kafka précédemment lancée et récupère les données de l'API en utilisant Kafka.

J'ai également déployé un troisième conteneur Docker qui offre une instance de PostgreSQL [9].

PostgreSQL est un système de gestion de base de données relationnelle et objet open source et conforme aux normes SQL. Les données précédemment collectées sont stockées dans une base de données PostgreSQL.

Pour pouvoir visualiser les données, j'ai codé un serveur Express [20] qui fait appel à la base de données et un site utilisant HTML/CSS/JavaScript qui affiche les données en temps réel.

Express ou express.js est un framework JavaScript utilisé pour développer des applications Web côté serveur. J'ai choisi ce framework pour sa simplicité à mettre en place pour une application comme celle-là.

Le site sur lequel les données sont restituées est composé de quatre pages (cf. figure 6) :

- la première affiche toutes les données contenues dans la base de données;
- la deuxième permet à l'utilisateur d'entrer le nom d'une ville. Si celle-ci est contenue dans la base de données, les informations la concernant s'affichent ainsi qu'un graphique montrant l'évolution des températures en fonction du temps;
- la troisième offre à l'utilisateur la possibilité de choisir un pays parmi un menu déroulant (tous les pays contenus dans la base de données sont proposés). Les informations sur le pays choisi sont alors affichées sur la page;
- la dernière affiche la météo la plus récente pour toutes les villes contenues dans la base de données, triée par ordre alphabétique des villes.

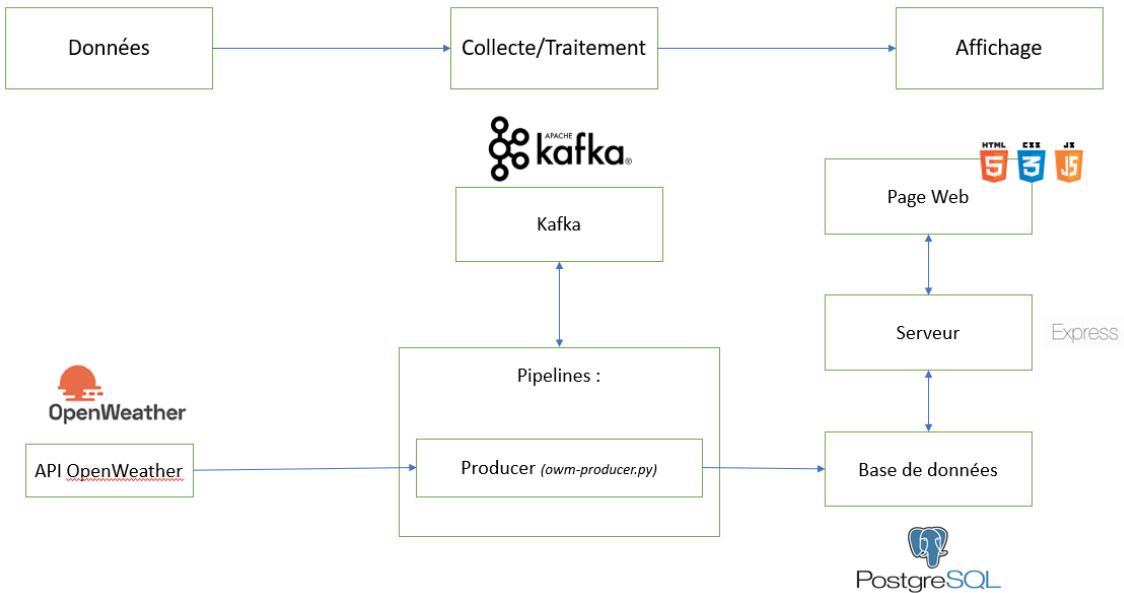


Figure 5 : Schéma de fonctionnement de la première architecture

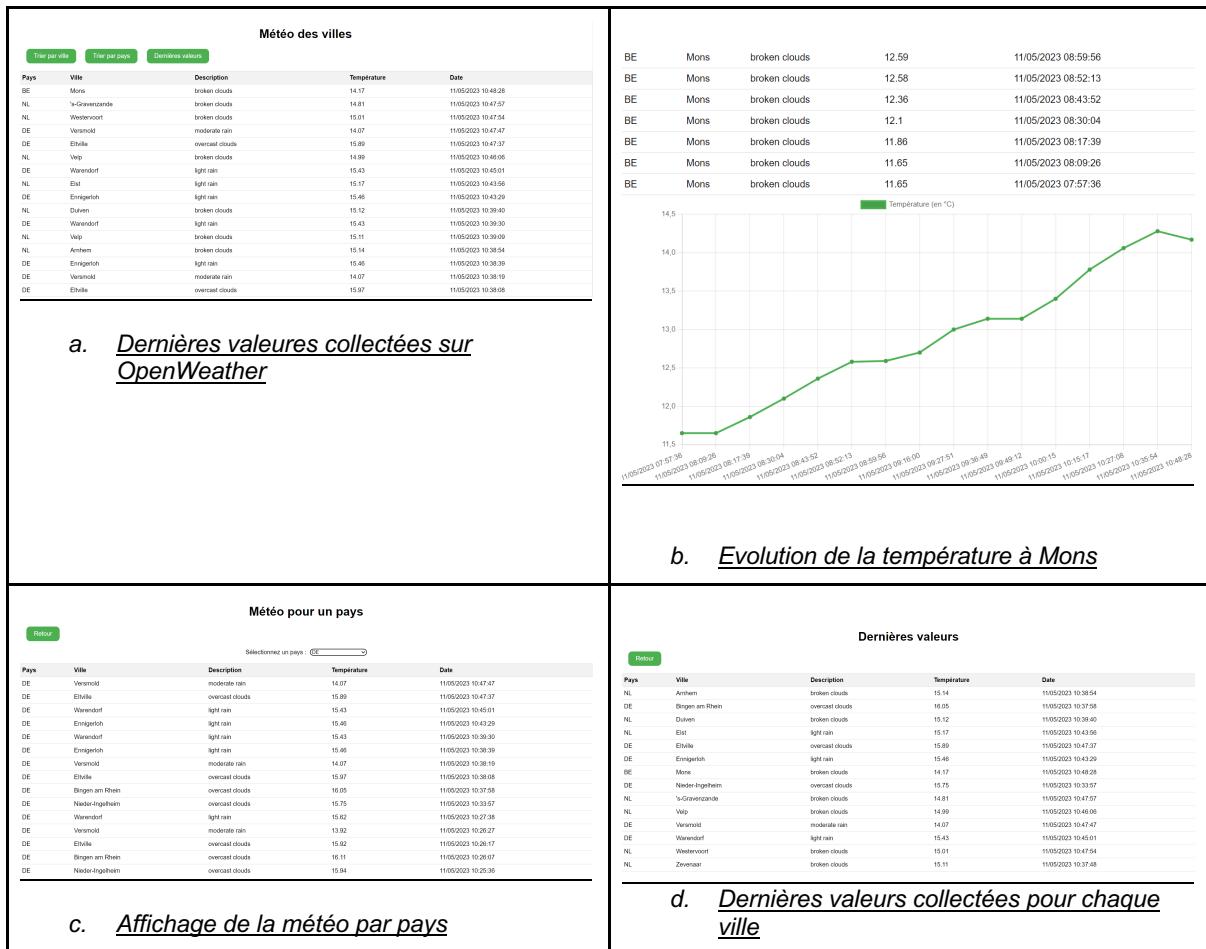


Figure 6: Affichage des pages du site

## 2. Seconde version : insertion d'un capteur de température

Dans un second temps, nous avons décidé, en collaboration avec mon tuteur, d'ajouter un capteur à l'architecture (cf. figure 7). A terme, RAMi collectera des données de la santé. Pour cette version de RAMi, nous avons choisi d'utiliser un DHT22 [21], un capteur de température et d'humidité. En effet, il s'agit d'un matériel accessible et facile d'utilisation. Nous l'avons relié à un ESP32 [22], un microcontrôleur qui présente l'intérêt de pouvoir envoyer les données reçues par le capteur via WiFi.

Ces données sont envoyées toutes les 30 secondes sur un serveur Mosquitto [8] lancé depuis un conteneur Docker. Mosquitto est un broker MQTT open-source qui implémente le protocole MQTT (Message Queuing Telemetry Transport). Les données du capteur sont transmises sur 3 topics différents :

- esp32/group1/temperature
- esp32/group1/humidity
- esp32/group1/time

Ensuite, j'ai codé un script dans un conteneur Docker qui se connecte aux topics mentionnés et ajoute les données ainsi récupérées dans une base de données PostgreSQL, sur un autre conteneur Docker.

Enfin, j'ai programmé un serveur Express ainsi qu'une page en HTML/CSS/JavaScript pour afficher les données. Sur cette page, il est possible de visualiser la dernière mesure de température et d'humidité. Un graphique affiche l'évolution de ces données au cours des 10 dernières minutes (cf. figure 8).

