

Ewen GIRAUD-CARRIER
Enzo BOISSENIN
Pierre GAUDET



Compte rendu - projet IT

Système de surveillance de l'environnement

Entreprise ENPIWEN



Table des matières

I. Introduction	4
1. Contexte et objectifs du projet	4
II. Gestion du projet	5
1. Gestion du versionnement et collaboration avec GitHub	5
2. Environnement de développement unifié avec Docker	6
III. Description du système	7
1. Matériel utilisé	7
2. Choix des logiciels	8
3. Architecture du système	9
4. Fonctionnement des capteurs	11
Le module RFID RC522	11
Le module RFID RC522	11
Capteur KY-038 - Microphone	12
Protocole de communication Firmata	13
Comment fonctionne Firmata ?	13
Structure des messages Firmata	13
Principales commandes Firmata	14
Utilisation de PyFirmata	14
Fonctionnement de PyFirmata	14
La boucle de communication	14
Commandes de base avec PyFirmata	15
Communication à distance entre le Raspberry Pi et un PC	16
1. Accès distant via SSH	16
2. Transfert de fichiers avec SCP	17
3. Authentification par clés SSH	17
IV. Interface web	18
1. Fonctionnement du site	18
Objectifs du Site Web	18
Architecture de l'Application	18
Introduction et Structure Générale	18

2. Définition des routes Flask	19
Détails du fonctionnement	19
3. Structure et Navigation	20
Page de Connexion → Page d'Accueil	22
Page d'Accueil → Page Placer Badge	24
Placer Badge → Formulaire Badge ou Suppression Badge	26
Formulaire Badge → Succès ou Échec Enregistrement	27
Suppression Badge → Succès ou Échec Suppression	29
Page d'Accueil → Page Affichage Graphiques	30
Page d'Accueil → Page Badge et Utilisateur	35
4. Logos	38
V. Sécurité	39
1. Sécurisation des communications avec SSL/TLS	39
2. Système d'authentification et gestion des utilisateurs	40
Utilisation de Flask-Login	40
Chargement des utilisateurs	41
Connexion et déconnexion des utilisateurs	42
Contrôle d'accès basé sur les rôles	42
1. Diagramme fonctionnel	45
2. Structures des bases de données	48
3. Structure des Messages et Services ROS	50
Messages ROS	50
Services ROS	51
VII. Explication des codes	52
Code concernant les databases	52
Base de données pour l'humidité	52
Base de données pour la température	54
Base de données pour le volume sonore	56
Base de données pour les relevés de badge RFID	57
Base de données pour les infos de badge RFID	60
Code concernant la lecture de capteurs	64
Température et humidité	64
Lecture du badge RFID	65
Ecoute du micro	67
Code pour les serveurs de service ROS	69
Le service pour ajouter un badge	71
Le service pour supprimer un badge	73
Le service pour récupérer un identifiant de la database	75
VIII. Tests et résultats	77
Test dans le package sensors	77
Tests du noeud badge_sensor_node.py	77
Test du noeud DHT11	78
Test du noeud micro	79
Tests des nœuds du package database	81

Test de la base de données d'humidité	81
Test de la base de données sonore	81
Test de la base de données RFID	82
Test de la base de données de température	83
Tests des nœuds du package badge_rfid	84
Test du service d'ajout de badge	84
Test du service de suppression de badge	86
Test du service de navigation dans database pour le login	88
Tests des nœuds du package web_interface	90
Test de l'initialisation de ROS	90
Test du service d'ajout de badge	90
Test du service de suppression de badge	91
Test de la récupération des données en base	91
Test de l'API /data	92
Test de l'authentification utilisateur	93
Tests d'intégration	94
Test d'intégration de l'ajout de badge	94
Test d'intégration de la suppression de badge	95
Test d'intégration de l'interface Flask	96
IX. Conclusion et perspectives	98
X. Annexes	99
1. Documentations du matériel	99
A. Capteur DHT11	99
B. Capteur KY-038	99
C. Capteur RFID RC522	99
D. Raspberry PI 4B	99
E. Câbles utilisés	99
2. Documentations des logiciels et outils utilisés	99
A. Visual Studio Code	99
B. Docker	100
C. Sqlite3	100
D. ROS	100
E. Flask	100

I. Introduction

1. Contexte et objectifs du projet

L'objectif de ce projet est de développer un **système de supervision industrielle** basé sur un Raspberry Pi 4 fonctionnant sous Linux. Ce système est conçu pour collecter des données en temps réel à partir de capteurs industriels (température, son, humidité, etc.), les stocker et les traiter, puis les visualiser via une interface utilisateur interactive. Nous avons choisi de réaliser une interface web.

Ce projet intègre plusieurs aspects de l'informatique industrielle :

- **Interfaçage matériel** avec des capteurs via des protocoles comme *I2C*, *SPI* et *UART*.
- **Programmation système** sous Linux pour l'acquisition et le traitement des données.
- **Développement d'interfaces utilisateur** pour la visualisation et l'interaction avec le système.
- **Stockage des données** dans une base de données pour permettre une analyse historique et la détection d'anomalies.
- **Communication sécurisée** entre le Raspberry Pi et un PC distant, notamment via *SSH* et des protocoles de chiffrement.
- **Utilisation de ROS (Robot Operating System)** pour assurer une architecture logicielle modulaire et évolutive, facilitant l'intégration future de nouvelles fonctionnalités.

Ce projet vise ainsi à développer une solution fiable, évolutive et sécurisée pour la supervision industrielle, en exploitant les technologies modernes de l'informatique embarquée et des systèmes distribués.

II. Gestion du projet

Dans le cadre de notre projet, nous avons travaillé en équipe de trois afin de développer un système intégrant **ROS, SQL et HTML** en utilisant une **Raspberry Pi**. Pour assurer une gestion efficace et une progression fluide, nous avons divisé le travail en trois tâches principales :

- **Enzo** a pris en charge le développement de l'**interface web**, permettant l'affichage et l'interaction avec les données collectées.
- **Pierre** s'est occupé de la partie **physique du système et de la gestion de la base de données**, assurant ainsi le stockage et l'organisation des informations.
- **Ewen** a développé les différents **nœuds ROS**, nécessaires à la communication entre les capteurs et le reste du système.

Cette répartition claire des responsabilités nous a permis de travailler en parallèle tout en assurant une intégration fluide des différentes parties du projet.

Pour le partage des documentations sur les capteurs, des tutos, et de toute autre ressource, nous avons créé un dossier sur Google Drive à ce lien :

Lien vers le google drive du projet : [Cliquez ici](#)

1. Gestion du versionnement et collaboration avec GitHub

Afin de structurer notre développement, nous avons mis en place un **workflow Git organisé** autour de plusieurs branches :

- **Branches de développement par fonctionnalité** : chaque membre travaillait sur une branche dédiée (**feature/ros-nodes**, **feature/database**, **feature/web-interface**).
- **Branche de test et d'intégration** : nous fusionnions nos travaux sur la branche **develop**, où nous effectuions des tests avant validation.
- **Branche principale (main)** : une fois la branche **develop** jugée stable et fonctionnelle, nous la fusionnions avec **main**, garantissant ainsi un code propre et opérationnel à chaque étape clé.

Cette organisation a facilité le travail collaboratif en évitant les conflits majeurs et en maintenant une bonne visibilité sur l'évolution du projet.

Lien vers le repository Github : [Cliquez ici](#)

2. Environnement de développement unifié avec Docker

ROS Noetic est principalement conçu pour fonctionner avec Ubuntu 20.04 (Focal Fossa) et Debian Buster (10). Cependant, les versions plus récentes de Raspberry Pi OS, telles que Bullseye (11), ne sont pas officiellement supportées pour ROS Noetic. De plus, pour éviter les problèmes de compatibilité entre nos différentes machines et assurer une cohérence dans le développement, nous avons utilisé **Docker**. Nous avons installé une **image ROS Noetic** sur la Raspberry Pi et avons utilisé le même conteneur sur nos PC personnels. Cette approche a présenté plusieurs avantages :

- **Développement sous ROS** sans avoir beaucoup d’installation à faire et éviter les problèmes d’incompatibilité entre Bullseye et ROS Noetic.
- **Uniformisation** des versions et des dépendances, évitant les incompatibilités.
- **Développement et test** dans un environnement identique à celui de la production.
- **Déploiement simplifié** sur la Raspberry Pi sans nécessiter d’installation complexe sur chaque poste.

Cette configuration nous a permis de développer depuis chez nous et chacun de notre côté, sans avoir besoin d’être tous les trois sur la Raspberry Pi en même temps.

Afin de simplifier le déploiement sur n’importe quelle machine, nous avons écrit un fichier Dockerfile ainsi qu’un fichier docker-compose.yml pour faciliter le déploiement. Toutes les commandes à utiliser pour le déploiement sont écrites dans le fichier README.md à la racine du repository Github.

III. Description du système

1. Matériel utilisé

Pour notre projet, nous utilisons diverses cartes et composants électroniques fournis par notre institution :

- Une carte **Raspberry Pi 4B**, cœur du système
- Une carte **Arduino UNO R3**, utilisée pour recevoir les données du micro
- Un capteur permettant de lire et d'écrire sur des badges **RFID RC522**
- Un capteur permettant de relever la température ainsi que l'humidité de l'environnement : **DHT11**
- Un capteur **KY-038**, permettant d'agir comme un micro et de mesurer le niveau sonore.
- Une multitude de **câbles mâle-mâle et mâle-femelle**, retrouvable dans la majorité des kits disponibles à la vente.
- Un **câble USB A vers B**, pour relier l'Arduino à la carte Raspberry.
- Une **résistance de 2K Ohm**.

2. Choix des logiciels

Pour le développement des différents programmes, nous avons utilisé l'IDE Visual Studio Code (lien de téléchargement disponible en Annexe).

Afin de pouvoir créer / modifier les codes directement sur la Raspberry, nous avons utilisé la fonctionnalité SSH Remote sur cet IDE. Cela permet de se connecter en SSH à la carte, et de modifier localement les fichiers ou autre.

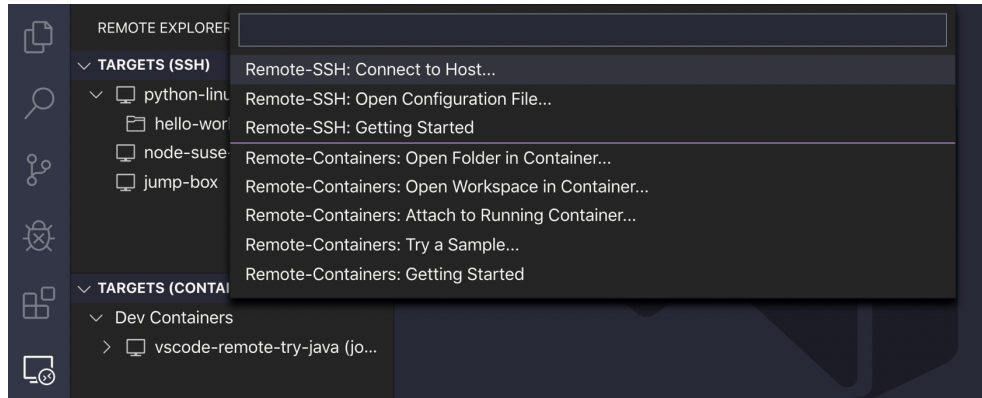


FIGURE 1 – Fenêtre de connexion SSH sur Visual Studio Code

3. Architecture du système

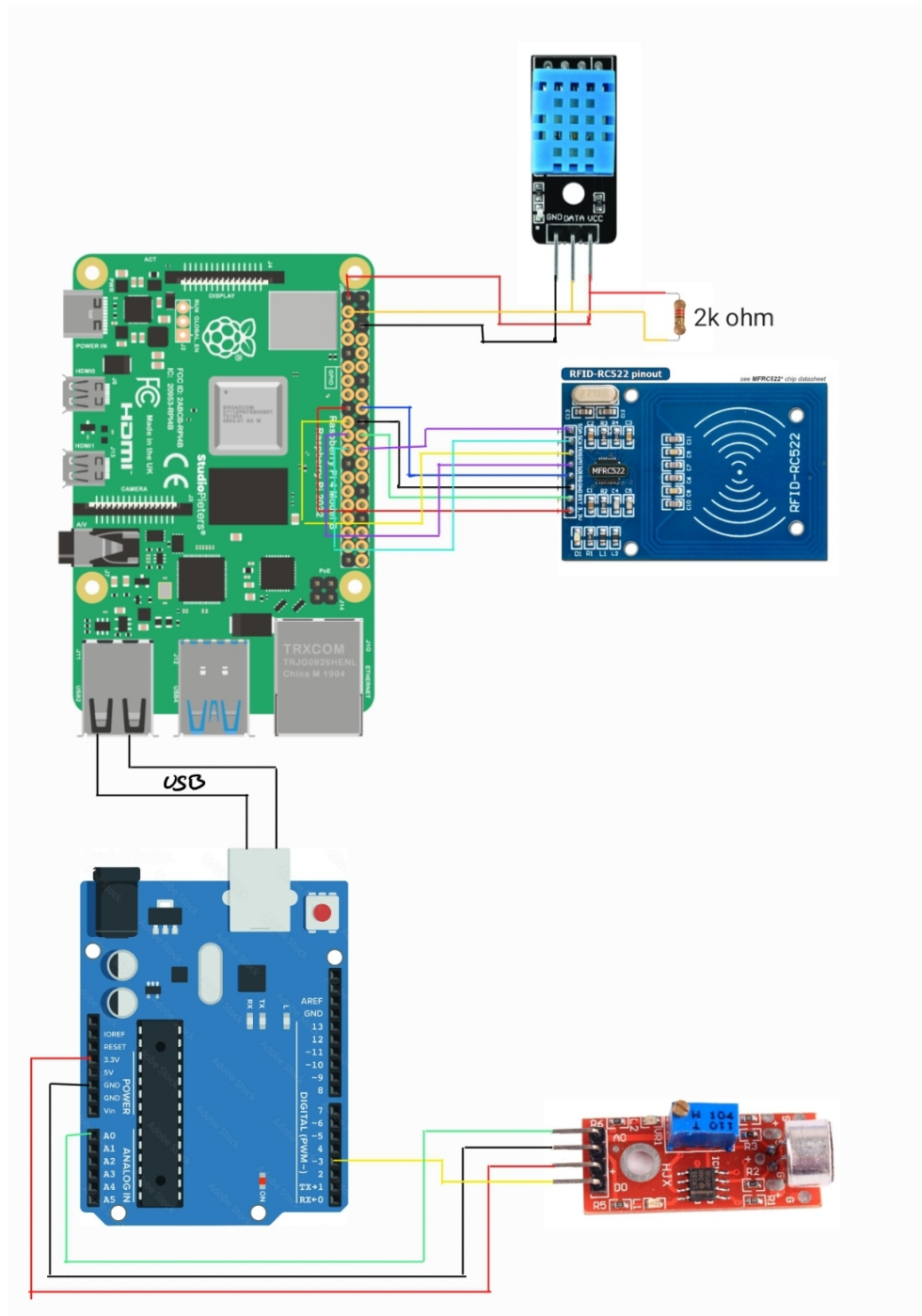


FIGURE 2 – Branchements du système

En ce qui concerne les branchements physiques de notre projet, nous utilisons des câbles basiques fournis dans les différents kits Arduino et Raspberry achetables sur le marché. Les liaisons sont effectuées ainsi :

Pour le capteur DHT11 (température et humidité) :

- La broche GND est reliée à un pin GND de la Raspberry, le choix de celui-ci n'a pas d'importance.
- La broche DATA du capteur est reliée au GPIO2 de la carte
- La broche VCC du capteur est reliée au 3.3V de la carte
- Une résistance de 2K ohm relie la borne DATA et VCC

Pour le capteur KY-038 (microphone) :

Pour récupérer les données de ce capteur, il nous faut utiliser une carte Arduino, car notre modèle de carte Raspberry n'est pas en mesure de les traiter correctement.

- La broche A0 du capteur est reliée au pin A0 de la carte Arduino
- La broche GND du capteur est reliée au pin DIGITAL 3 de la carte Arduino.
- La broche GND du capteur est reliée à un pin GND de la carte, peu importe lequel.
- La broche VCC du capteur est reliée à la broche 3.3V de l'Arduino.

La carte Arduino est elle-même reliée à la Raspberry via un câble USB B vers USB A, de préférence branché sur le port usb0 (celui en haut à gauche si l'on met les ports USB de la Raspberry face à nous).

Pour le capteur RFID RC522 (lecteur de badge) :

- La broche SDA du capteur est reliée au GPIO8 de la Raspberry
- La broche SCK est reliée au GPIO11
- La broche MOSI est reliée au GPIO10
- La broche MISO est reliée au GPIO9
- La broche IRQ est reliée au GPIO24
- La broche GND est reliée à une broche GND de la carte, peu importe laquelle.
- La broche RST est reliée au GPIO25
- La broche 3.3V est, comme son nom l'indique, reliée à la borne 3.3V de la Raspberry, située entre le GPIO22 et GPIO10

4. Fonctionnement des capteurs

Le module RFID RC522

Le module **RFID RC522** est un lecteur de badges RFID fonctionnant à une fréquence de 13.56 MHz. Il est couramment utilisé pour des applications de contrôle d'accès et d'identification. Ce module est conçu pour interagir avec des cartes RFID passives de type MIFARE, qui ne possèdent pas de batterie et sont alimentées par le champ électromagnétique généré par le lecteur.

Principe de fonctionnement

Le module RFID RC522 fonctionne selon le principe de la *radio-identification* :

1. Le module génère un champ électromagnétique à 13.56 MHz.
2. Lorsqu'un badge RFID passif entre dans ce champ, il capte l'énergie émise et s'active.
3. Le badge renvoie alors son identifiant unique (*UID*) en modulant le signal reçu.
4. Le module RC522 décode ces informations et les transmet à un microcontrôleur ou un ordinateur via l'interface *SPI* (*Serial Peripheral Interface*).

Connexion avec une Raspberry Pi

Le module RC522 communique avec une Raspberry Pi via l'interface SPI, nécessitant les connexions suivantes :

- **VCC** : alimentation en 3.3V (ne pas utiliser 5V, car le module n'est pas tolérant en tension).
- **GND** : masse.
- **SCK, MOSI, MISO, NSS (SDA)** : lignes de communication SPI.
- **RST** : broche de réinitialisation du module.

Capteur de température et d'humidité DHT11

Le **DHT11** est un capteur numérique capable de mesurer la température et l'humidité relative de l'air. Il est largement utilisé dans les projets de domotique et d'automatisation en raison de sa simplicité et de son faible coût. Il communique avec un microcontrôleur ou un ordinateur via une interface numérique en utilisant un protocole de communication propriétaire.

Principe de fonctionnement

Le capteur DHT11 repose sur deux éléments principaux :

- **Un capteur d'humidité capacitif**, qui mesure l'humidité relative de l'air en analysant la variation de capacité d'un polymère absorbant l'eau.
- **Un thermistor à coefficient de température négatif (NTC)**, qui permet de mesurer la température en fonction de la résistance électrique du composant.

Le DHT11 échantillonne ces mesures et transmet les données sous forme de signaux numériques.

Caractéristiques techniques

Les principales caractéristiques du capteur sont les suivantes :

- **Plage de température** : 0 à 50 degrés ± 2 degrés de précision).
- **Plage d'humidité** : 20 % à 90 % RH $\pm 5\%$ de précision).
- **Tension d'alimentation** : 3.3 V à 5 V.
- **Fréquence d'échantillonnage** : une mesure toutes les 2s.

Capteur KY-038 - Microphone

Principe de fonctionnement

Le **KY-038** est un module microphone conçu pour détecter des variations sonores. Il repose sur :

- **Un microphone électret**, qui capte les ondes sonores et les convertit en un signal électrique de faible amplitude.
- **Un amplificateur intégré**, qui augmente le signal pour le rendre exploitable.
- **Un comparateur LM393**, qui génère un signal numérique indiquant si l'intensité sonore dépasse un seuil défini par un potentiomètre.

Le module dispose de deux sorties distinctes :

- **Sortie analogique (A0)** : fournit une tension proportionnelle à l'intensité du son capté.
- **Sortie numérique (D0)** : passe à l'état haut (1) lorsque le niveau sonore dépasse le seuil défini par l'utilisateur.

Pourquoi ne pas connecter directement le KY-038 à la Raspberry Pi ?

La Raspberry Pi ne possède **pas de convertisseur analogique-numérique (CAN)** intégré, ce qui l'empêche de lire directement la sortie analogique (A0) du capteur KY-038.

Différence entre les deux sorties :

- **La sortie A0** produit une tension continue variable, correspondant à l'amplitude du son. Ce signal étant **analogique**, la Raspberry Pi ne peut pas le traiter sans ajouter un convertisseur externe comme le MCP3008.
- **La sortie D0**, en revanche, est un simple signal binaire (0 ou 1) généré par un comparateur. Cette sortie peut être lue directement par un GPIO de la Raspberry Pi sans conversion.

Solutions pour utiliser la sortie A0 avec une Raspberry Pi :

- Ajouter un **convertisseur analogique-numérique (ADC)** comme le MCP3008, qui transforme le signal analogique en numérique avant de l'envoyer à la Raspberry Pi.
- Utiliser un **microphone USB**, qui intègre déjà un CAN et envoie directement des données numériques exploitables.

Ainsi, la sortie numérique (D0) peut être utilisée directement sur la Raspberry Pi, tandis que la sortie analogique (A0) nécessite un module ADC ou un microcontrôleur intermédiaire comme l'Arduino.

Protocole de communication Firmata

Firmata est un protocole de communication standardisé permettant de piloter un microcontrôleur, comme une carte Arduino, à partir d'un ordinateur ou d'un autre appareil. Ce protocole transforme une carte Arduino en une *interface universelle*, capable d'interagir avec n'importe quel langage ou système via une liaison série (USB, Bluetooth, etc.).

Comment fonctionne Firmata ?

1. Le programme Firmata est téléversé sur la carte Arduino.
2. L'ordinateur envoie des **commandes codées** selon le protocole Firmata via le port série.
3. Le firmware Firmata, installé sur la carte Arduino, **décodes les commandes** et exécute les actions demandées (allumer une LED, lire un capteur, etc.).
4. Si nécessaire, l'Arduino **renvoie des données** vers l'ordinateur, également sous forme de messages Firmata.

Structure des messages Firmata

Les messages Firmata sont constitués de plusieurs octets :

- **Octets de commande** : Spécifient l'action à réaliser (par exemple, allumer une LED ou lire une broche).
- **Octets de données** : Contiennent les informations nécessaires à l'exécution de la commande.

Exemple de message

- **Commande numérique** (DIGITAL_MESSAGE) :
 - Commande : 0x90
 - Données : Numéro de la broche + état (HIGH ou LOW)

Principales commandes Firmata

Commande	Octet (hex)	Description
DIGITAL_MESSAGE	0x90	Écriture sur une broche numérique
ANALOG_MESSAGE	0xE0	Lecture/écriture sur une broche analogique
REPORT_ANALOG	0xC0	Active/désactive le rapport d'une broche analogique
SET_PIN_MODE	0xF4	Configure le mode d'une broche (entrée, sortie, etc.)
SYSEX_START	0xF0	Début d'un message système étendu
SYSEX_END	0xF7	Fin d'un message système étendu

Utilisation de PyFirmata

Fonctionnement de PyFirmata

PyFirmata est une bibliothèque Python qui simplifie l'utilisation du protocole Firmata. Elle permet de contrôler une carte Arduino directement depuis un script Python sans avoir à écrire de code embarqué.

Étapes principales

1. PyFirmata établit une liaison série avec la carte Arduino.
2. Les commandes Python sont converties en messages Firmata.
3. L'Arduino exécute les actions correspondantes et renvoie des données si nécessaire.

La boucle de communication

Pour des lectures continues (par exemple sur un capteur analogique), PyFirmata utilise une **boucle de communication** avec un *Iterator*.