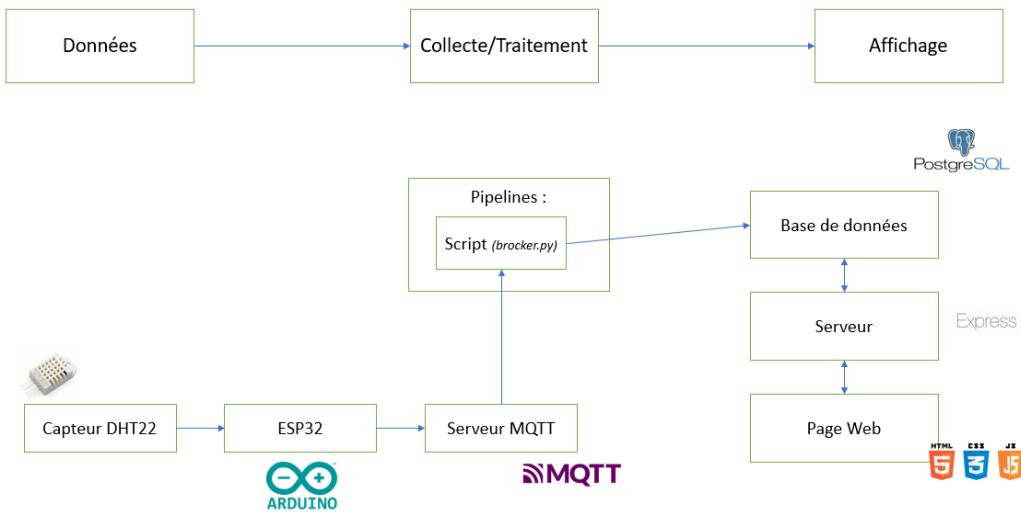


Figure 7 : Schéma du fonctionnement de la deuxième architecture

### Affichage de données en temps réel

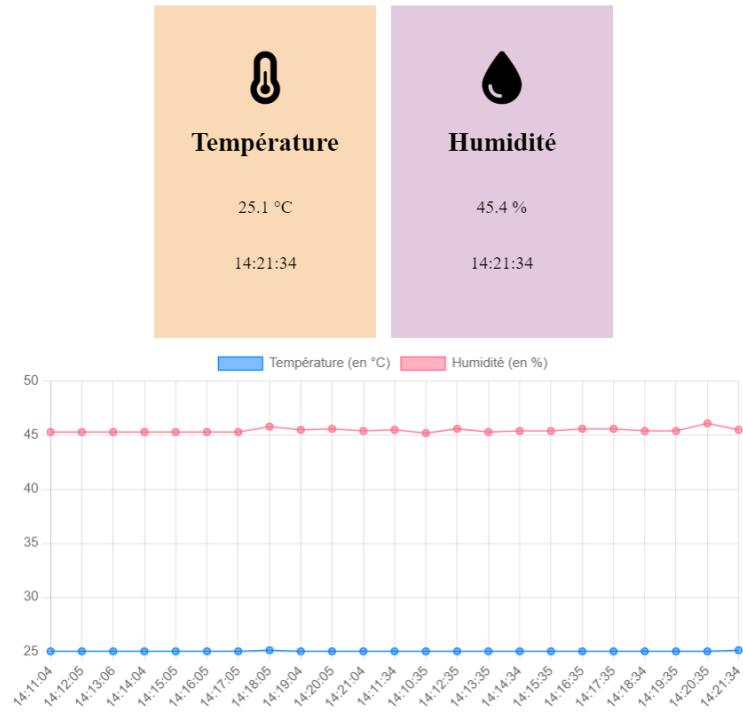


Figure 8 : Site présentant les données collectées par le DHT22

### **3. Déploiement d'une API et d'un site Internet**

Après avoir travaillé sur ces deux premières architectures, nous avons discuté, avec mon tuteur et les autres stagiaires du service, des objectifs finaux de RAMi et des points à développer en priorité pour rendre le système le plus opérationnel possible.

Dans la mesure où nous devions pouvoir collecter des données envoyées par des capteurs pour pouvoir les afficher en temps réel et les conserver afin qu'elles servent de base d'entraînement pour du *Federated Learning*, nous avons décidé de déployer une API et un site Internet.

Les capteurs, reliés à des ESP32, devaient pouvoir envoyer leur mesure à l'API, ces mesures étant ensuite enregistrées dans une base de données.

Par ailleurs, l'API devait être adaptée à tous les types de données et à leurs spécificités. Par exemple, pour pouvoir tracer un électrocardiogramme, il faut pouvoir envoyer des centaines de mesures à la seconde, tandis que la température évolue beaucoup plus lentement et ne nécessite donc pas plus d'un envoi par heure.

Enfin, le site Internet devait pouvoir se connecter à l'API pour récupérer les données et les afficher en temps réel.

Une grande partie de la suite de mon travail a été consacrée au développement et au déploiement de cette API et du site Internet, en collaboration avec un autre stagiaire, Lilian SOLER, étudiant en 4ème année en informatique à Polytech Grenoble.

#### **a. Organisation du travail et répartition des actions en mode collaboratif**

Comme expliqué ci-dessus, j'ai travaillé en collaboration avec un autre étudiant stagiaire dans le service.

Pour organiser au mieux notre travail, nous avons opté pour une stratégie agile. Nous avons créé deux dépôts GitLab pour nous permettre de versionner le code du front et du back de notre site. Dès que nous identifions un point d'amélioration ou une fonctionnalité à ajouter nous ouvrons une *Issue*, détaillant la problématique et ce que nous souhaitons réaliser.

Au fur et à mesure, nous ajoutons des « tags » (cf. figure 9) de manière à préciser pour chaque “Issue” si celle-ci nous paraissait nécessaire ou non selon la méthode MoSCoW, le domaine qu'elle concernait (sécurité, optimisation, ...) ainsi que son état d'avancement (backlog, planned, doing, done, reviewed).

Cette façon de procéder nous permettait de suivre précisément l'avancement des travaux et de classer les tâches dans un tableau Kanban (cf. figure 10).

Choose number of measurements shown in the chart	rami/umons-sensor-frontend#18 · created 1 month ago by Thomas Pont	Sprint 1	Jul 2, 2023	1d	1  0	updated 1 month ago
Category : UI/UX Priority : could Status : review Team : frontend Type : feature						
Adjust date according to my localisation	rami/umons-sensor-frontend#16 · created 1 month ago by Lilian Soler	Sprint 1	Jun 28, 2023	4h	0  0	updated 1 month ago
Category : UI/UX Priority : won't Status : backlog Team : frontend Type : feature						
change layout according to screen size	rami/umons-sensor-frontend#7 · created 1 month ago by Thomas Pont			1d	0  0	updated 1 month ago
Category : UI/UX Priority : should Status : backlog Team : frontend Type : feature						

Figure 9 : Exemple d'Issues avec tags

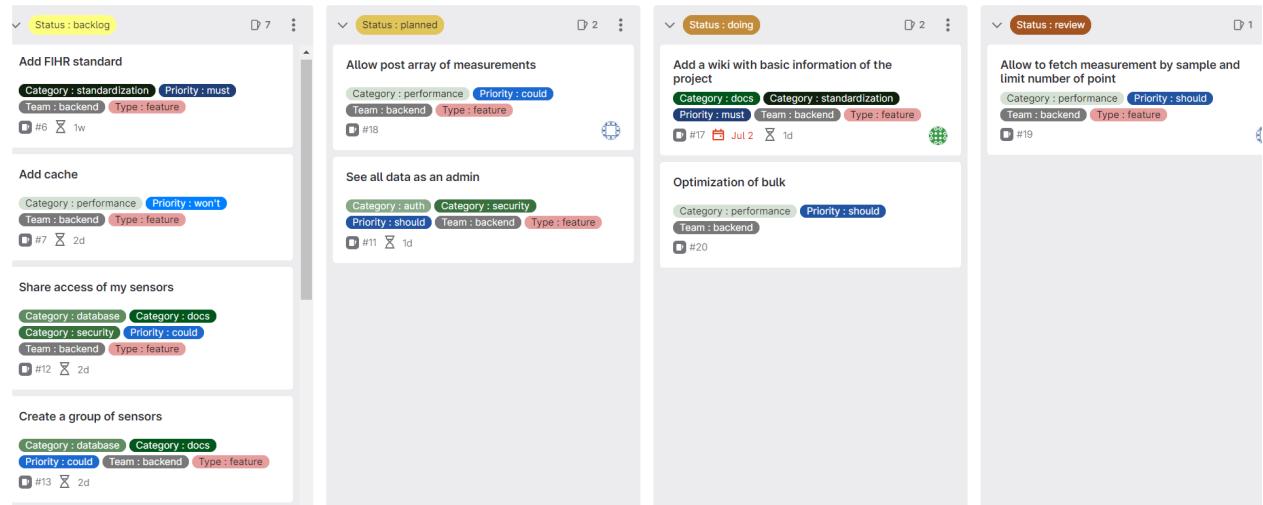


Figure 10 : Tableau Kanban

Chaque début de semaine, nous nous réunissions entre stagiaires pour définir les tâches à réaliser en priorité dans la semaine et nous nous répartissions le travail après avoir estimé au préalable le temps nécessaire à la réalisation de chacune des tâches. En fin de semaine, nous avions une réunion avec notre tuteur, au cours de laquelle nous partagions le travail réalisé et discutions des perspectives.

Par ailleurs, pour pouvoir travailler en parallèle sur RAMi sans avoir de souci de versionning et pour optimiser la qualité de nos développements et de nos tests, nous avons systématiquement respecté les principes suivants :

- Lorsque nous commençons à travailler sur une *Issue*, nous ouvrons une Merge Request (MR) et nous créons une nouvelle branche au nom de */Issue* sur le dépôt GitLab. Puis, nous travaillons sur cette branche pour réaliser */Issue* souhaitée.
- Une fois */Issue* traitée, nous demandons une validation de la Merge Request à l'autre stagiaire, de façon à ce qu'il note les erreurs laissées par inadvertance et/ou fasse des commentaires sur comment optimiser le code.
- Une fois tous les retours corrigés, nous mergeons la branche créée pour l'issue dans la branche *develop*.
- Lorsque plusieurs améliorations étaient portées à la branche *develop* nous faisons une *release* et nous mergeons la branche *develop* dans la branche *main*.
- Lorsqu'un de nous écrit un code, c'est l'autre qui écrit les tests unitaires pour éviter d'être biaisé dans la manière de les concevoir et de les réaliser.

Enfin, nous avons écrit un script sur Gitlab pour que le Front et le Back se déploient directement en cas de modification sur les branches main et develop pour avoir un environnement de déploiement et un environnement de test.

#### b. Description des solutions retenues pour l'API et l'interface utilisateur

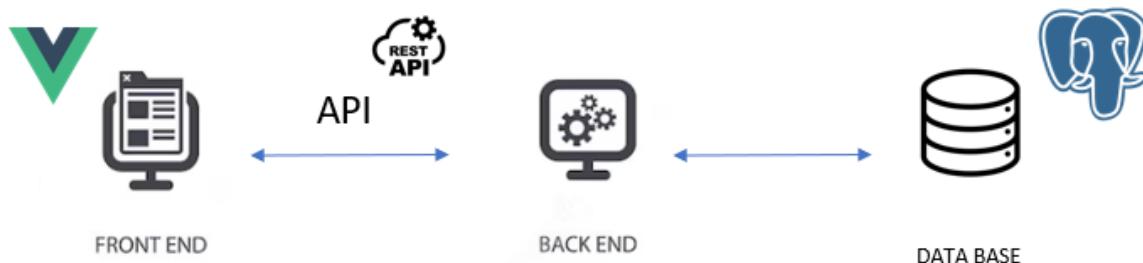
Pour l'API, nous sommes partis sur une API REST avec Node/TypeScript. Une API REST (Representational State Transfer) ou API RESTful est une API qui respecte des règles et des conventions. Elle permet à des systèmes informatiques de communiquer via des protocoles HTTP.

Une des décisions majeures a été de la dockeriser entièrement, de manière à pouvoir la déployer facilement et rapidement dans n'importe quel environnement, en particulier - à terme - dans les hôpitaux et centres de santé.

Pour le stockage des informations envoyées à l'API, nous avons choisi une base de données PostgreSQL. En effet, PostgreSQL est un système de gestion de bases de données relationnelles qui est Open Source et qui a l'avantage d'être plus fiable et plus robuste que les autres systèmes de base de données existants.

Pour l'interface utilisateur, nous avons décidé d'utiliser le framework Vue.js [23] et de coder en TypeScript [24]. En effet, Vue.js est facile à apprendre et utiliser. De plus, il possède une large documentation et une communauté active pour nous aider en cas de problème. Nous avons enfin choisi de coder en TypeScript, de façon à avoir un code typé et donc plus clair, plus robuste et plus facile à maintenir pour un projet qui pourra ensuite être utilisé par d'autres personnes.