Principe

1. L'ordinateur active le mode "rapport continu" pour une broche.
2. L'Arduino envoie périodiquement les données.
3. PyFirmata écoute ces données via une boucle et les rend accessibles.

Exemple de boucle de communication

```
1 from pyfirmata import Arduino, util
2 import time
3
4 board = Arduino('/dev/ttyUSB0')
5
6 # Initialisation de l'iterateur
7 it = util.Iterator(board)
8 it.start()
9
10 # Activer le rapport continu sur A0
11 board.analog[0].enable_reporting()
12
13 # Lire les valeurs en boucle
14 for _ in range(10):
15     value = board.analog[0].read()
16     print(f"Valeur analogique : {value}")
17     time.sleep(0.5)
18
19 board.exit()
```

Commandes de base avec PyFirmata

Commande	Exemple	Description
Connexion	<code>Arduino('/dev/ttyUSB0')</code>	Initialise la connexion avec la carte
Définir une broche	<code>board.get_pin('d:13:o')</code>	Configure la broche 13 en sortie
Écrire numérique	<code>board.digital[13].write(1)</code>	Allume une LED sur la broche 13
Lire numérique	<code>board.digital[13].read()</code>	Renvoie l'état (1 ou 0) de la broche
Lire analogique	<code>board.analog[0].read()</code>	Renvoie une valeur entre 0 et 1 sur A0
Écrire PWM	<code>board.digital[3].write(0.5)</code>	Envoie une valeur PWM (0.0 à 1.0) sur D3
Activer le rapport continu	<code>board.analog[0].enable_reporting()</code>	Active le rapport périodique sur A0

Syntaxe des commandes avec `get_pin`

Les commandes PyFirmata utilisent une syntaxe concise pour configurer et utiliser les broches :

— **Syntaxe** : `'type:pin:mode'`

— **Paramètres** :

— **type** :

— `'d'` : broche numérique.

— `'a'` : broche analogique.

— **pin** : Numéro de la broche.

— **mode** :

— `'i'` : entrée.

— `'o'` : sortie.

— `'p'` : PWM (modulation de largeur d'impulsion).

— `'s'` : servo.

Communication à distance entre le Raspberry Pi et un PC

La communication à distance entre un Raspberry Pi et un PC est essentielle pour le développement, la maintenance et le déploiement des applications embarquées. Cette section présente différentes méthodes pour établir une connexion sécurisée entre un PC et un Raspberry Pi, en utilisant SSH, SCP et l'authentification par clés SSH que nous avons du suivre pour réaliser ce projet.

1. Accès distant via SSH

Le protocole SSH (Secure Shell) permet d'accéder à distance au terminal d'un Raspberry Pi de manière sécurisée.

Activation du serveur SSH

Par défaut, le serveur SSH n'est pas toujours activé sur un Raspberry Pi. Pour l'activer, utilisez la commande suivante :

```
1 $ sudo raspi-config
```

Naviguez ensuite dans :

1. Interfacing Options -> SSH
2. Sélectionnez **Yes**
3. Quittez en enregistrant les modifications

On peut vérifier si le SSH fonctionne avec la commande :

```
1 $ sudo systemctl status ssh
```

Connexion SSH depuis un PC

Pour se connecter au Raspberry Pi depuis un PC sous Linux, macOS ou Windows, on tape la commande :

```
1 $ ssh pi@<ip de la Raspberry>
```

Lors de la première connexion, acceptez la clé en tapant **yes**, puis on entre le mot de passe.

2. Transfert de fichiers avec SCP

Le protocole SCP (Secure Copy Protocol) permet de copier des fichiers entre le Raspberry Pi et un PC via SSH.

Copie d'un fichier vers le Raspberry Pi

```
1 $ scp fichier.txt pi@<ip de la Raspberry>:/home/pi/
```

Copie d'un fichier depuis le Raspberry Pi

```
1 $ scp pi@<ip de la Raspberry>:/home/pi/fichier.txt /chemin/vers/dossier/local/
```

Copie d'un dossier entier

```
1 $ scp -r dossier/ pi@<ip de la Raspberry>:/home/pi/
```

3. Authentification par clés SSH

L'authentification par clés SSH permet de sécuriser la connexion sans saisir le mot de passe à chaque fois.

Génération d'une paire de clés SSH sur le PC

On exécute la commande suivante sur notre PC :

```
1 $ ssh-keygen -t rsa -b 4096 -C "votre_email@example.com"
```

On appuie sur **Entrée** pour accepter l'emplacement par défaut.

Copie de la clé publique sur le Raspberry Pi

```
1 $ ssh-copy-id pi@<ip de la Raspberry>
```

Désormais, la connexion SSH ne demandera plus de mot de passe :

```
1 $ ssh pi@<ip de la Raspberry>
```

Conclusion L'accès distant et sécurisé au Raspberry Pi est crucial pour la gestion de notre projet. SSH permet un contrôle total, SCP et SFTP facilitent le transfert de fichiers, et l'authentification par clés SSH améliore la sécurité tout en supprimant la nécessité d'un mot de passe, facilitant la connexion.

IV. Interface web

1. Fonctionnement du site

L'interface web développée permet de superviser l'état des capteurs et de gérer l'accès des utilisateurs via des badges RFID. Elle repose sur l'API Flask (interface de programmation d'application) et est intégrée avec ROS afin d'assurer la communication avec les capteurs et les services de gestion des badges. Cette plateforme est conçue pour offrir une navigation intuitive et un accès sécurisé aux différentes fonctionnalités essentielles du système de supervision.

1.1. Objectifs du Site Web

Nous avons conçu cette interface pour :

- Permettre une **visualisation en temps réel** des capteurs (température, humidité, volume sonore).
- Gérer l'**authentification des utilisateurs**, avec un contrôle des accès.
- Offrir une **interaction avec les badges RFID**, pour ajouter ou supprimer des utilisateurs via ROS.

1.2. Architecture de l'Application

L'application suit un modèle MVC (**Modèle-Vue-Contrôleur**) :

- **Le modèle** gère les utilisateurs et les données des capteurs.
- **La vue** affiche les informations sous forme de graphiques interactifs.
- **Le contrôleur** assure la liaison entre l'utilisateur et le système, en interagissant avec ROS et la base de données.

1.3. Introduction et Structure Générale

Le fichier `app.py` est le cœur de l'application Flask permettant la gestion des capteurs et des badges RFID. Il est structuré en plusieurs sections :

- **Authentification des utilisateurs** (avec Flask-Login et un service ROS)
- **Gestion des badges RFID** (ajout et suppression via des services ROS)
- **Acquisition des données des capteurs** (via SQLite)
- **Définition des routes Flask** (pages HTML et API pour les données)
- **Lancement de l'application Flask avec ROS**

2. Définition des routes Flask

Le fichier définit plusieurs routes pour gérer l'interface web. Par exemple, la route d'accueil :

```
1 @app.route('/')
2 def login():
3     return render_template('login.html')
```

Ou encore celle permettant d'afficher les graphiques des capteurs :

```
1 @app.route('/graph_capteurs')
2 @login_required
3 def graph_capteurs():
4     return render_template('graph_capteurs.html')
```

2.1 Lancement de l'Application

Le code suivant constitue le point d'entrée de l'application :

```
1 if __name__ == '__main__':
2     init_ros()
3     app.run(host="0.0.0.0", port=5000, ssl_context=(os.path.join(server_path, '
    server.crt'), os.path.join(server_path, 'server.key')))
```

Ce code sert à initialiser le système ROS et à démarrer le serveur Flask.

2.1.1. Détails du fonctionnement

Initialisation de ROS

L'instruction :

```
1 init_ros()
```

sert à initialiser le système ROS, pour configurer les nœuds, les services et les abonnements nécessaires à la communication avec les capteurs et le système RFID.

Lancement du serveur Flask

Le serveur Flask est lancé avec l'instruction :

```
1 if __name__ == '__main__':
2     init_ros()
3     app.run(host="0.0.0.0", port=5000, ssl_context=(os.path.join(server_path, '
    server.crt'), os.path.join(server_path, 'server.key')))
```

Voici comment cela fonctionne :

- `host="0.0.0.0"` : permet à l'application d'être accessible depuis n'importe quelle adresse IP sur le réseau.
- `port=5000` : définit le port sur lequel l'application tourne.
- `ssl_context=(...)` : active le protocole **HTTPS** en fournissant les chemins des fichiers de certificat et de clé SSL stockés dans `server_path`.

3. Structure et Navigation

Notre interface web est composée de plusieurs pages, chacune ayant un rôle spécifique. La navigation entre ces pages se fait autour d'une page d'accueil centrale (et principale) accessible après authentification de l'utilisateur. Vous pouvez observer ci-dessous un organigramme qui explique le fonctionnement complet des redirections des différentes pages de notre site. Pour bien comprendre chaque partie de cet organigramme, nous allons les détailler une à une. Avant de vouloir commencer à coder, il fallait qu'on réfléchisse au fonctionnement des différentes pages que nous voulions afficher. Un des principaux buts de ce projet est de pouvoir se servir des données fournies par les différents capteurs. C'est d'abord le système de badge RFID qui nous a intéressé : nous nous sommes dit que nous pourrions créer une interface qui permet à des employés d'une entreprise de s'identifier leur badge, ce qui leur permettra ensuite de faire le pointage quotidien. Nous avons, pour commencer, créé une page d'accueil avec deux boutons : 'ajouter un badge' et 'supprimer un badge'.

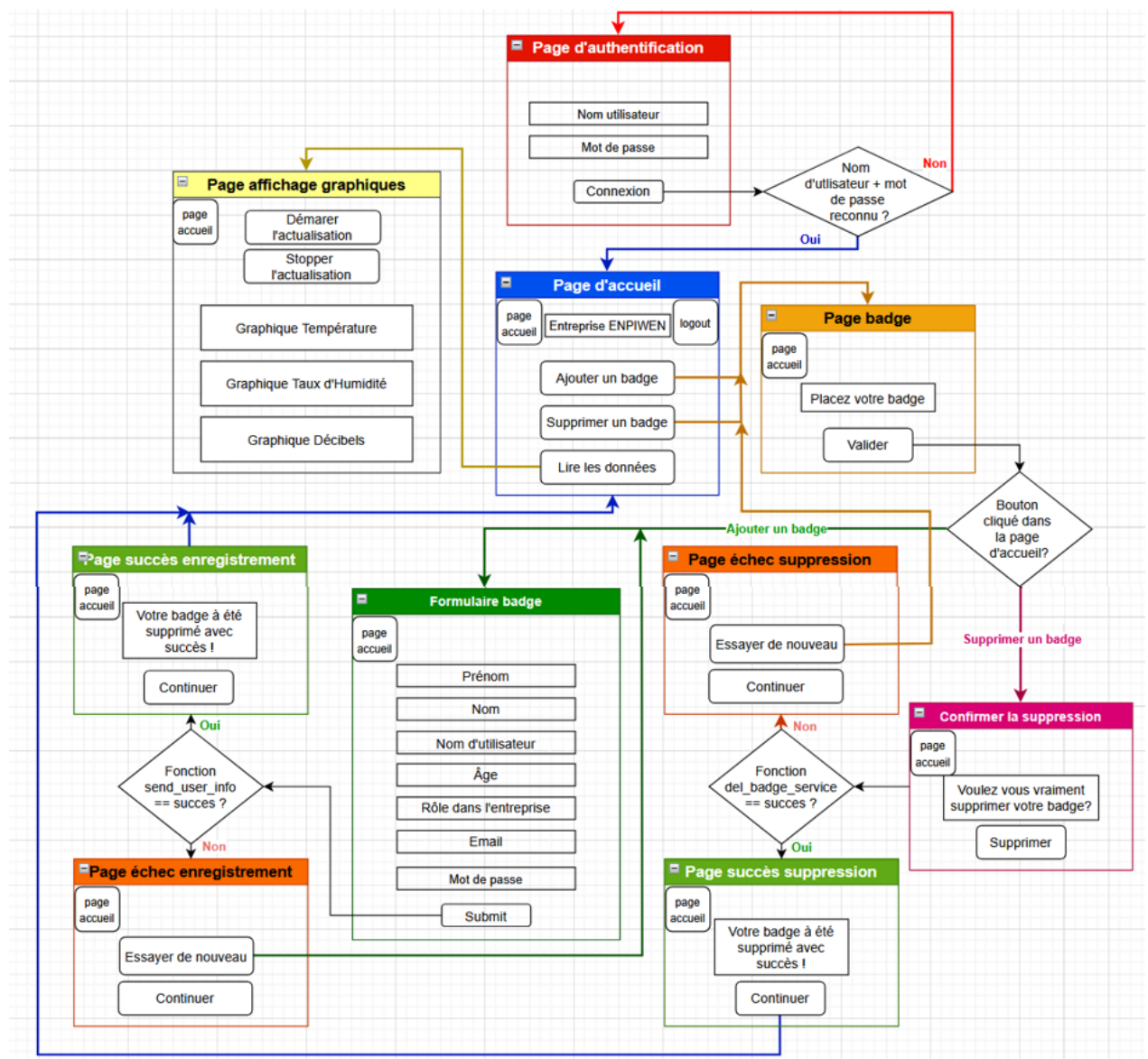


FIGURE 3 – Schéma général de l'interface web

3.1. Page de Connexion → Page d'Accueil

Lorsqu'un utilisateur tente de se connecter, il saisit ses identifiants (nom d'utilisateur et mot de passe). Une fois validés par le service ROS : - Si l'authentification réussit, l'utilisateur est redirigé vers la page d'accueil. - Si l'authentification échoue, un message d'erreur s'affiche et la connexion est refusée.

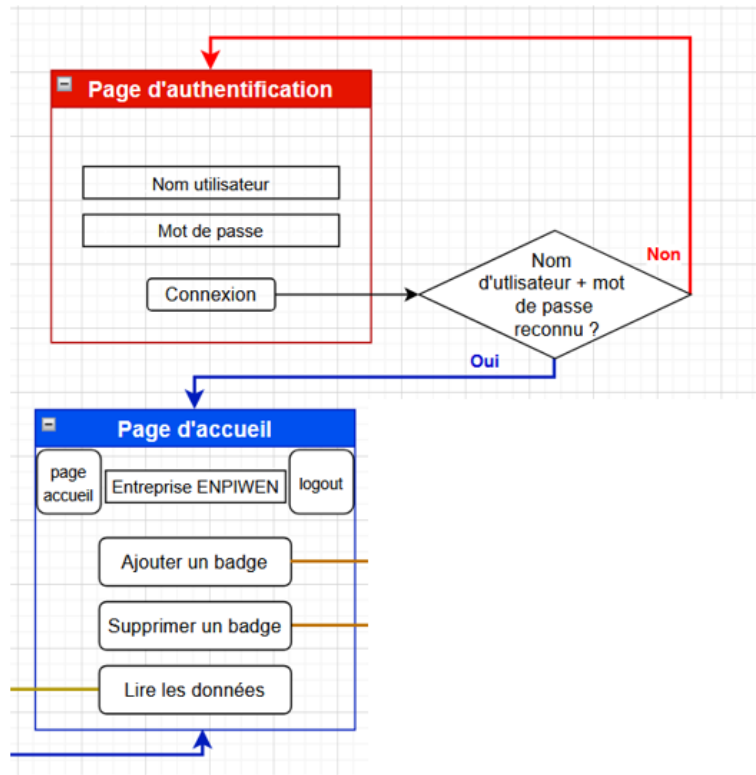


FIGURE 4 – Flux de connexion

Fonctionnement de la page '/login'

Voici comment se construit le code dans app.py :

```
1 @app.route('/', methods=['GET', 'POST'])
2 def login():
3     if request.method == 'POST':
4         username = request.form['username']
5         password = request.form['password']
6         user = authenticate(username, password)
7
8         if user:
9             login_user(user)
10            users_cache[user.id] = user # Stocke l'utilisateur pour viter d'
11                                         appeler ROS a chaque requete
12
13            print(f"Utilisateur connect ? {current_user.is_authenticated}")
14
15            return redirect(url_for('page_accueil'))
16        else:
17            flash('Nom d\'utilisateur ou mot de passe incorrect', 'danger')
18
19    return render_template('login.html')
```

On demande à l'utilisateur de rentrer son nom d'utilisateur et son mot de passe. Si les données correspondent bien à celles de la base de données, alors on redirige l'utilisateur sur la page d'accueil. Sinon, on lui indique que quelque chose ne va pas et on l'invite à recommencer.

Fonctionnement de la page '/page_accueil'

Avant de vouloir commencer à coder, il fallait qu'on réfléchisse au fonctionnement des différentes pages que nous voulions afficher. Un des principaux buts de ce projet est de pouvoir se servir des données fournies par les différents capteurs. C'est d'abord le système de badge RFID qui nous a intéressé : nous nous sommes dit que nous pourrions créer une interface qui permet à des employés d'une entreprise d'identifier leur badge, ce qui leur permettra ensuite de faire le pointage quotidien. Nous avons, pour commencer, créé une page d'accueil avec deux boutons : 'ajouter un badge' et 'supprimer un badge' (cf 'Gestion de la redirection'). Nous avons ensuite décidé de placer un bouton 'lire les données' qui nous permettrait de visualiser sous forme de graphe les données provenant des autres capteurs (cf Page 'graph_capteur'). Pour finir, nous avons placé un bouton 'Badges et utilisateurs' afin que l'utilisateur puisse vérifier que son badge a bien été enregistré et de voir toutes les personnes identifiées dans la base de données.

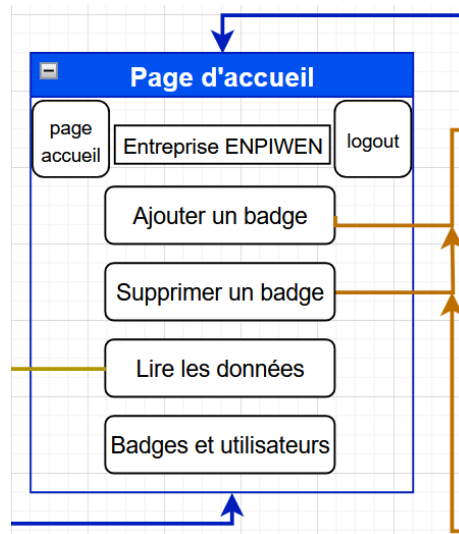


FIGURE 5 – Affichage de la page d'accueil

3.2. Page d'Accueil → Page Placer Badge

Depuis l'accueil, un utilisateur ayant les permissions d'administrateur peut accéder à la gestion des badges RFID. - En cliquant sur le bouton correspondant, il est dirigé vers la page "Placer Badge", où il peut choisir entre ajouter un badge ou en supprimer un.

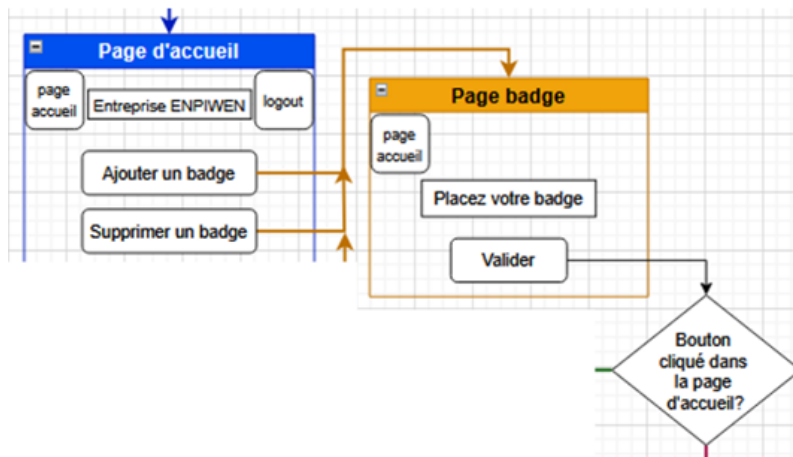


FIGURE 6 – Gestion des badges RFID

Gestion de la redirection

Que nous voulions ajouter ou supprimer un badge, il faut dans les deux cas que l'utilisateur place son badge devant le capteur afin que celui-ci soit reconnu. Nous nous sommes demandé comment faire pour n'utiliser qu'une seule page html qui nous permette ensuite de rediriger l'utilisateur vers une page qui correspond au bouton qu'il avait sélectionné initialement ?

Ci-dessous, vous pouvez voir le code permettant l'ajout des boutons 'ajouter badge' et 'supprimer badge'. Dans la balise form, nous avons ajouté une action qui permet de rediriger l'utilisateur vers la page '/placez_badge'. Et pour chaque bouton, on ajoute un champ caché <input type="hidden" name="action" value="ajouter" (ou "supprimer")>.

```
1 <p>Si vous tes nouveau dans l'entreprise, rentrez un nouveau badge</p>
2 <form method="GET" action="{ url_for('placez_badge') }">
3   <input type="hidden" name="action" value="ajouter">
4   <button type="submit">Ajouter un badge</button>
5   <br>
6 </form>
7 <p>Si vous quittez l'entreprise, supprimer votre badge</p>
8 <form method="GET" action="{ url_for('placez_badge') }">
9   <input type="hidden" name="action" value="supprimer">
10  <button type="submit">Supprimer un badge</button>
11  <br>
12 </form>
```

Puis partie se trouvant dans app.py (voir ci-dessous), on récupère soit la valeur "ajouter" et on redirige vers la page '/formulaire_badge', soit la valeur "supprimer" et on redirige vers la page '/conf_supression'

```
1 @app.route('/placez_badge', methods=['GET'])
2 @login_required
3 @admin_required # Vérifie que l'utilisateur est admin
4 def placez_badge():
5     """ Page intermédiaire pour ajouter ou supprimer un badge """
6     if request.method == 'GET':
7         action = request.form.get('action')
8         if action == "ajouter":
9             return redirect(url_for('formulaire_badge'))
10        elif action == "supprimer":
11            return redirect(url_for('conf_supression'))
12
13        action = request.args.get('action')
14        if not action:
15            return "Erreur : aucune action spécifiée", 400
16
17        return render_template('placez_badge.html', action=action)
```

3.3. Placer Badge → Formulaire Badge ou Suppression Badge

Nous venons de voir le principe de fonctionnement des redirections des boutons 'ajouter_badge' (renvoie à /formulaire badge) et 'supprimer_badge' (renvoie à /conf_supression) depuis la page d'accueil. Nous allons voir maintenant comment ces pages fonctionnent :

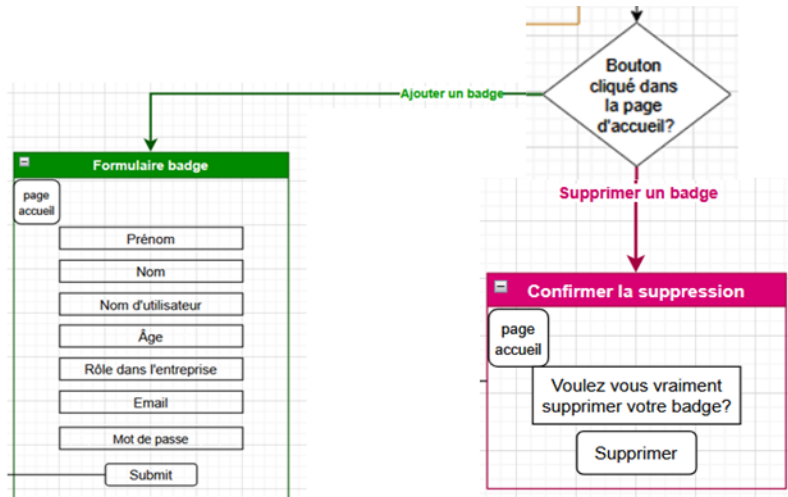
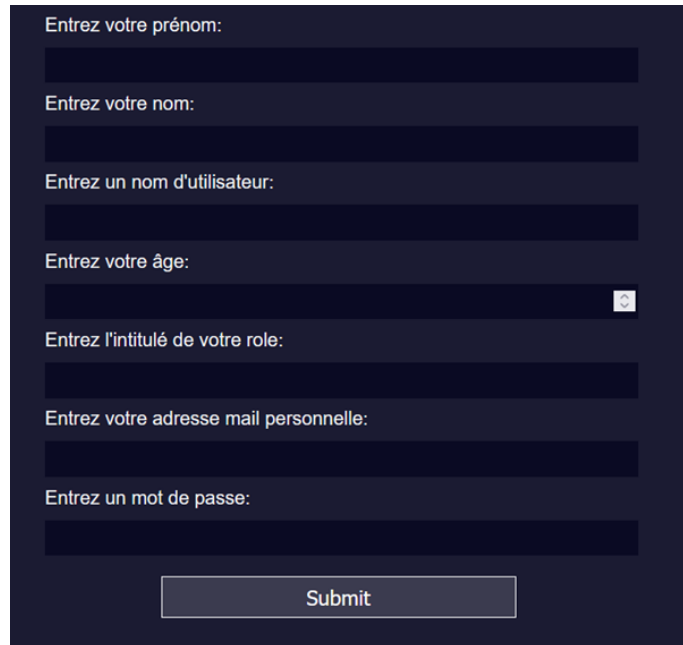