En synthèse, l'architecture de notre solution est présentée dans le schéma suivant (figure 11) :



*Figure 11 : Schéma fonctionnement de notre solution*

### c. Mise en place de l'API

Dans un premier temps, nous avons travaillé sur l'API, nécessaire pour pouvoir collecter les données qui seront ensuite affichées.

#### Modèle d'entités

Pour pouvoir stocker les données, il a été indispensable de définir au préalable un modèle d'entités (cf. figure 12) permettant de visualiser les rapports entre les différentes entités. Pour la première version de l'API, nous sommes partis sur un modèle avec trois entités :

- *measurements* (mesures) ;
- *measurementsTypes* (types de mesure) ;
- *sensor* (capteur).

Ce modèle nous semblait en effet pertinent dans la mesure où les mesures peuvent être de tout type et sont envoyées depuis différents capteurs. Par ailleurs, un capteur peut avoir différents types de mesures. Par exemple, le DHT22 peut mesurer la température et le taux d'humidité dans l'air.

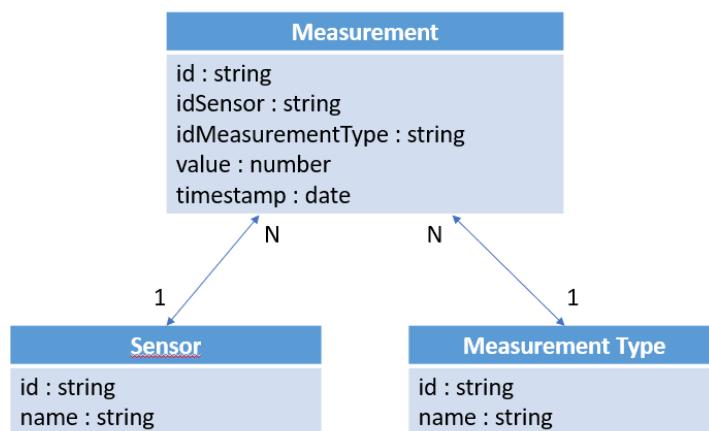


Figure 12 : Modèle d'entité

#### Mise en place des routes de l'API

Nous avons ensuite codé toutes les routes nécessaires au bon fonctionnement de notre API. Pour chacune des trois entités mentionnées précédemment, nous avons mis au point les fonctions :

- GET : pour obtenir la liste des éléments de l'entité choisie
- POST : pour créer un nouvel élément
- PUT : pour mettre à jour un élément
- DELETE : pour supprimer un élément

Par ailleurs, pour l'entité *Measurement*, nous avons ajouté la possibilité d'obtenir des mesures avec PUT en les filtrant par date et/ou type de mesure et/ou capteur.

Nous avons également mis au point une documentation avec Swagger [25] (cf. figure 13) qui précise toutes les routes existantes et les paramètres possibles. Cette documentation est consultable à l'adresse : <https://dev-api.rami.ig.umons.ac.be/api/v1/docs/>. Elle présente aux utilisateurs les différentes requêtes disponibles et décrit également ce qui a été fait pour les personnes qui se serviront de notre projet par la suite.

The screenshot shows the Swagger UI for a RESTful API. It is organized into three main sections: **Sensor**, **MeasurementType**, and **Measurement**.

- Sensor:** Contains four endpoints:
  - GET /sensors**: Get all sensors or sensors by id or sensor by name.
  - POST /sensors**: Create a new sensor.
  - PUT /sensors**: Update a sensor name.
  - DELETE /sensors**: Delete a sensor.
- MeasurementType:** Contains four endpoints:
  - GET /measurementTypes**: Get all measurement type or measurement type by id.
  - POST /measurementTypes**: Create a new measurement type.
  - PUT /measurementTypes**: Update a measurement type name.
  - DELETE /measurementTypes**: Delete a measurement type.
- Measurement:** Contains one endpoint:
  - GET /measurements**: Get all measurement or measurement by id.

*Figure 13 : Capture d'écran de la documentation des différentes requêtes*

## Test de l'API

Une grande partie de notre travail sur l'API a également été la rédaction et l'exécution de tests unitaires, destinés à vérifier que notre code donnait bien les résultats attendus pour toutes les requêtes possibles.

Nous avons choisi de tester notre code avec Jest [26]. Notre choix s'est porté sur cette bibliothèque car elle offre des fonctionnalités puissantes pour créer des suites de tests claires et bien organisées. Elle facilite également la mise en place de scénarios de test complexes en permettant la création de mocks et de simulateurs pour isoler différentes parties du code.

Pour tester notre API, nous avons écrit des tests pour chacune des requêtes potentielles des utilisateurs. A l'aide de “tests positifs”, nous avons vérifié que le résultat obtenu correspond bien à l'attendu, c'est-à-dire que le code HTTP et le message retour sont les bons. Nous avons également écrit des tests négatifs pour nous assurer que les situations d'erreur sont toutes correctement gérées. Cela inclut la vérification des réponses et des codes HTTP appropriés lorsqu'une requête invalide est soumise à l'API, notamment dans le cas de données manquantes, de paramètres incorrects ou dans le cas de conflit.

Certaines requêtes font appel à d'autres fonctions externes à notre API. Pour ne pas retester ces fonctions, nous mockions leurs réponses. Cela signifie que nous simulions leur comportement et choisissions ce qu'elles devaient renvoyer. Cela nous a permis de concentrer nos tests uniquement sur la logique interne de notre API, sans être influencés par le fonctionnement correct ou incorrect des autres fonctions externes.

Une partie essentielle de notre travail de test a été de maintenir tous nos tests à jour au fur et à mesure de l'évolution du code. Ainsi, à chaque fois que nous apportions des modifications au code, nous modifions les tests correspondants pour qu'ils restent corrects et nous les complétons si nécessaire. Cela nous a permis tout au long du projet de nous assurer que notre API était toujours opérationnelle et avait bien toutes les fonctionnalités que nous souhaitions.

Ainsi, 100% de notre code a été testé, comme le montre la figure 14.

File	% Stmt	% Branch	% Funcs	% Lines	Uncovered Line #s
<code>user.ts</code>	100	100	100	100	
<code>middlewares</code>	100	100	100	100	
<code>auth.ts</code>	100	100	100	100	
<code>routes</code>	100	100	100	100	
<code>auth.ts</code>	100	100	100	100	
<code>home.ts</code>	100	100	100	100	
<code>measurement.ts</code>	100	100	100	100	
<code>measurementType.ts</code>	100	100	100	100	
<code>routes.ts</code>	100	100	100	100	
<code>sensor.ts</code>	100	100	100	100	
<code>testRoutes.ts</code>	100	100	100	100	
<code>user.ts</code>	100	100	100	100	

```
Test Suites: 7 passed, 7 total
Tests:     189 passed, 189 total
Snapshots: 0 total
Time:      10.524 s
Ran all test suites.
```

Figure 14 : Tests de l'API

### Hébergement et déploiement de l'API

Pour pouvoir avoir des runners, nous avons hébergé ce projet sur le GitLab du cluster privé du service de l'UMONS. Le code est disponible ici : <https://gitlab.ig.umons.ac.be/rami/umons-sensor-backend>. Le lien du code est également disponible en [Annexe 1](#).

Pour le déploiement, nous avons utilisé les CI/CD de GitLab et nous avons écrit un script qui déploie automatiquement notre API lorsque nous modifions certaines branches. Cela permet d'automatiser le processus afin de limiter les erreurs qui peuvent arriver lors d'un déploiement manuel et de gagner du temps. Les différentes étapes du déploiement de notre API sont présentées sur la figure 15 ci-dessous.

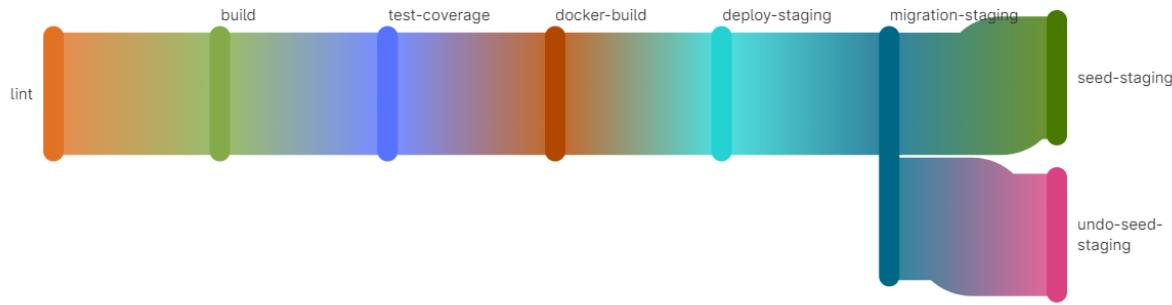


Figure 15 : Pipeline de déploiement de l'API

Tout d'abord l'étape de *lint* vérifie que notre code vérifie les normes de formatage. Ensuite, l'étape de *build* permet de transformer notre code dans une version exécutable et déployable.

L'étape suivante, *test-coverage*, permet de vérifier que les tests unitaires sont bien exécutés. Après cela, lors de l'étape *docker-build*, nous procédons à la création du conteneur Docker qui contient notre API. Ce conteneur est essentiel pour garantir la portabilité et la gestion efficace de notre application.

Enfin, nous arrivons à l'étape de déploiement proprement dite, où le conteneur Docker est déployé dans l'environnement de production. Avant de mettre en service la nouvelle version de l'API, des migrations de base de données sont exécutées pour assurer la cohérence entre la structure de la base de données et les modifications apportées par la mise à jour de l'API.

Ce processus en plusieurs étapes garantit que chaque modification de code est soumise à des contrôles rigoureux avant d'être déployée, ce qui assure la stabilité et la fiabilité de l'API.

L'API est disponible à l'adresse <https://dev-api.rami.ig.umons.ac.be/api/v1/>. On peut par exemple voir à ce lien : <https://dev-api.rami.ig.umons.ac.be/api/v1/sensors> l'ensemble des capteurs enregistrés.

#### d. Envoi de données à partir d'un capteur

Après avoir codé et déployé l'API, nous avons voulu tester l'envoi de données directement depuis un capteur relié à un ESP32. Nous avons choisi d'utiliser le protocole Wifi et non un protocole bas débit comme dans la version finale de RAMi pour simplifier cette étape de communication. Pour cela, nous avons utilisé un DHT22 (comme présenté dans la partie précédente) et un capteur cardiaque, RAMi ayant vocation à servir dans le domaine de la santé.

Sur ce point, j'ai travaillé avec Timothé BRENIER, stagiaire dans le même service que moi et étudiant en 4ème année en électronique à l'école Polytech Lille.

Nous avons tout d'abord choisi un capteur cardiaque, le AD8232 [27], pour son faible coût et sa simplicité d'utilisation.

Nous avons ensuite écrit le code Arduino qui récupère les mesures du capteur et les envoie à l'API au format que nous avons défini.