FIGURE 7 – Options de gestion des badges

3.4. Formulaire Badge → Succès ou Échec Enregistrement

Nous demandons à l'utilisateur de rentrer ses coordonnées :



Entrez votre prénom:

Entrez votre nom:

Entrez un nom d'utilisateur:

Entrez votre âge:

Entrez l'intitulé de votre rôle:

Entrez votre adresse mail personnelle:

Entrez un mot de passe:

Submit

FIGURE 8 – Formulaire utilisateur

Vous pouvez voir ci-dessous la construction du formulaire :

```
1 <fieldset>
2   <label for="prenom">Entrez votre prenom: <input id="prenom" name="prenom"
3     type="text" required /></label>
4   <label for="nom">Entrez votre nom: <input id="nom" name="nom" type="text"
5     required /></label>
6   <label for="nom">Entrez un nom d utilisateur: <input id="username" name="
7     username" type="text" required /></label>
8   <label for="age">Entrez votre age: <input id="age" type="number" name="age
9     " min="13" max="120" /></label>
10  <label for="intitule_du_poste">Entrez l intitulé de votre rôle: <input id=
    "job_title" name="job_title" type="text" required /></label>
    <label for="email">Entrez votre adresse mail personnelle: <input id="email
      " name="email" type="email" required /></label>
    <label for="mdp">Entrez un mot de passe: <input id="mdp" name="mdp" type="
      password" pattern="[a-z0-5]{8,}" required /></label>
    <input type="submit" value="Submit" />
  </fieldset>
```

Nous avons plusieurs champs à remplir et chacun d'entre eux a un type spécifique. Chaque type a ses spécificités : email requiert un @ dans la chaîne de caractère, text requiert un texte etc... Et nous avons ajouté des limites dans le cas de l'âge : en prenant une fourchette large, l'âge d'une personne est définie entre 13 et 120 ans. De même pour le mot de passe, on demande à l'utilisateur d'entrer au minimum 8 caractères compris entre a et z et entre 0 et 5. Pour finir, nous ajoutons l'attribut 'required' à chaque balise 'input', ce qui permet à la page de vérifier si tous les champs ont été rentrés après avoir appuyé sur le bouton 'submit'.

Lorsqu'un utilisateur soumet le formulaire d'ajout d'un badge : - Si l'ajout est réussi, il est dirigé vers la page "Succès Enregistrement". Il peut ensuite revenir sur la page d'accueil. - Si une erreur se produit (par exemple, si les informations sont incorrectes ou si le service échoue), il est redirigé vers la page "Échec Enregistrement" et a la possibilité de revenir à la page d'accueil ou bien de corriger ses informations sur la page formulaire.

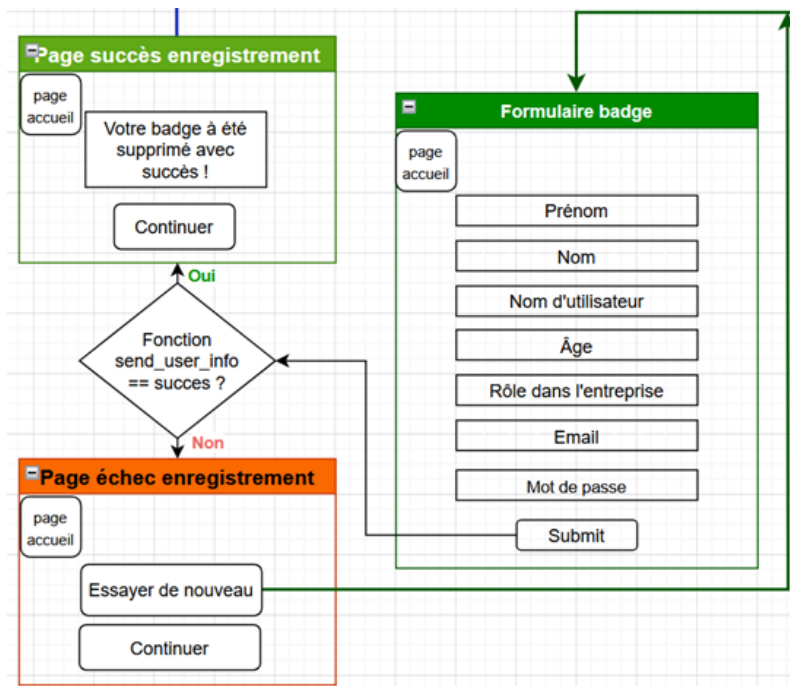


FIGURE 9 – Succès ou échec enregistrement d'un badge

3.5. Suppression Badge → Succès ou Échec Suppression

Lorsqu'un utilisateur demande la suppression d'un badge : - Si la suppression est validée avec succès, il est redirigé vers la page "Succès Suppression". - Si une erreur survient (par exemple, si le service n'est pas disponible), il est redirigé vers la page "Échec Suppression".

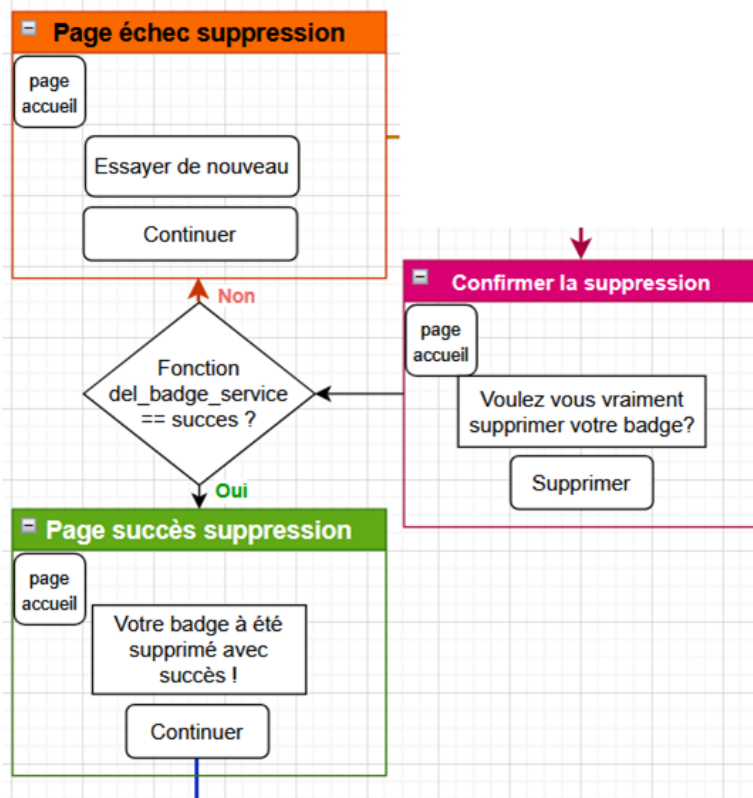


FIGURE 10 – Suppression d'un badge

3.6. Page d'Accueil → Page Affichage Graphiques

Depuis la page d'accueil, l'utilisateur peut accéder à la page affichant les graphiques des capteurs.

- En cliquant sur le bouton dédié, l'utilisateur est redirigé vers la page qui affiche les dernières mesures de température, d'humidité et de volume sonore.
- Les données sont récupérées en temps réel via l'API Flask.

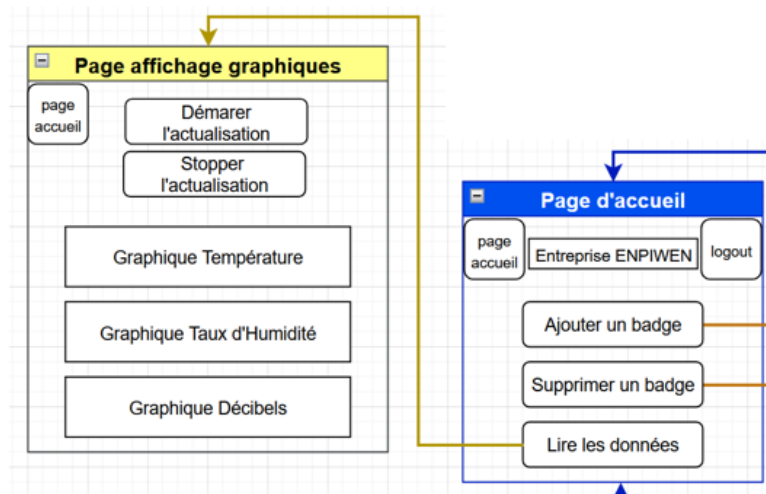


FIGURE 11 – Affichage des graphiques des capteurs

Page 'graph_capteur'

Script JavaScript pour les Graphiques : Le script JavaScript gère plusieurs fonctionnalités pour l'affichage et la mise à jour des graphiques de données en temps réel. C'est la seule solution que nous avons trouvée pour ajouter de l'interactivité à nos graphiques.

```
1 $(document).ready(function () {
2   var temperatureChart = new Chart(document.getElementById('temperatureChart'), {
3     type: 'line',
4     data: { labels: [], datasets: [{ label: 'Temp rature', data: [], borderColor: 'red'
5       , fill: false }] }
6   });
7
8   function loadInitialData() {
9     $.getJSON('/data', function(data) {
10       var now = new Date();
11       var labels = [];
12
13       for (var i = 9; i >= 0; i--) {
14         labels.push(new Date(now - i * 2000).toLocaleTimeString());
15       }
16
17       temperatureChart.data.labels = labels;
18       temperatureChart.data.datasets[0].data = data.temperature;
19       temperatureChart.update();
20     });
21   }
22
23   loadInitialData();
24   setInterval(function() {
25     $.getJSON('/data', function(data) {
26       var now = new Date().toLocaleTimeString();
27       var newTemperature = data.temperature[data.temperature.length - 1];
28       temperatureChart.data.labels.push(now);
29       temperatureChart.data.datasets[0].data.push(newTemperature);
30       if (temperatureChart.data.labels.length > 10) {
31         temperatureChart.data.labels.shift();
32         temperatureChart.data.datasets[0].data.shift();
33       }
34       temperatureChart.update();
35     });
36   }, 2000);
37 });
```

— Création des graphiques avec Chart.js :

- Trois graphiques sont créés pour afficher :
 - La température (courbe rouge)
 - L'humidité (courbe bleue)
 - Le volume sonore (courbe verte)
- Chaque graphique est associé à un élément <canvas> dans la page HTML via :
 - document.getElementById('temperatureChart')
 - document.getElementById('humiditeChart')

- `document.getElementById('volumeChart')`
- Initialement, les labels (les abscisses correspondant au temps) et les **données** (les ordonnées correspondant aux mesures) sont vides.
- **Récupération des données depuis /data :**
 - La fonction `loadInitialData()` envoie une requête AJAX pour récupérer des données au format JSON depuis /data (présenté plus tard) que l'on récupère depuis le serveur Flask.
- **Mise en forme des labels (axe X) :**
 - Un tableau `labels` est créé pour afficher les horodatages des mesures.
 - Les 10 derniers points de temps sont générés en soustrayant $i * 2000$ millisecondes (soit 2 secondes) de l'heure actuelle.
 - `new Date(...).toLocaleTimeString()` formate les heures pour les afficher de manière lisible.
- **Mise à jour des graphiques :**
 - Les **labels** sont mis à jour dans chaque graphique.
 - Les **données** sont mises à jour avec les valeurs récupérées depuis /data :
 - `data.temperature` pour la température
 - `data.humidity` pour l'humidité
 - `data.volume` pour le volume sonore
 - `chart.update()` est appelé pour redessiner chaque graphique avec les nouvelles valeurs.

Gestion des Données des Capteurs

Ces graphiques sont maintenant créés mais ils n'affichent encore rien. Dans `app.py`, nous avons donc créé la fonction `get_last_10_values` qui récupère les 10 dernières valeurs de la base de données. Nous avons ensuite créé une route `/data` qui permet de renvoyer les 10 dernières données pour chaque graphique.

```

1  @app.route('/data')
2  @login_required
3  def get_database_data():
4      """ Renvoie les 10 dernieres valeurs des bases de donn es SQLite """
5      rospy.loginfo("Recherche dans database")
6
7      temperature = get_last_10_values("dht11_temperature.db", "temperature", "
          temperature")
8      humidite = get_last_10_values("dht11_humidite.db", "humidite", "humidite")
9      volume = get_last_10_values("volumeMicro.db", "son", "volSon")
10     return jsonify({
11         "temperature": temperature,
12         "humidity": humidite,
13         "volume": volume
14     })

```

Voici à quoi ressemblent nos 3 graphiques (avec des données simulées)

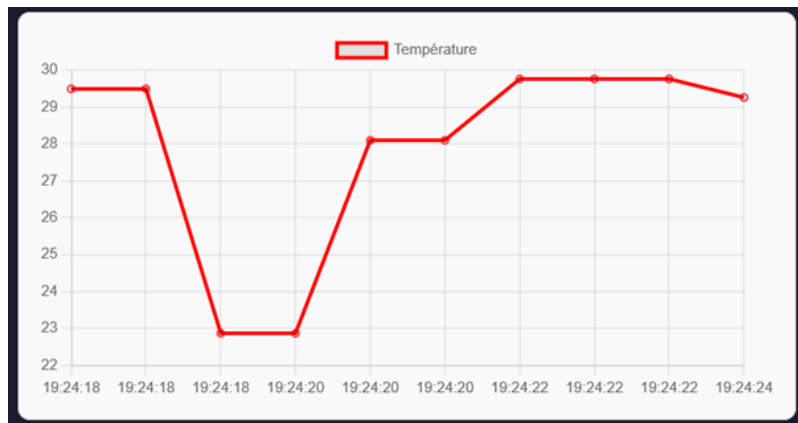


FIGURE 12 – Graphique température

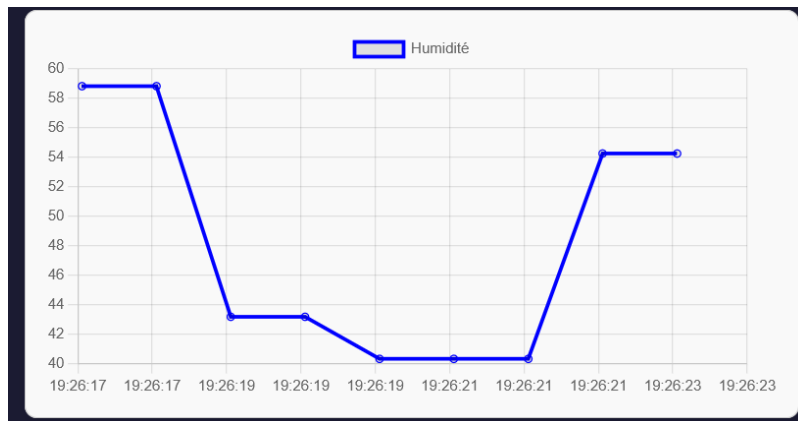


FIGURE 13 – Graphique humidité

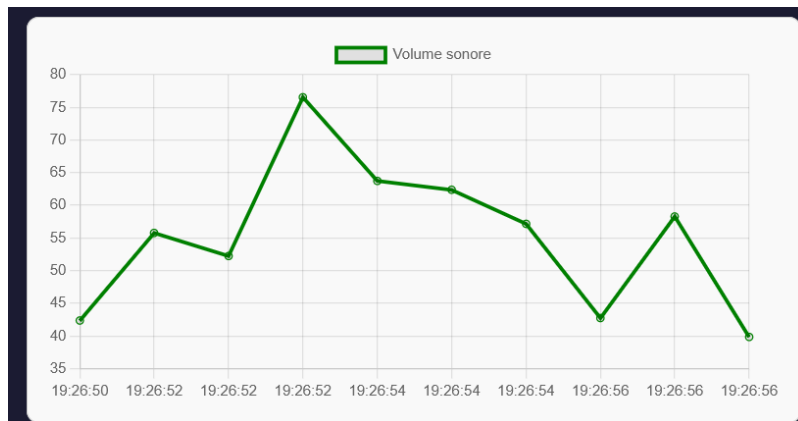


FIGURE 14 – Graphique volume sonore

Boutons de contrôle et formulaire

Dans cette page, nous pouvons visualiser les graphiques mais nous avons ajouté la possibilité d'arrêter temporairement l'affichage des données :

- **Boutons** « Démarrer » et « Arrêter » l'actualisation.

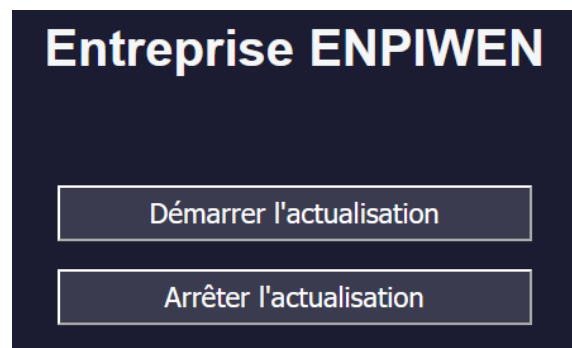


FIGURE 15 – Boutons démarrer l'actualisation et stopper l'actualisation

De plus, nous avons ajouté un petit formulaire qui permet, à la demande, de supprimer les données présentent sur un graphique spécifique :

- **Formulaire** permettant de réinitialiser certains graphiques.

Quel(s) graphique(s) voulez-vous réinitialiser ?

☒ Température

☐ Humidité

☐ Son

Submit

FIGURE 16 – Formulaire des graphiques à réinitialiser

3.7. Page d'Accueil → Page Badge et Utilisateur

Comme pour la page '/graph_captteur', nous avons besoin d'utiliser JavaScript afin d'obtenir un affichage de graphiques de manière dynamique sans avoir besoin de rafraîchir la page manuellement. Voici le code que nous avons utilisé dans la page '/gestion_utilisateur' et son principe de fonctionnement :

```

1 <script>
2     $(document).ready(function () {
3         function updateTables() {
4             $.getJSON('/rfid_data', function(data) {
5                 console.log("Donn es re ues :", data); // V rifie si la
6                     requ te fonctionne
7                 // Mise      jour du tableau des badges scann s
8                 $('#badgesTable tbody').empty();
9                 data.mesures.forEach(function(row) {
10                     $('#badgesTable tbody').append(
11                         '<tr><td>${row.date_time}</td><td>${row.numBadge}</td><
12                             td>${row.register}</td></tr>'
13                     );
14                 });
15                 // Mise      jour du tableau des utilisateurs enregistr s
16                 $('#usersTable tbody').empty();
17                 data.infos.forEach(function(row) {
18                     $('#usersTable tbody').append(
19                         '<tr><td>${row.numBadge}</td><td>${row.user}</td><td>${
20                             row.prenom}</td><td>${row.nom}</td><td>${row.age}</
21                             td><td>${row.mail}</td><td>${row.poste}</td></tr>'
22                     );
23                 });
24             });
25         }
26         updateTables(); // Charger les donn es au d marrage
27         setInterval(updateTables, 2000); // Rafra chir toutes les 2 secondes
28     });
29 </script>

```

Ce script en JavaScript utilise la bibliothèque jQuery pour mettre à jour dynamiquement deux tableaux HTML toutes les 2 secondes en récupérant des données depuis le fichier app.py. Le script commence par attendre que la page soit complètement chargée avant d'exécuter du code :

```
1 $(document).ready(function () {
```

La fonction updateTables() effectue une requête AJAX en GET vers /rfid_data (qui se situe dans app.py pour récupérer des données au format JSON.

- La console affiche les données reçues pour le débogage (console.log).
- Mise à jour des tableaux :
 - badgesTable : Contient les badges récemment scannés.
 - usersTable : Contient les informations des utilisateurs enregistrés.
- Avant d'ajouter de nouvelles données, on vide le contenu précédent avec empty() pour éviter les doublons.

Gestion des données du capteur RFID

Comme pour le cas de la page '/graph_capteur', ces tableaux sont maintenant créés mais ils n'affichent encore rien. Dans app.py, nous avons donc créé la fonction get_last_10_values_rfid_mesures qui récupère les 10 dernières valeurs de la base de données. Nous avons ensuite créé une route /rfid_data qui permet de renvoyer les 10 dernières données pour chaque tableau.

```
1 @app.route('/rfid_data')
2 @login_required
3 def get_database_data_rfid():
4     """ Renvoie les 10 dernières valeurs des bases de données SQLite rfid """
5     rospy.loginfo("Recherche dans database")
6
7     mesures = get_last_values_rfid_mesures("RFID_mesures.db", "mesures")
8     infos = get_rfid_infos("RFID_infos.db", "infos")
9
10    return jsonify({
11        "mesures": mesures,
12        "infos": infos,
13    })
```

La fonction updateTables() est exécutée immédiatement au chargement de la page, puis toutes les 2 secondes grâce à la fonction setInterval() :

```
1 updateTables(); // Charger les données au démarrage
2 setInterval(updateTables, 2000); // Rafraichir toutes les 2 secondes
```

Voici l'affichage que nous obtenons :

Derniers Badges Scannés		
Heure	Numéro de Badge	Enregistré
2025-03-23 17:48:07	26057	NO
2025-03-23 17:48:02	73065	NO
2025-03-23 17:47:57	69994	NO
2025-03-23 17:47:52	37045	NO
2025-03-23 17:47:47	32701	NO
2025-03-23 17:47:42	37039	NO
2025-03-23 17:47:37	53825	NO
2025-03-23 17:47:32	82514	NO
2025-03-19 13:07:04	90018	NO
2025-03-19 13:06:59	29968	NO

FIGURE 17 – Tableau des derniers badges scannés

Utilisateurs Enregistrés						
Numéro de Badge	Utilisateur	Prénom	Nom	Âge	Email	Rôle
999999	admin	admin	admin	99	admin@admin.com	admin
70254	eb474581	enzo	Boissenin	22	enzo.boissenin@gmail.com	jdsgfdhejzk

FIGURE 18 – Tableau des utilisateurs enregistrés

4. Logos

Afin d'ajouter un peu plus de fluidité au site, nous avons décidé de positionner un logo 'maison' en haut à droite de chaque page afin de permettre à l'utilisateur de revenir à la page principale en cas d'erreur :

```
1 <div id="logo1">
2   <a href="/page_accueil">
3     
5   </a>
  </div>
```

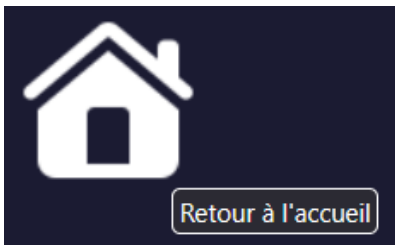


FIGURE 19 – Logo retour_accueil

De plus, nous avons aussi ajouté un bouton 'logout' qui permet à l'utilisateur de se déconnecter de sa session :

```
1 <div id="logo2">
2   <a href="/logout">
3     
5   </a>
  </div>
```

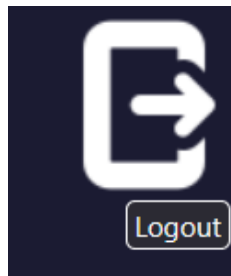


FIGURE 20 – Logo logout

V. Sécurité

Dans le cadre de notre projet de supervision industrielle, la sécurité est un enjeu crucial. Il est impératif de garantir que seules les personnes autorisées puissent interagir avec le système, de protéger les données contre les accès non autorisés et d'assurer la confidentialité des informations échangées. Pour cela, nous avons mis en place plusieurs mesures de sécurité, notamment l'utilisation du protocole HTTPS, un système d'authentification sécurisé et un contrôle d'accès basé sur les rôles.