Pour cela, nous avons dû utiliser plusieurs bibliothèques Arduino :

- Wifi.h qui permet à la carte de se connecter au Wifi pour envoyer des données ;
- ArduinoJson.h [28] qui permet de créer un Json pour pouvoir ensuite l'envoyer à l'API ;
- HTTPClient.h [29] qui permet de faire une requête HTTP.

Le code est disponible en [Annexe 2](#). Il concerne un DHT22 qui envoie des données sur la température et le pourcentage d'humidité dans l'air toutes les secondes à notre API.

#### e. Visualisation des données et partie front

Une fois les données collectées, nous avons développé une application Vue.js pour visualiser les données des différents capteurs.

Pour cela, nous avons codé une page avec une barre de recherche où l'utilisateur peut taper le nom du capteur dont il souhaite voir les données (cf. figure 16).

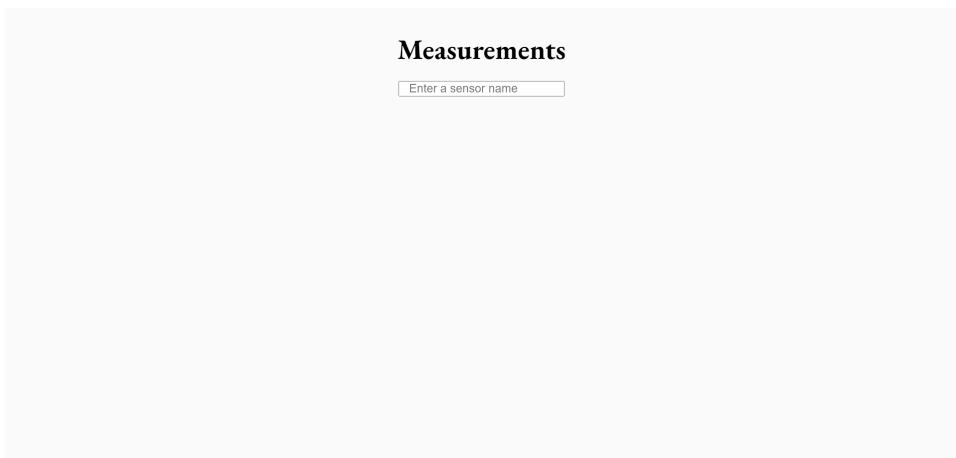


Figure 16 : Page d'accueil du site

Au fur et à mesure que l'utilisateur tape le nom du capteur recherché, la liste des capteurs affichés se réduit de façon à ne présenter que les capteurs dont le nom contient la chaîne de caractères saisie (cf. figure 17).

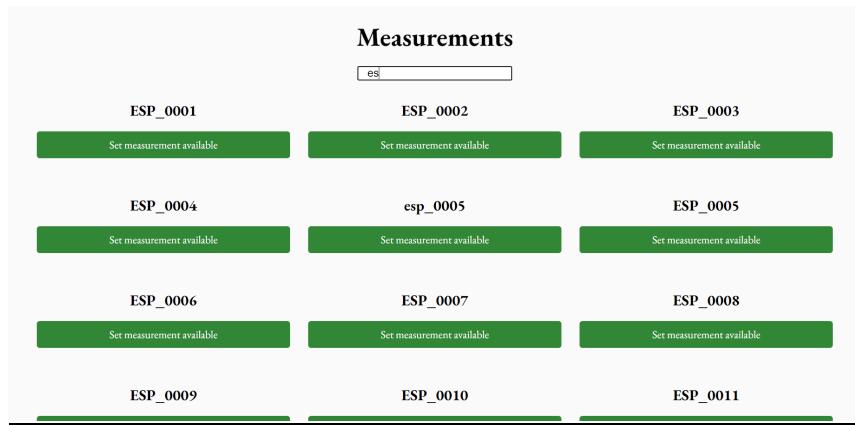


Figure 17 : Recherche d'un capteur

Puis, si l'utilisateur clique sur le bouton "Set measurement available" sous le nom d'un capteur, la liste des différents types de mesures que ce capteur a effectuées s'affiche (cf. figure 17).

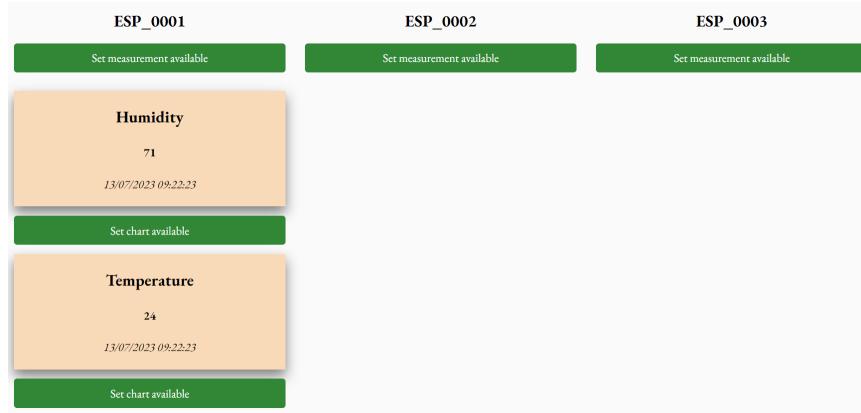


Figure 18 : Affichage du type de mesure des données envoyées par un capteur

Si l'utilisateur clique sur une carte, celle-ci grandit et un tableau avec les 5 dernières valeurs envoyées s'affiche (cf. figure 19).



Figure 19 : Affichage du tableau des 5 dernières valeurs envoyées

Enfin, si l'utilisateur clique sur le bouton "Set Chart available", un graphique avec les dernières valeurs envoyées s'affiche à l'écran (cf. figure 19). Pour le graphique, nous avons choisi d'utiliser la bibliothèque Chart.js [30]. En effet, elle permet de tracer simplement des graphiques de tout type entièrement personnalisables. De plus, elle est très bien documentée.

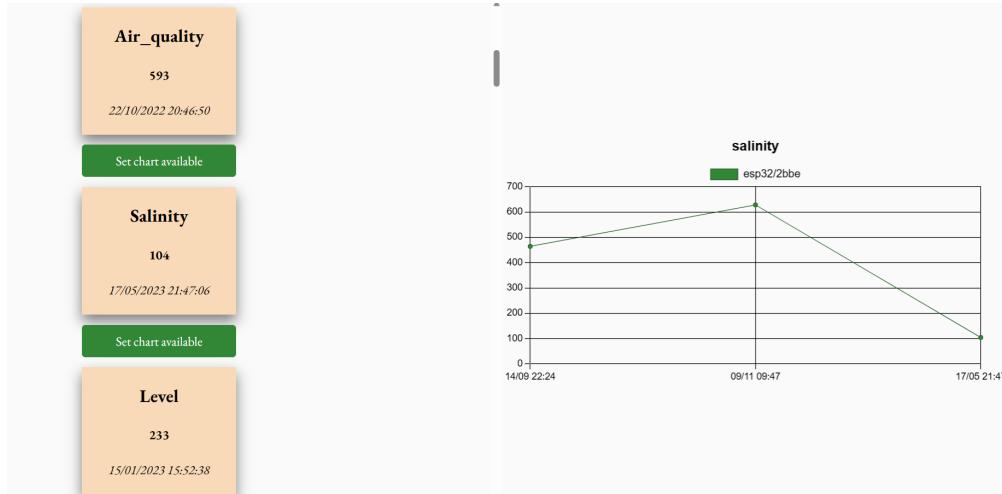


Figure 20 : Affichage du graphique des dernières valeurs envoyées

L'application possède également un mode sombre (cf. figure 21).

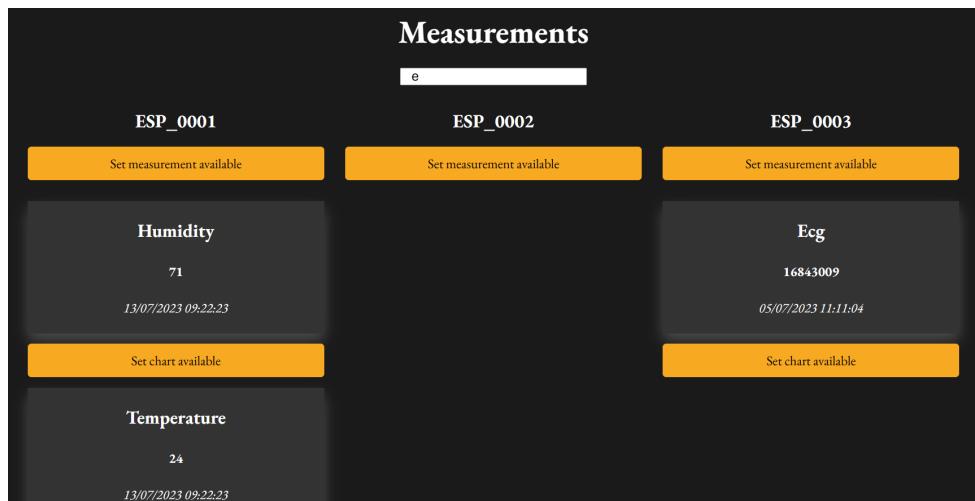


Figure 21 : Mode sombre de l'application

Enfin, la base de données pouvant contenir plusieurs millions de données, pour éviter que le site ne prenne trop de temps à s'afficher, nous avons recherché les modalités les plus efficaces pour le chargement des données. Le principe que nous avons retenu est de ne précharger aucune donnée, et de récupérer uniquement les données à afficher à l'écran lorsque l'utilisateur fait une action qui nécessite d'afficher de nouveaux éléments.

Le code est disponible ici : <https://gitlab.ig.umons.ac.be/rami/umons-sensor-frontend> et le lien est également noté en [Annexe 3](#).

## **IV. Imperfections rencontrées et solutions apportées**

Après avoir réalisé cette première version de l'API et de l'interface visuelle, nous avons relevé plusieurs points à améliorer, en particulier :

- les graphiques s'affichent sans choix de paramétrage pour l'utilisateur ce qui peut poser problème car tous les capteurs ne nécessitent pas une visualisation similaire. De plus, il n'y a aucune information pour aider à bien lire les données ;
- les données ne peuvent être envoyées que l'une après l'autre ce qui pose problème quand certains capteurs nécessitent d'envoyer deux données presque simultanément ;
- pour le moment toutes les données sont accessibles à tout le monde sans aucune restriction.

Dans cette partie, nous allons détailler ces trois points, ainsi que les axes d'amélioration envisagés et les solutions que nous avons déployées.

### **1. Choix des paramètres sur le graphique et informations sur les valeurs**

Comme expliqué ci-dessus, dans la première version de l'interface, il n'était pas possible de paramétriser l'affichage du graphique de suivi de l'évolution des mesures envoyées par un capteur. Ce sont les dix dernières valeurs qui étaient toujours affichées.

#### **a. Echantillonnage**

Ce mode de fonctionnement ne permettait pas de restituer de façon optimale les différentes mesures à suivre à terme dans le cadre de RAMi. En effet, leurs variations dans le temps sont très différentes. Ainsi, la température du corps humain évolue peu à l'échelle d'une heure, contrairement à la tension dans un électrocardiogramme qui, elle, varie beaucoup au sein même d'une seconde. Or, certains capteurs de température peuvent envoyer des valeurs à intervalle très court. Il était dès lors nécessaire de pouvoir échantillonner les valeurs pour pouvoir afficher les dernières valeurs prises à un intervalle cohérent avec la variation de la mesure.

Ainsi, nous avons implémenté différents laps de temps permettant d'échantillonner à des intervalles différents. Nous avons choisi d'être le plus exhaustifs possible et de nous adapter à tout type de capteur et de mesure, de façon à ce que notre API et notre site puissent servir pour des domaines autres que le médical, avec des variations de mesures faibles à l'échelle d'une année ou d'une décennie. Les intervalles que nous avons retenus sont:

- la seconde ;
- la demi-minute ;
- la minute ;
- le quart d'heure ;
- la demi-heure ;

- l'heure ;
- la journée ;
- la semaine ;
- les deux semaines ;
- le mois ;
- le trimestre ;
- la demie année ;
- l'année;
- la décennie;
- le siècle.

Nous avons implémenté ces différents intervalles dans le back. Il est alors possible d'obtenir les dernières mesures échantillonnées à l'échelle de temps choisi lors d'une requête GET.

Nous avons ensuite modifié le front pour que l'utilisateur puisse choisir son temps d'échantillonnage et que les modalités d'affichage du graphique s'adaptent à la sélection de l'utilisateur. Ainsi, le graphique s'affiche initialement comme dans la version initiale(les dernières valeurs). Puis, l'utilisateur peut choisir un *Sample* (échantillon) parmi la liste des possibilités mentionnées ci-dessus. Le graphique affiche alors les dernières valeurs en respectant l'échantillonnage choisi.

L'exemple ci-dessous montre comment l'utilisateur peut choisir un échantillonnage.

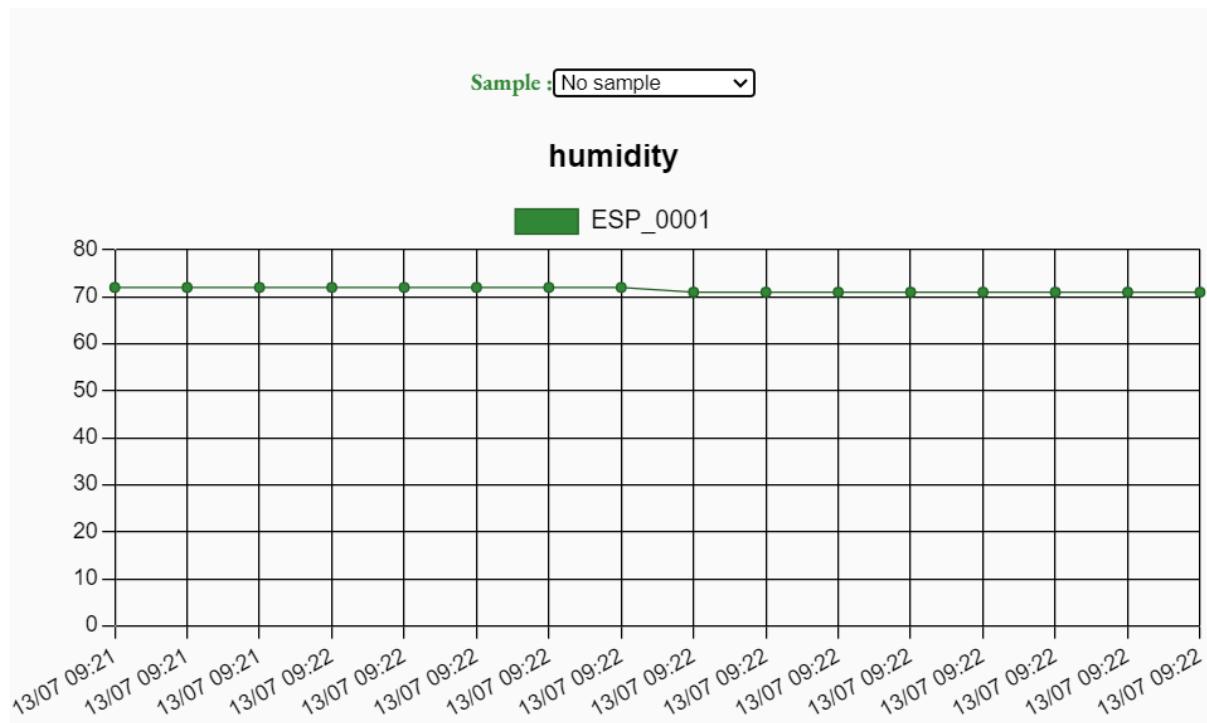
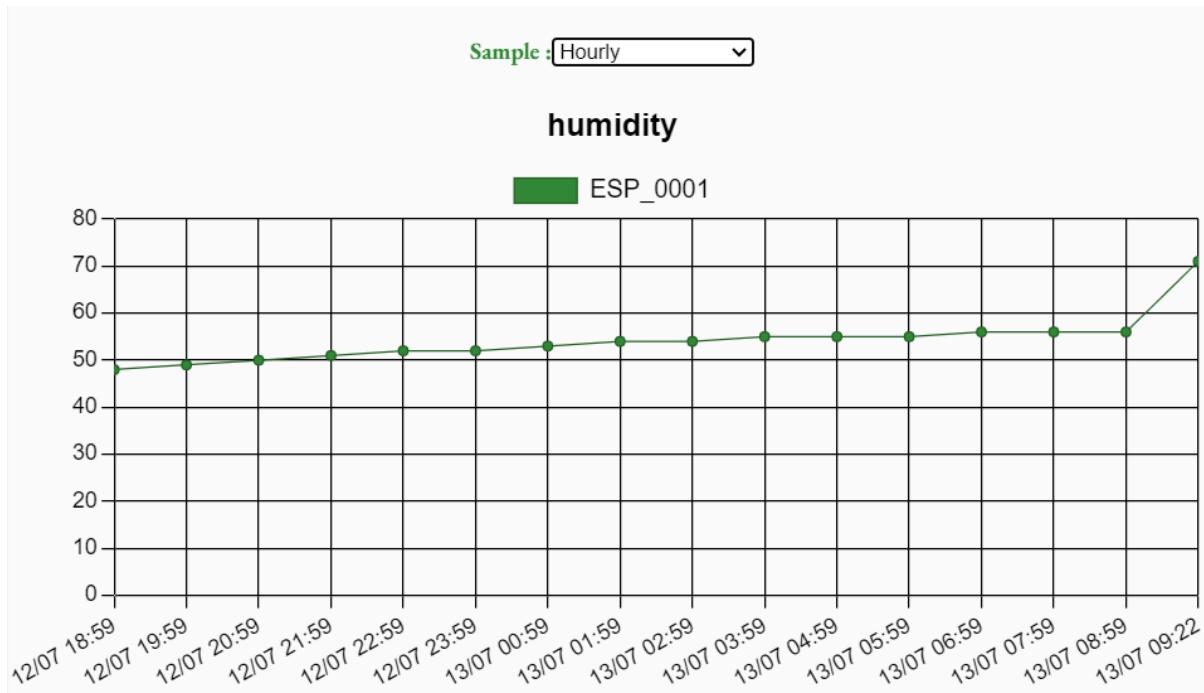


Figure 22 : Evolution du taux d'humidité dans l'air sans échantillonnage

La figure 22 montre l'évolution du taux d'humidité dans l'air sans échantillonnage. Les 16 dernières valeurs sont donc affichées sur le graphique. Or, le capteur envoie plusieurs

données par minute alors que l'humidité évolue très peu dans ce laps de temps. Il est dès lors pertinent de choisir un échantillon pour pouvoir mieux visualiser les variations du taux d'humidité au cours du temps.



*Figure 23 : Evolution du taux d'humidité dans l'air avec un échantillonnage de une heure*

Le taux d'humidité peut varier d'une heure sur l'autre. En ajoutant un échantillonnage d'une heure, les 16 valeurs affichées sur la figure 23 sont beaucoup plus pertinentes pour étudier les variations de cette mesure.

#### b. Nombres de points affichés

Dans la première version du site, les utilisateurs ne peuvent pas non plus choisir le nombre de mesures affiché sur le graphique. Cela peut engendrer des problèmes pour une bonne visualisation. En effet, un électrocardiogramme nécessite de nombreux points pour être exploitable alors que peu de points suffisent à la bonne lecture d'un graphique montrant l'évolution de la température d'un patient.

Ainsi, nous avons implémenté une fonctionnalité permettant de choisir le nombre de mesures récupérées lors d'une requête GET.

Pour l'interface utilisateur, nous avons choisi de proposer à l'utilisateur un menu déroulant, sur le même principe que pour l'échantillonnage, qui permet de sélectionner le nombre de points parmi une liste. Pour avoir un panel de choix pertinent quel que soit le type de capteur, nous sommes partis sur l'échelle des 2<sup>n</sup> avec n allant de 0 à 13.

L'exemple ci-dessous montre comment l'utilisateur peut utiliser l'outil de sélection du nombre de points.