1. Sécurisation des communications avec SSL/TLS

Pour assurer la confidentialité et l'intégrité des échanges entre les utilisateurs et notre application Flask, nous avons mis en place une connexion HTTPS en générant un certificat SSL auto-signé. Ce certificat permet de chiffrer les communications et de garantir que les données ne peuvent pas être interceptées par un tiers malveillant.

Nous avons généré notre certificat avec la commande suivante :

```
1 openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout server.key -out server.crt
```

Cette commande nous génère deux fichiers : server.key et server.crt.

Puis, nous avons configuré notre application Flask pour utiliser ce certificat, en indiquant le chemin de ces deux fichiers :

```
1 package_path = rospack.get_path('webinterface')
2 server_path = os.path.join(package_path, 'src', 'flask_project') #chemin d'accès
3 if __name__ == '__main__':
4     init_ros() # Initialiser ROS avant Flask
5     app.run(host="0.0.0.0", port=5000, ssl_context=(os.path.join(server_path, 'server.crt'), os.path.join(server_path, 'server.key')))
```

Ici le fichier est dans le package "web-interface" et à partir de ce package, on va en /src/-flask/project et on trouve nos deux fichiers. Cette configuration permet à notre serveur Flask de fonctionner en HTTPS, assurant ainsi la sécurité des échanges.

2. Système d'authentification et gestion des utilisateurs

Pour protéger l'accès aux fonctionnalités sensibles de notre application, nous avons mis en place un système d'authentification basé sur Flask-Login.

Utilisation de Flask-Login

Flask-Login est une extension qui permet de gérer l'authentification des utilisateurs dans une application Flask. Nous avons défini une classe `User` qui hérite de `UserMixin`. Cette classe permet de gérer les informations essentielles d'un utilisateur, notamment son identifiant, son nom d'utilisateur, son mot de passe haché et son rôle.

```
1 from flask_login import UserMixin
2
3 class User(UserMixin):
4     def init(self, id, username, password, role):
5         self.id = str(id)
6         self.username = username
7         self.password = password
8         self.role = role
9
10    def get_id(self):
11        return str(self.id)
```

Le module `UserMixin` fournit des méthodes standard nécessaires pour gérer l'authentification des utilisateurs, comme la récupération de l'ID utilisateur.

Chargement des utilisateurs

Pour charger un utilisateur à partir de son identifiant lors d'une session, nous utilisons la fonction `load_user`. Le décorateur `@login_manager.user_loader` dit à Flask-Login quelle fonction utiliser pour charger un utilisateur à partir de son identifiant stocké en session. :

```
1 @login_manager.user_loader
2 def load_user(user_id):
3     if user_id in users_cache: # Verifie si le user ne s'est pas deja connecte
4         return users_cache[user_id]
5
6     rospy.wait_for_service('login_serv')
7     try: # Service pour interoger base de donnee pour info utilisateur
8         login_service = rospy.ServiceProxy('login_serv', login_member)
9         response = login_service(user_id)
10        if response.success:
11            user = User(response.id, response.username, response.password, response.
12                        role)
13            users_cache[user_id] = user
14            return user
15        else:
16            return None
17    except rospy.ServiceException as e:
18        return None
19
20 def authenticate(username, password):
21     """ Verifie si l'utilisateur existe dans la base de donnees et si le mot de
22     passe est correct """
23     user = load_user(username) # Recupere l'objet User
24
25     if user:
26         if check_password_hash(user.password, password): # Verifie le mot de passe
27             hache
28             # Si le mot de passe est correct, renvoie l'objet utilisateur
29             return user
30         else:
31             return None # Mot de passe incorrect
32     else:
33         return None # Utilisateur non trouve
```

Cette fonction permet de récupérer les informations de l'utilisateur depuis un service ROS qui va interroger la base de données et de les stocker temporairement en mémoire.

Connexion et déconnexion des utilisateurs

Nous avons implémenté une route pour gérer la connexion des utilisateurs :

```
1 @app.route('/', methods=['GET', 'POST'])
2 def login():
3     if request.method == 'POST':
4         username = request.form['username'] # Recup nom utilisateur
5         password = request.form['password'] # Recup mot de passe
6         user = authenticate(username, password) # Verifie si mot de passe correct
7         if user:
8             login_user(user) # Si correct on connecte l'utilisateur
9             users_cache[user.id] = user # Stocke l'utilisateur pour eviter d'
              appeler ROS a chaque requete
10            print(f"Utilisateur connecte ? {current_user.is_authenticated}")
11            return redirect(url_for('page_accueil'))
12        else:
13            # Ecrit un message de mot de passe incorrect sur la page web
14            flash('Nom d\'utilisateur ou mot de passe incorrect', 'danger')
15
16    return render_template('login.html')
```

Et une autre pour gérer la déconnexion :

```
1 @app.route('/logout')
2 @login_required
3 def logout():
4     logout_user() # Deconnecte l'utilisateur
5     flash('Vous avez ete deconnecte')
6     return redirect(url_for('login'))
```

Contrôle d'accès basé sur les rôles

Nous avons défini deux rôles :

- **Admin** : Peut ajouter et supprimer des badges puis visualiser des données.
- **Utilisateur** : Peut uniquement visualiser des données.

Pour gérer ces permissions, nous avons créé un décorateur `admin_required` qui bloque l'accès aux pages sensibles pour les utilisateurs non-administrateurs :

```
1 from functools import wraps
2 from flask import abort
3
4 def admin_required(f):
5     @wraps(f)
6     def decorated_function(*args, **kwargs):
7         if current_user.role != 'admin': # Verifie si l'utilisateur est un admin
8             abort(403) # Acces interdit (Erreur HTTP 403)
9         return f(*args, **kwargs) # Execute la fonction si l'utilisateur est admin
10    return decorated_function
```

Nous utilisons ce décorateur sur les routes nécessitant un accès administrateur :

```
1 @app.route('/gestion_badges')
2 @login_required
3 @admin_required
4 def gestion_badges():
5     return render_template('gestion_badges.html')
```

Ainsi, les utilisateurs sans privilèges administratifs ne peuvent pas accéder aux fonctionnalités sensibles.

Conclusion

Grâce à ces différentes implémentations, notre système est sécurisé à plusieurs niveaux :

- Chiffrement des communications via SSL/TLS.
- Authentification des utilisateurs avec Flask-Login.
- Gestion des accès basée sur les rôles.

Ces mesures garantissent la protection des données et restreignent l'accès aux fonctionnalités sensibles uniquement aux utilisateurs autorisés.

VI. Implémentation

1. Diagramme fonctionnel

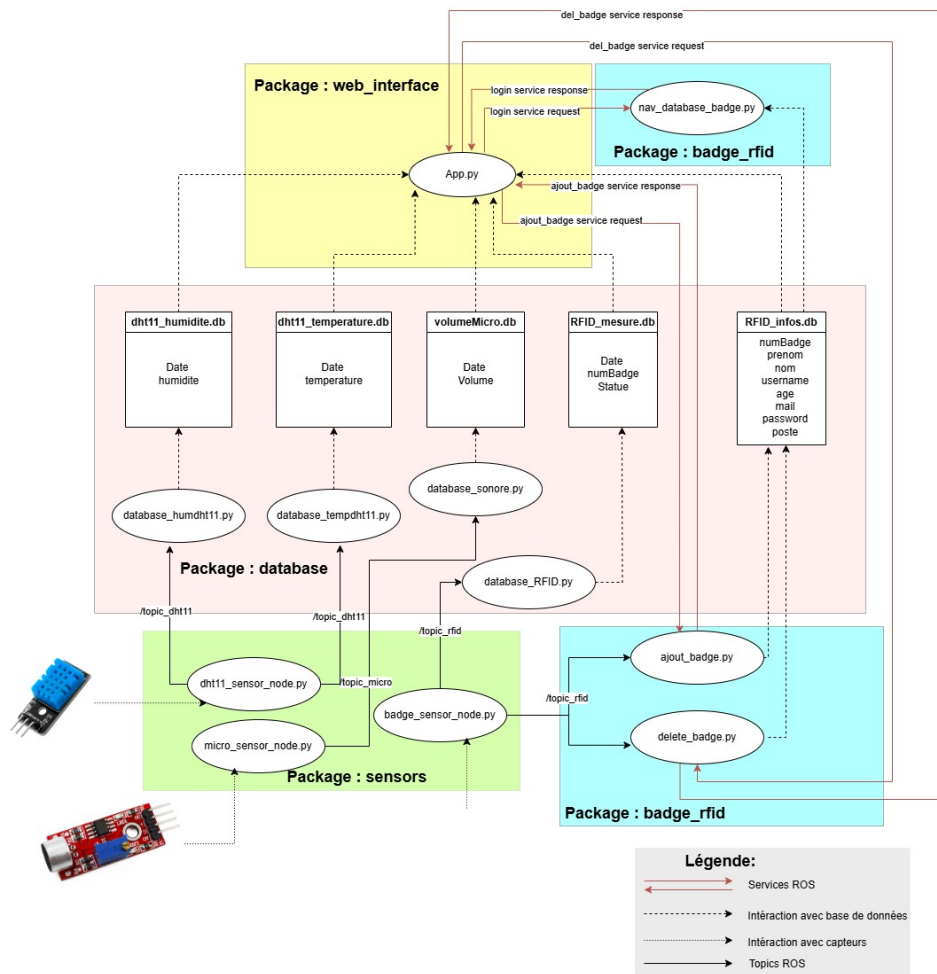


FIGURE 21 – Diagramme fonctionnel

Dans cette section, nous détaillons l'implémentation du système de supervision industrielle, en mettant en avant l'architecture logicielle et l'organisation des différents composants développés. La figure ci-dessus illustre la structure de notre application, qui repose sur une architecture modulaire intégrant des bases de données, des nœuds, des topics et des services ROS. On peut aussi voir les différents packages utilisés.

Notre système est structuré autour de plusieurs bases de données, dans lesquelles les différentes mesures et informations sont stockées :

- **dht11_humidite.db** et **dht11_temperature.db** : enregistrent les mesures d'humidité et de température du capteur DHT11.
- **volumeMicro.db** : stocke les valeurs de volume sonore captées.
- **RFID_mesure.db** : conserve les informations des badges RFID scannés avec leur statut.
- **RFID_infos.db** : contient les informations personnelles des utilisateurs associés aux badges (numéro, nom, âge, mail, etc.).

Les bases de données sont alimentées par des scripts Python dédiés qui récupèrent les données des capteurs et les enregistrent de manière structurée.

La communication entre les différentes parties du système repose sur l'utilisation de **nœuds ROS** (représentés par des ovales dans le schéma), qui échangent des messages via des **topics** (flèches noires).

Les principaux nœuds ROS sont :

- **dht11_sensor_node.py** et **micro_sensor_node.py** : publient les mesures des capteurs DHT11 et du capteur sonore sur les topics respectifs `/topic_dht11` et `/topic_micro`.
- **badge_sensor_node.py** : interagit avec les badges RFID et publie les données sur `/topic_rfid`.

L'application principale, **App.py**, centralise l'interaction avec les données et coordonne les échanges entre les différents composants. Pour la gestion des utilisateurs RFID, nous utilisons des **services ROS** (représentés par les flèches rouges) :

- **ajout_badge.py** : permet d'ajouter un badge à la base de données RFID.
- **nav_database_badge.py** : permet de vérifier si un utilisateur est enregistré dans la base de donnée pour la connexion au site web.
- **delete_badge.py** : permet de supprimer un badge existant.