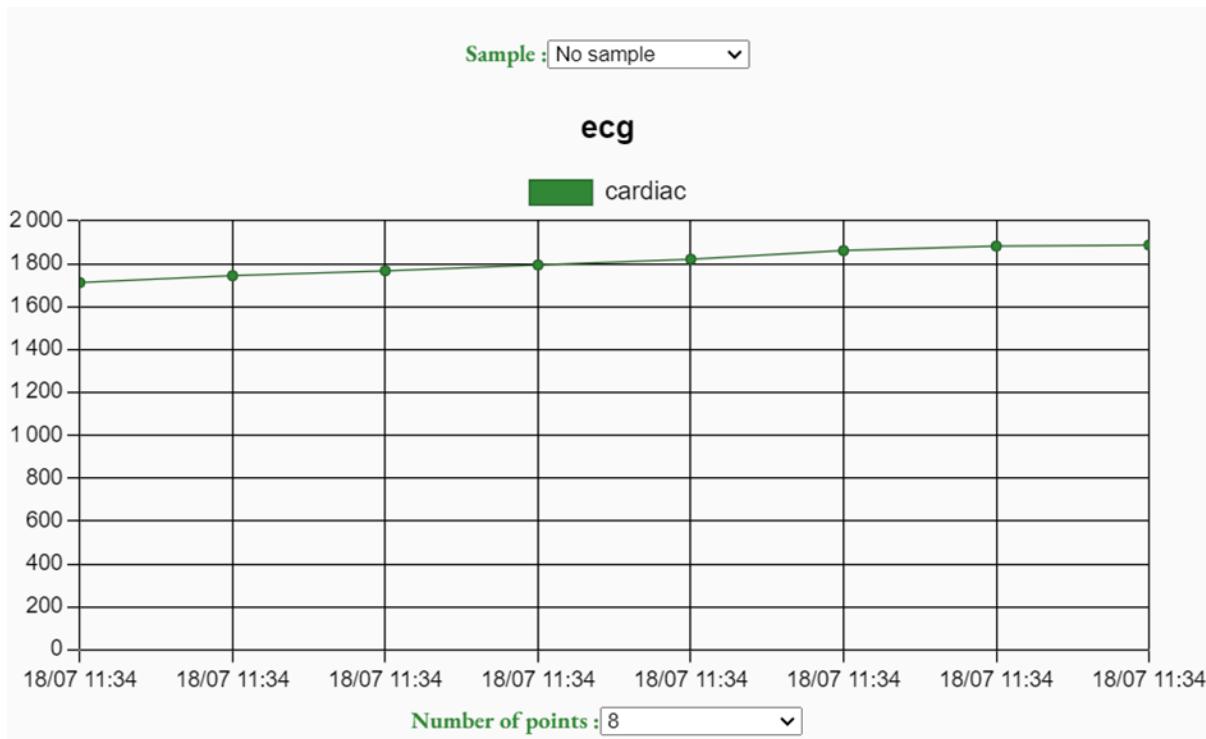


Figure 24 : Électrocardiogramme tracé avec 8 points

Sur la figure 24, on observe les 8 dernières valeurs d'un électrocardiogramme. Or, ce nombre de point n'est pas suffisant pour pouvoir bien le visualiser

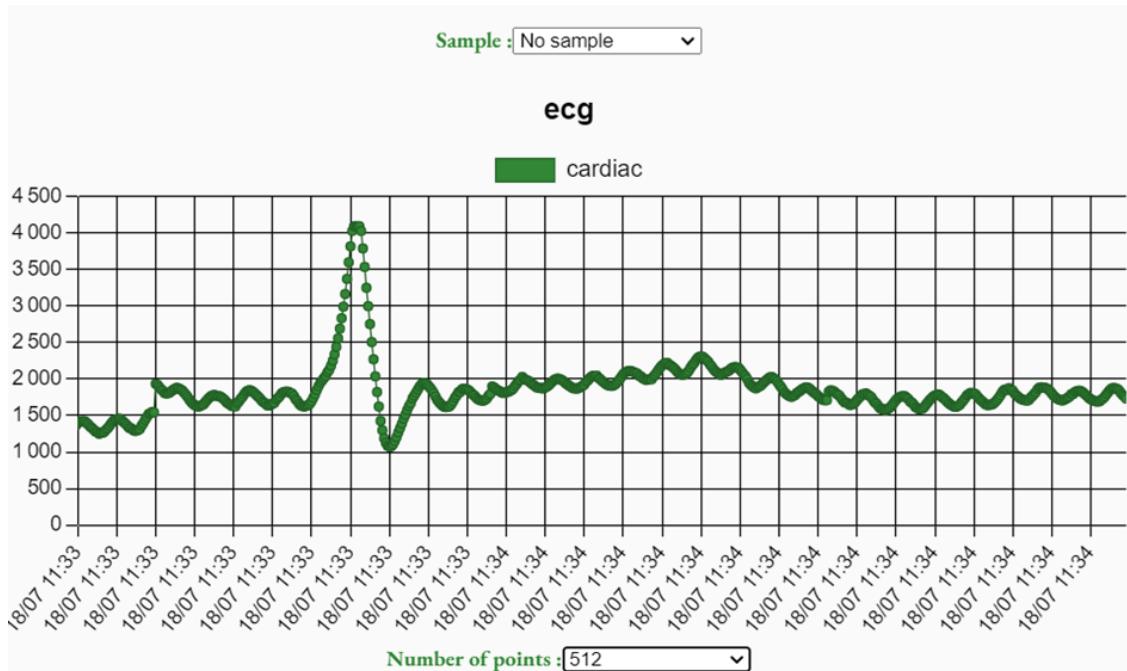


Figure 25 : Électrocardiogramme tracé avec 512 points

En augmentant ce nombre de points à 512 comme sur la figure 25, il est alors possible de bien visualiser l'électrocardiogramme et toutes les phases d'onde qui le forment.

### c. Possibilité de zoomer sur le graphique

Le dernier point à améliorer sur le graphique est la possibilité de zoomer. En effet, il est parfois intéressant de pouvoir zoomer sur une zone pour avoir plus de précision sur les mesures à un instant donné.

Nous avons choisi d'inclure le plugin “chartjs-plugin-zoom” [31] à notre interface visuelle pour pouvoir zoomer sur le graphique. Nous avons limité cette possibilité au zoom horizontal. En effet, le zoom vertical nous semblait peu pertinent pour de l'affichage de données et rend le zoom horizontal plus compliqué.

### d. Ajout d'informations sur les valeurs du graphique

Il est désormais possible pour l'utilisateur de visualiser le graphique comme il le désire. Cependant, il n'y a pour le moment aucune information pour l'aider à bien comprendre les mesures qu'il consulte. C'est pourquoi, nous avons choisi d'afficher la valeur minimale, moyenne et maximale des données affichées sous le graphique. Cela permet d'avoir des informations précises.

La figure 26, ci-dessous, montre ces informations sur un graphique montrant l'évolution de la température dans une pièce.

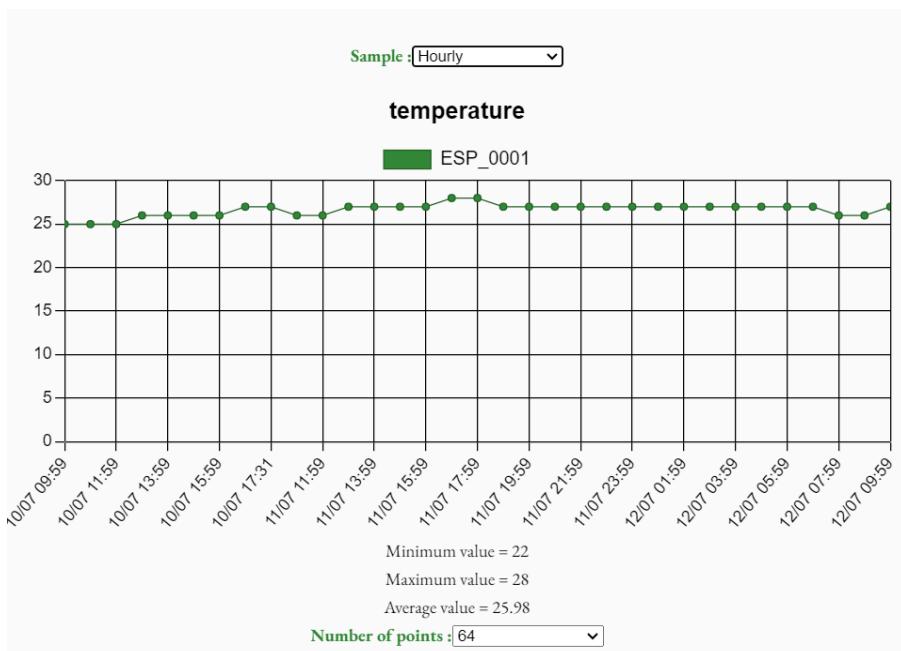


Figure 26 : Capture d'écran montrant l'affichage des informations complémentaires

## 2. Envoi par batch

Le second axe d'amélioration que nous avons noté à l'issue de la première version de l'API concerne le nombre de mesures que l'on peut envoyer à partir des capteurs. En effet, cette version ne permettait d'en envoyer que l'une après l'autre. La requête devait être une requête POST avec un JSON de la forme :

```
{  
    sensor : "XXX",  
    type : "XXX",  
    date : "AAAA-MM-JJTHH:mm:ss.iiiZ"  
    value:NN  
}
```

Cela pouvait poser problème par exemple pour envoyer des données pour un électrocardiogramme. En effet, un électrocardiogramme nécessite beaucoup de mesures par seconde pour être bien visualisé. Or, lorsque la carte reliée au capteur fait une requête HTTP, il y a un délai de latence pour recevoir la réponse et pouvoir faire une autre requête. Les électrocardiogrammes ainsi obtenus ont trop de données manquantes et ne sont pas exploitables.

L'idée que nous avons eue est de pouvoir envoyer des batchs de valeurs c'est à dire plusieurs valeurs d'un coup pour minimiser les données manquantes que l'on peut avoir dans l'électrocardiogramme. Pour cela, il fallait adapter l'API et créer une nouvelle route permettant l'envoi de ces batchs. Dans la suite de cette partie, nous détaillerons les solutions envisagées ainsi que leurs avantages et inconvénients. Puis nous indiquerons la solution retenue et les raisons de ce choix.

#### a. Vérification des envois un par un

La première option que nous avons envisagée est d'envoyer une liste des différentes valeurs au format précédent. Ainsi, notre programme lit un par un les éléments de la liste et fait une requête POST classique avec chacun d'eux. Les éléments qui vérifient le bon format et qui ont un capteur et un type de données valides sont ajoutés à la base de données alors que les autres, eux, ne le sont pas, et leurs messages d'erreur sont compilés les uns après les autres et affichés à la fin de la requête.

L'avantage de cette solution est qu'elle permet de bien afficher les erreurs, de sorte que l'utilisateur peut voir lesquelles de ses données ont été bien envoyées.

Cependant, cette méthode nous a permis de voir des limites à l'envoi de données par batch. Tout d'abord l'espace mémoire sur la carte est limité et permet de stocker un tableau d'une centaine de données. De plus, cette manière de procéder rend la requête HTTP très longue. En moyenne, envoyer une liste de 100 valeurs prend près d'une seconde (960ms). Ainsi, les valeurs contenues dans un tableau montrent un début d'électrocardiogramme de bonne qualité, mais il manque près d'une seconde de valeur entre deux envois de listes consécutifs. Cela ne permet pas de bien visualiser l'électrocardiogramme.

### b. Sans vérification des envois

Nous avons par la suite réfléchi et implémenté une deuxième solution. Dans cette version, l'utilisateur doit envoyer directement les valeurs au format dans lequel elles sont stockées dans la base de données. Soit le format suivant :

```
timestamp : Date  
value : int  
idSensor : uuid  
idMeasurementType : uuid
```

Cette version peut poser des problèmes, notamment car elle nécessite que l'utilisateur connaisse les identifiants qui sont sous la forme d'un UUID. Par ailleurs, cette manière d'écrire prend beaucoup plus de caractères que la précédente. Ainsi, le nombre de valeurs que la carte permet de stocker et d'envoyer est plus faible. Enfin, en cas d'erreur, la requête ne renvoie pas de messages explicatifs. L'utilisateur ne sait pas si c'est parce que toutes ses données sont au mauvais format ou s'il y a seulement eu un problème avec l'une d'entre elle.

Cependant, cette méthode possède un grand avantage : elle est beaucoup plus rapide. En effet, elle permet de gagner du temps en ne faisant pas la conversion entre le nom du capteur et son identifiant et le nom du type de mesure et son identifiant. Les valeurs sont directement insérées et, si il y a une erreur, aucune des valeurs n'est ajoutée à la base de données. Avec des tests de vitesse, nous avons trouvé qu'en moyenne, une requête pour envoyer 100 valeurs prend 16ms. Cela permet d'obtenir des ECG exploitables.

### c. Vérification groupée

Enfin, nous avons pensé à une dernière solution pour tenter de résoudre le problème. Comme dans la première solution présentée (en partie a. Vérification des envois un par un), l'utilisateur envoie une liste de valeurs au format présenté au début de la partie. Puis, notre programme crée deux listes. La première sert à stocker tous les capteurs qui sont présents dans la liste des mesures ainsi que les positions dans la liste des mesures ayant ce capteur tandis que la seconde stocke tous les types de mesures ainsi que les positions dans la liste des mesures ayant ce type de mesure.

Ensuite, on vérifie que les capteurs et les types de mesures existent bien dans nos bases de données et on ne vérifie plus que le format de la date et de la valeur pour insérer une valeur ou non.

En cas d'erreur sur un capteur, un type de mesure, une valeur ou une date, le message d'erreur est compilé avec la position dans la liste de la mesure incorrecte.

A la fin, le message d'erreur est affiché ainsi que le nombre de mesures qui ont été créées sur le nombre de mesures qui étaient contenues dans la liste initialement envoyée.

En faisant des tests de vitesse, nous avons pu noter que la vitesse moyenne pour envoyer une liste de 100 valeurs est de 22 ms. En effet, on envoie souvent un tableau de 100 valeurs mesurées du même capteur et d'un ou deux types de mesures. On gagne donc beaucoup de temps à ne pas vérifier à chaque fois les mêmes informations.

#### d. Choix de la méthode

Ainsi, pour choisir la méthode la plus adéquate, nous avons dressé un tableau récapitulatif des différents critères d'évaluation des méthodes :

	Méthode "a"	Méthode "b"	Méthode "c"
Temps moyen d'envoi de 100 mesures	960 ms	16 ms	22 ms
Nombre de caractères moyen d'une mesure (taille de la requête)	88	160	88
Format clair pour envoyer les données pour l'utilisateur	Oui	Non (nécessite de connaître les UUID)	Oui
Message d'erreur explicite	Oui	Non	Oui

*Figure 27 : Tableau comparatif des différentes solutions*

Du fait du temps d'envoi beaucoup trop important de la méthode "a" (plus de 40 fois plus lent que les deux autres solutions) et vu l'importance de ce critère nous avons directement écarté cette hypothèse.

Il nous restait alors le choix entre la méthode "b" et la méthode "c". La méthode "c" est certes plus longue que la méthode "b" (elle prend environ un tiers de plus en termes de temps) mais elle permet d'envoyer des listes de valeurs presque deux fois plus grandes. De plus, la méthode "c" est plus simple pour l'utilisateur. En effet, elle affiche des messages d'erreurs et ne nécessite pas que celui-ci connaisse l'identifiant des capteurs et des types de mesure.

Pour toutes les raisons mentionnées ci-dessous nous avons choisi de conserver la méthode présentée en partie c. Cette méthode nous permet d'obtenir un ECG où l'on peut voir les différentes phases (cf. figure 16).

### 3. Sécurité

Dans la première version du code, tout le monde pouvait faire des requêtes à l'API, que ce soit pour consulter la liste des capteurs enregistrés, supprimer une mesure ou autre. Ceci était volontaire car nous ne connaissions pas encore la politique souhaitée en termes de sécurité. Mais, puisque ce sont des données médicales, et donc des données sensibles, qui seront à terme transmises, il n'était pas souhaitable de garder un fonctionnement aussi ouvert sur le long terme.

Après discussions et réflexions, nous avons choisi d'ajouter un système d'utilisateur et de connexion. Chaque utilisateur aura des droits limités sur ce qu'il peut faire ou non. Il ne pourra accéder à tous les capteurs et obtenir, modifier ou supprimer des données de quelqu'un d'autre.

### a. Changement de modèle d'entités

Afin d'implémenter notre nouvelle solution, nous avons dû réfléchir à un nouveau modèle d'entités incluant des utilisateurs.

Nous avons souhaité créer trois rôles d'utilisateurs avec plus ou moins de droits :

- admin : autorisé à voir les données de tous les capteurs et à modérer les différentes demandes des utilisateurs ;
- privileged et regular : autorisés à voir uniquement les données des capteurs auxquels ils ont été préalablement habilités par un administrateur. Pour avoir accès aux données d'un capteur ou pouvoir créer un nouveau capteur ou un type de mesure, ils doivent demander à un administrateur.

Nous avons créé deux rôles différents pour privileged et regular pour le cas où, à l'avenir, il deviendrait nécessaire de différencier leurs droits.

Pour pouvoir implémenter ces différents niveaux d'utilisateurs, nous sommes repartis du diagramme d'entité présenté en figure 12 tout en ajoutant 4 nouvelles entités (cf. figure 28).

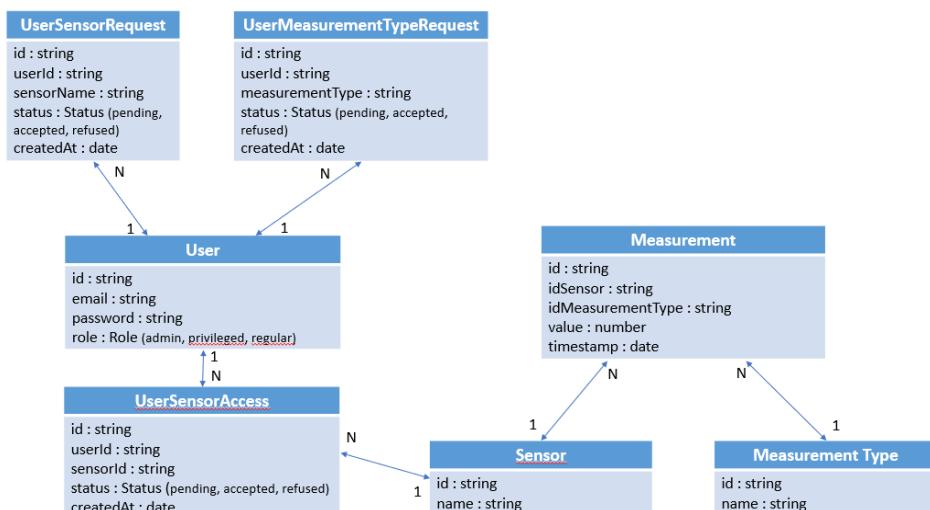


Figure 28 : Schéma d'entité incluant la sécurité

Les nouvelles classes sont les suivantes :

- User : permet de définir les différents utilisateurs et leurs rôles;
- UserSensorAccess : permet de définir quels utilisateurs ont accès à quels capteurs;
- UserSensorRequest : permet à un utilisateur de faire la demande de création de capteur;
- UserMeasurementTypeRequest : permet à un utilisateur de faire la demande de création de type de mesure.

## b. Implémentation de la solution

Par la suite, nous avons implémenté la solution explicitée ci-dessus aussi bien pour le back que pour le front.

Pour le back, nous avons ajouté les différentes classes. Nous avons également écrit les différentes requêtes possibles avec ces classes et modifié les requêtes précédentes en incluant la connexion. Celle-ci est gérée à l'aide d'un token de connexion qui est utilisé dans toutes les requêtes pour savoir si l'utilisateur a le droit de faire cette requête ou non. Ce token expire après un certain temps d'inactivité.

Toutes les routes sont presque intégralement testées (cf. figure 29).

File	% Stmt	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	97.23	92.75	94.91	97.43	
controllers	98.18	93.81	94.54	98.4	
home.ts	100	100	100	100	
measurement.ts	97.62	95.55	84.21	98.28	202, 250-260, 275
measurementType.ts	100	100	100	100	
sensor.ts	92.96	69.84	100	92.96	129, 144, 220, 262-268, 273, 284
user.ts	100	100	100	100	
userMeasurementType.ts	100	100	100	100	
userSensor.ts	100	100	100	100	
middlewares	75.55	50	100	74.41	
auth.ts	75.55	50	100	74.41	51-54, 61-66, 70-81
routes	100	100	100	100	
auth.ts	100	100	100	100	
home.ts	100	100	100	100	
measurement.ts	100	100	100	100	
measurementType.ts	100	100	100	100	
routes.ts	100	100	100	100	
sensor.ts	100	100	100	100	
testRoutes.ts	100	100	100	100	
user.ts	100	100	100	100	

```
Test Suites: 9 passed, 9 total
Tests:      279 passed, 279 total
Snapshots:  0 total
Time:       8.681 s, estimated 9 s
```

Figure 29 : Résultat des tests pour l'API avec l'ajout des utilisateurs

Pour le front, nous avons ajouté les différentes pages nécessaires. Ainsi, pour accéder au menu principal, l'utilisateur doit se connecter. Si il n'a pas de compte, il doit s'inscrire (cf. figure 30).

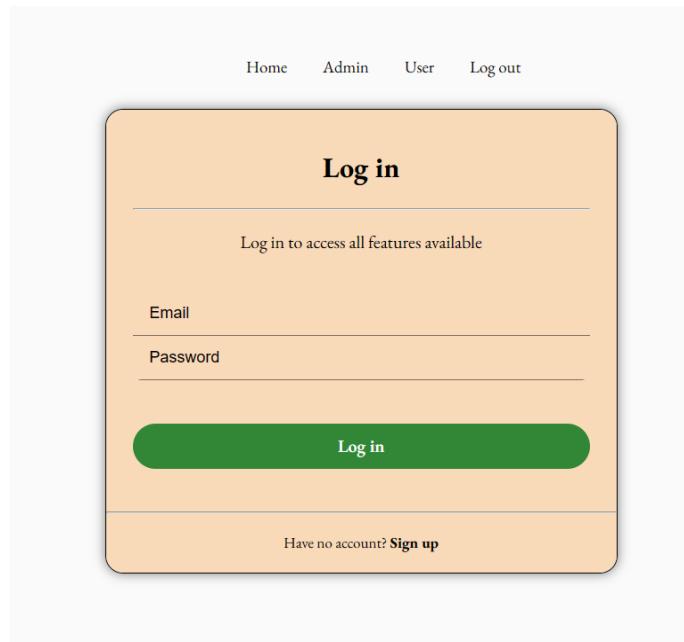


Figure 30 : Formulaire de connexion

Une fois connecté, sur la page principale, il ne peut voir que les capteurs qu'il est autorisé à voir si son rôle est regular ou privileged. Sur la page de l'onglet *User* (cf. figure 31), il peut demander à un administrateur d'avoir le droit de voir les données un capteur. Il peut également demander la création d'un nouveau capteur ou d'un nouveau type de mesure si celui-ci n'existe pas déjà.

Figure 31 : Page permettant de demander la création d'un capteur, d'un type de mesure ou l'accésion à un capteur

Enfin, lorsqu'un administrateur se connecte, sur la page principale, il peut voir tous les capteurs et leurs mesures. Il peut également accéder à l'onglet *Admin* qui sert pour la modération. Sur celui-ci, il peut voir les différentes demandes des utilisateurs pour accéder à un capteur ou créer un capteur ou un type de mesure. Il peut alors les accepter ou les refuser.

The screenshot shows a web-based administrative interface. At the top, there is a navigation bar with links for 'Home', 'Admin', 'User', and 'Log out'. Below this, the word 'Admin' is centered above a horizontal line. Underneath the line, the text 'Select Component: Sensor creation' is followed by a dropdown menu. The main area is titled 'Admin Request' and contains the sub-instruction 'Grant or revoke sensor creation'. A filter option 'Filter by status Show All' is present. Below these, a table lists a single request:

User	Sensor	Status	Created at	Accept	Reject
thomas@ig.umons.ac.be	cardiac_12	pending	29/08/2023 12:36	<button>Accept</button>	<button>Rejected</button>

*Figure 32 : Page de modération pour les administrateurs*

Cependant, à l'heure actuelle, cette version n'a pas encore été déployée. En effet, d'autres options sont envisagées en complément pour sécuriser les données dont l'implémentation d'une blockchain. Pour ce faire, la version avec la connexion pourra être déployée quand l'ajout de celle-ci aura débuté.

## Conclusion et perspectives

En conclusion, nous avons réussi à déployer un système permettant de collecter des données issues de tous types de capteurs et de les afficher. L'affichage est personnalisable par l'utilisateur pour qu'il puisse choisir la meilleure visualisation possible : nombre de points, échantillonnage, zoom. Cette brique logicielle pourra être utilisée et intégrée à RAMi.

Cependant, pour que l'architecture RAMi soit entièrement développée, il reste encore des actions à mener, notamment le développement d'algorithmes de *Federated Learning* pour étudier en temps réel les données collectées. Plusieurs personnes du service, notamment un doctorant dans le cadre de sa thèse, travaillent actuellement sur cette partie du projet.

La partie sur l'intégration de la sécurité est encore en cours. Il reste à déployer la partie sur les utilisateurs et la connexion. Par ailleurs, il est prévu d'intégrer la blockchain dans RAMi afin de sécuriser les données en les authentifiant. J'ai pu discuter avec un doctorant qui vient de commencer sa thèse dans le service et qui a déjà effectué son travail de fin d'étude sur la blockchain. Il sera chargé d'implémenter la blockchain sur notre API.

Enfin, au cours de mon stage, j'ai pu discuter avec une personne travaillant sur le format FHIR (Fast Healthcare Interoperability Resources - Ressources d'interopérabilité rapide des soins de santé) [32]. Il s'agit d'un format de données qui a pour but de standardiser le format des données dans le domaine médical afin de faciliter leur transmission entre différentes plateformes. Il serait donc intéressant de modifier le format des données collectées par RAMi afin qu'elles respectent ce format.

Par ailleurs, ce stage m'a beaucoup appris et m'a permis de confirmer mon attrait pour l'informatique et la programmation. Toutes les notions abordées m'ont beaucoup intéressé et apprendre et chercher dans ce domaine tout au long de ces 5 mois m'a confirmé que je souhaitais orienter ma carrière professionnelle vers ce domaine.

De plus, il m'a également permis de mettre en pratique de nombreuses notions vues en cours tout au long de ma formation à Centrale. Cela m'a aidé à mieux appréhender et comprendre certains concepts aussi bien en informatique, électronique et gestion de projet.

Enfin, réaliser le déploiement complet d'un site et d'une API en ne partant de rien a été très gratifiant et formateur. En effet, j'ai pu appréhender les différentes étapes que nécessite ce travail, depuis la phase de conception au déploiement, ainsi que les difficultés concrètes auxquelles on peut être confronté. Cela m'a beaucoup aidé à gagner en confiance dans ma capacité à réaliser une mission de bout en bout.

## Bibliographie

- [1] - Site de l'UMons : <https://web.umons.ac.be/fr/>
- [2] - Site de l'Infortech : <https://web.umons.ac.be/infortech/fr/>
- [3] - Site du service ILIA : <https://web.umons.ac.be/ilia/>
- [4] - Rapport sur le vieillissement de la société française et européenne :  
[https://www.gouvernement.fr/sites/default/files/contenu/piece-jointe/2023/02/hcp\\_vieillissement\\_de\\_la\\_societe\\_francaise.pdf](https://www.gouvernement.fr/sites/default/files/contenu/piece-jointe/2023/02/hcp_vieillissement_de_la_societe_francaise.pdf)
- [5] - Étude montrant l'évolution des maladies en fonction de l'âge :  
<https://www.insee.fr/fr/statistiques/4238405?sommaire=4238781#:~:text=Avec%20l%C3%A0%20perception,58%20%25%20%C3%A0%2059%20%25>.
- [6] - Potential of Internet of Medical Things (IoMT) applications in building a smart healthcare system: A systematic review : <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8664731/>
- [7] - Article de présentation de RAMi : <https://www.mdpi.com/2078-2489/13/9/423>
- [8] - Site d'Eclipse Mosquitto : <https://mosquitto.org/>
- [9] - Site de PostgreSQL : <https://www.postgresql.org/>
- [10] - Site d'Apache Camel : <https://camel.apache.org/>
- [11] - Site d'Apache Druid : <https://druid.apache.org/>
- [12] - Site de Redis : <https://redis.com/>
- [13] - Site de MinIO : <https://min.io/>
- [14] - Site de Elasticsearch : <https://www.elastic.co/fr/>
- [15] - Site d'Apache Superset : <https://superset.apache.org/>
- [16] - Site d'Apache Spark : <https://spark.apache.org/>
- [17] - Site d'Apache Kafka : <https://kafka.apache.org/>
- [18] - Site d'OpenWeatherMap : <https://openweathermap.org/>
- [19] - Site de Zookeeper : <https://zookeeper.apache.org/>
- [20] - Site de Express.js : <https://expressjs.com/fr/>
- [21] - Site présentant les caractéristiques du DHT22 :  
<https://components101.com/sensors/dht22-pinout-specs-datasheet#:~:text=The%20DHT22%20is%20a%20commonly,to%20interface%20with%20other%20microcontrollers>.
- [22] - Présentation de l'ESP32 : <https://fr.wikipedia.org/wiki/ESP32>
- [23] - Site de Vue.js : <https://vuejs.org/>
- [24] - Site de TypeScript : <https://www.typescriptlang.org/>
- [25] - Site de Swagger : <https://swagger.io/>
- [26] - Site de Jest : <https://jestjs.io/fr/>
- [27] - Présentation de l'AD8232 : <https://www.analog.com/media/en/technical-documentation/data-sheets/ad8232.pdf>
- [28] - Bibliothèque ArduinoJson : <https://www.arduino.cc/reference/en/libraries/arduinojson/>
- [29] - Bibliothèque HTTPClient : <https://www.arduino.cc/reference/en/libraries/httpclient/>
- [30] - Site de chart.js : <https://www.chartjs.org/>
- [31] - Site du plugin zoom de chart.js : <https://www.chartjs.org/chartjs-plugin-zoom/latest/>
- [32] - Site de FHIR : <https://fhir.org/>

## Annexe

Annexe 1 : Lien vers le GitLab du back du projet

<https://gitlab.ig.umons.ac.be/rami/umons-sensor-backend>

Annexe 2 : Code Arduino permettant l'envoi de données d'un DHT22 relié à un ESP32 à notre API

```
#include <DHT.h>
#include <WiFi.h>
#include <NTPClient.h>
#include <ArduinoJson.h>
#include <HTTPClient.h>
#include <time.h>

#define wifi_ssid "XXX" // to be completed
#define wifi_password "XXX" // to be completed

#define DHTPIN 4 // Broche digitale connectée au capteur DHT
#define DHTTYPE DHT22 // DHT 22 (AM2302), AM2321

const long gmtOffset_sec = 0;
const int daylightOffset_sec = 3600;

String DEVICE_ID = "DHT22_office";
String DEVICE_TYPE1 = "temperature";
String DEVICE_TYPE2 = "humidity";

long lastMsg = 0;
DHT dht(DHTPIN, DHTTYPE);
WiFiClient espClient;
const char* ntpServer = "pool.ntp.org";

void setup() {
    Serial.begin(115200);
    setup_wifi();
    dht.begin();
    configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
}

void setup_wifi() {
    delay(10);
    Serial.println();
    Serial.print("Connexion à ");
    Serial.println(wifi_ssid);
```

```

WiFi.begin(wifi_ssid, wifi_password);

while (WiFi.status() != WL_CONNECTED) {
    delay(500);
}
}

void httpPOST() {
    float humidity = dht.readHumidity();
    float temperature = dht.readTemperature();

    struct tm timeinfo;
    if(!getLocalTime(&timeinfo)){
        return;
    }

    struct tm *timestamp=&timeinfo;
    char buffer [25];
    strftime(buffer, 25, "%FT%T.000Z", timestamp);

    if (WiFi.status() == WL_CONNECTED) {
        WiFiClient client;
        HTTPClient http;

        http.begin("https://dev-api.rami.ig.umons.ac.be/api/v1/measurements");
        http.addHeader("Content-Type", "application/json");

        StaticJsonDocument<200> doc;
        doc["date"] = buffer;
        doc["value"] = humidity;
        doc["type"] = DEVICE_TYPE2;
        doc["sensor"] = DEVICE_ID;

        String requestBody;
        serializeJson(doc, requestBody);

        int httpResponseCode = http.POST(requestBody);

        Serial.print("HTTP Response code: ");
        Serial.println(httpResponseCode);

        http.end();

        http.begin("https://dev-api.rami.ig.umons.ac.be/api/v1/measurements");
        http.addHeader("Content-Type", "application/json");

        StaticJsonDocument<200> doc2;
        doc2["date"] = buffer;
    }
}

```

```

doc2["value"] = temperature;
doc2["type"] = DEVICE_TYPE1;
doc2["sensor"] = DEVICE_ID;

String requestBody2;
serializeJson(doc2, requestBody2);

int httpResponseCode2 = http.POST(requestBody2);

Serial.print("HTTP Response code: ");
Serial.println(httpResponseCode2);

http.end();
}

else {
    Serial.println("WiFi Disconnected");
}
}

void loop() {
    httpPOST();

    //delay : 1s between the send of two measurements

    delay(1000 * 1);
}

```

Annexe 3 : Lien vers le GitLab du front du projet

<https://gitlab.ig.umons.ac.be/rami/umons-sensor-frontend>