Enfin, plusieurs scripts assurent la lecture des bases de données et leur exploitation :

- **database_RFID.py** : lit les données des badges RFID.
- **database_sonore.py** : récupère les mesures du capteur sonore.
- **database_humdht11.py** et **database_tempdht11.py** : lisent respectivement l'humidité et la température enregistrées.

Tous ces codes sont contenus dans plusieurs packages différents, pour rendre l'architecture plus modulaire. Au total, nous avons 5 packages mais seulement 4 packages contenant réellement du code.

- **sensors** : Ce package stocke les nœuds qui lisent les données des capteurs et qui publient les données sur les topics. C'est la zone de couleur verte sur le schéma.
- **database** : Ce package stocke les nœuds qui écoute les topics contenant les données des capteurs et inscrivent les infos dans les databases. C'est la zone de couleur rouge sur le schéma.

Ce package contient aussi les databases.

- **badge_rfid** : C'est un package un peu plus spécial. Il permet la gestion du badge et des utilisateurs dans la base de données. C'est ce package qui contient les différents services et serveur de service pour ajouter, supprimer ou naviguer dans la database à partir du site web. C'est la zone de couleur bleue sur le schéma.
- **web_interface** : Ce package contient le noeud qui fait tourner l'interface web et toute les pages html et css qui vont avec. C'est la zone de couleur jaune sur le schéma.
- **main** : Ce dernier package ne contient pas de code mais simplement des fichier .launch permettant de démarrer les différents noeuds.

Cette architecture modulaire permet d'assurer une gestion efficace des données et une communication fluide entre les différentes parties du système.

Pour développer depuis chez soi sans être dépendant de la carte Raspberry Pi et des capteurs, nous avons créé des noeuds dans le package **sensors**, permettant de publier des données aléatoires sur les topics. Cela nous permet de simuler un capteur pour le développement des autres noeuds. Ainsi dans le package **main**, nous retrouvons un fichier de lancement principal :

system.launch avec les capteurs

Et un fichier de lancement pour développer sans les capteurs : **system_without_sensor.launch**

2. Structures des bases de données

Comme mentionné précédemment, le stockage des données de ce projet est basé sur des bases de données SQLite3. Dans cette section, vous trouverez plus d'informations concernant le contenu et la structure de celles-ci.

Base de données *dht11_temperature.db*

Cette base de données sert à stocker les données relatives à la température mesurées par le capteur DHT11. La table créée dans ce fichier, nommée "temperature", se compose de la manière suivante :

id (INTEGER)	date_time (TEXT)	temperature (REAL)
1	heure n°1	20.0
2	heure n°2	21.5
3	heure n°3	23.0

Le paramètre "id" correspond au numéro de la mesure, "date_time" correspond à l'heure à laquelle a été prise la mesure, et "temperature" est la valeur de la température relevée par le capteur au moment de la mesure.

Base de données *dht11_humidite.db*

Cette base de données sert à stocker les données relatives à l'humidité mesurée par le capteur DHT11. La table créée dans ce fichier, nommée "humidite", se compose de la manière suivante :

id (INTEGER)	date_time (TEXT)	humidite (REAL)
1	heure n°1	50.0 (%)
2	heure n°2	61.2 (%)
3	heure n°3	58.0 (%)

Le paramètre "id" correspond au numéro de la mesure, "date_time" correspond à l'heure à laquelle a été prise la mesure, et "humidite" est le pourcentage d'humidité présent dans l'air relevé par le capteur au moment de la mesure.

Base de données *volumeMicro.db*

Cette base de données sert à stocker les mesures du niveau sonore enregistré par le capteur KY-038 (micro). La table créée dans ce fichier, nommée "son", se compose de la manière suivante :

id (INTEGER)	date_time (TEXT)	volSon (REAL)
1	heure n°1	10.17 (db)
2	heure n°2	51.02 (db)
3	heure n°3	64.00 (db)

Le paramètre "id" correspond au numéro de la mesure, "date_time" correspond à l'heure à laquelle a été prise la mesure, et "volSon" est le nombre de décibels mesuré par le capteur au moment de la mesure.

Base de données *RFID_mesures.db*

Cette base de données sert à stocker les passages de badge RFID devant le capteur ainsi que leur état (enregistré ou non). Elle se structure de la manière suivante :

id (INTEGER)	date_time (TEXT)	numBadge (int32)	register (TEXT)
1	heure n°1	147852369	NO
2	heure n°2	789456123	YES
3	heure n°3	369852147	YES

Le paramètre "id" correspond au numéro de la mesure, "date_time" correspond à l'heure à laquelle le badge a été placé devant le capteur, "numBadge" correspond au numéro affecté au badge, et "register" correspond au statut d'enregistrement du badge. Pour déterminer si un badge est enregistré ou non, il y a une comparaison du numéro de badge avec la table présente dans RFID_infos.db. Vous trouverez la description de celle-ci juste ci-dessous.

Base de données *RFID_infos.db*

Cette base de données sert à stocker les informations relatives aux badges RFID. Cette base de données se remplit lorsque l'utilisateur décide d'ajouter un badge à la liste des badges enregistrés via l'interface web.

id (INTEGER)	numBadge (INTEGER)	user (TEXT)	prenom (TEXT)	nom (TEXT)	age (INTEGER)	mail (TEXT)	mdp (TEXT)	poste (TEXT)
1	147852369	admin	Admin	Admin	99	admin@admin.com	mdpCryptéAdmin	admin
2	987456321	username1	Jean	Dupont	35	mail@mail.com	mdpCryptéUser	user
3	987456321	username2	Jack	Dutrand	35	mail2@mail2.com	mdpCryptéUser2	user

Le paramètre "id" correspond au numéro de la ligne contenant les informations d'un badge, "numBadge" est le numéro du badge RFID qui est enregistré, "user" correspond au nom d'utilisateur de la personne enregistrée (utile pour se connecter sur la page web), "prenom" correspond au prénom de l'utilisateur enregistré, "nom" est le nom de l'utilisateur enregistré, "age" correspond à son âge, "mail" est l'adresse électronique de l'utilisateur, "mdp" correspond au mot de passe lié au nom d'utilisateur pour se connecter, celui-ci étant crypté pour rajouter de la sécurité. Enfin, "poste" correspond au rôle de l'utilisateur enregistré au sein de l'entreprise.

3. Structure des Messages et Services ROS

Dans notre projet, nous utilisons plusieurs types de messages et services ROS pour l'échange de données entre les différents nœuds. Ces messages sont soit issus des types standards fournis par `std_msgs`, soit définis sous forme de messages personnalisés.

Messages ROS

Nous avons trois types principaux de messages utilisés dans notre architecture :

- **Message contenant le numéro de badge et le volume du micro** : Ces messages utilisent directement le type `Float32` provenant de la bibliothèque standard de ROS :

```
1 from std_msgs.msg import Float32
```

- **Message personnalisé pour le capteur DHT11** : Étant donné que le capteur DHT11 fournit deux valeurs (température et humidité), nous avons défini un message personnalisé nommé `dht11.msg` sous le package `sensors` :

```
1 from sensors.msg import dht11
```

Sa structure est la suivante :

```
1 float32 temperature
2 float32 humidity
```

Services ROS

Trois services principaux sont utilisés pour la gestion des badges et l'authentification des membres. Voici leurs définitions :

- **Service ajout_badge.srv** : Ce service permet l'ajout d'un badge en prenant plusieurs informations en entrée et en retournant un booléen indiquant le succès de l'opération.

```
1  string prenom
2  string nom
3  string username
4  int32 age
5  string mail
6  string password
7  string poste
8  ---
9  bool success
```

- **Service suppr_badge.srv** : Ce service est utilisé pour la suppression d'un badge et retourne une validation de l'opération. Il prend en entrée le numéro de badge à supprimer et retourne si l'opération a bien été réalisé.

```
1  bool suppr_badge
2  ---
3  bool validation
```

- **Service login_member.srv** : Ce service permet l'authentification d'un utilisateur en fonction de son username et retourne les informations associées en cas de succès.

```
1  string username
2  ---
3  bool success
4  int32 id
5  string username
6  string password
7  string role
```

Ces messages et services constituent la base de communication de notre architecture ROS et permettent une interaction efficace entre les différents modules.

VII. Explication des codes

Code concernant les databases

Base de données pour l'humidité

Création de la base de donnée

Avant de pouvoir travailler avec une base de donnée, il faut la créer. Pour cela, nous utilisons la fonction `create_database()` suivante :

```
1 db_path=os.path.join(package_path, 'database', 'dht11_humidite.db') #chemin d'accès
2
3 # Creation de la base de donnees
4 def create_database():
5
6     conn = sqlite3.connect(db_path)
7     cursor = conn.cursor()
8     try :
9         cursor.execute('''
10             CREATE TABLE IF NOT EXISTS humidite (
11                 id INTEGER PRIMARY KEY AUTOINCREMENT,
12                 date_time TEXT NOT NULL,
13                 humidite REAL NOT NULL
14             )
15         ''')
16         rospy.loginfo("Creation de la table pour humidite") #DEBUG
17     except sqlite3.Error as e:
18         rospy.logerr(f"Erreur lors de la creation : {e}")
19     finally:
20         conn.commit()
21         conn.close()
```

La première étape est de déterminer à quel emplacement la base de données est créée. Cela permettra de vérifier si elle n'est pas déjà existante par la suite, et de savoir à quel endroit la chercher lorsque l'on voudra entrer des informations à l'intérieur.

Ensuite, on crée un "curseur" Sqlite, qui permet d'interagir avec les bases de données. On affecte celui-ci au chemin d'accès que l'on a spécifié. On construit une table appelée *humidite* si elle n'existe pas déjà (vérifié avec l'instruction Sqlite **IF NOT EXISTS**). Si tout se passe correctement, un message est affiché indiquant la création de la table, sinon on affiche l'erreur générée, puis l'on "ferme" notre curseur, afin d'éviter toute modification inattendue sur notre table.

Lecture des données via topic ROS

Afin de récupérer les données du capteur et pouvoir les enregistrer, nous allons nous inscrire à un topic ROS appelé "topic_dht11". Celui-ci envoie un message contenant notamment une partie "humidity", que l'on va extraire pour la stocker dans une variable, qui nous servira plus tard à l'écriture dans la base de données.

```
1 # Callback pour traiter les messages du topic
2 def hum_callback(msg):
3     humidite = msg.humidity # La temperature est stockee dans msg.data
4     rospy.loginfo(f"humidite recue : {humidite} %")
5     insert_measurement(humidite) # Enregistrer dans la base de donnees
```

Écriture la base de donnée

Une fois la table créée, on peut y entrer les informations que l'on souhaite enregistrer. Avec la commande **INSERT INTO**, on peut renseigner les valeurs qui nous seront utiles. Ici, il s'agit de l'heure à laquelle a été prise la mesure ainsi que le pourcentage d'humidité. Pour ce faire, on ouvre un curseur comme précédemment, on insère dans la table les valeurs récupérées du topic ROS concerné, puis l'on ferme ce curseur.

```
1 # Insertion des mesures dans la base de donnees
2 def insert_measurement(humidite):
3     conn = sqlite3.connect(db_path)
4     cursor = conn.cursor()
5     cursor.execute('''
6         INSERT INTO humidite (date_time, humidite)
7         VALUES (?, ?)
8     ''', (datetime.now().strftime('%Y-%m-%d %H:%M:%S'), humidite))
9     rospy.loginfo(f"Écriture dans la table (humidite) : {humidite} %") #DEBUG
10    conn.commit()
11    conn.close()
```

Enfin, toutes ces fonctions sont appelées par la fonction principale du code, dont le fonctionnement reste simple et est détaillé dans les commentaires :

```

1  # Noeud ROS pour ecouter le topic et enregistrer les donnees
2  def hum_listener():
3      # Initialisation du noeud ROS
4      rospy.init_node('humidite_listener', anonymous=True)
5
6      # S abonner au topic "topic_tempDHT11" pour recuperer les donnees de l humidite
7      rospy.Subscriber('/topic_dht11', dht11, hum_callback)
8
9      # Creer la base de donnees si elle n existe pas
10     create_database()
11
12     rospy.loginfo("Ecoule du topic 'topic_humDHT11'. Enregistrement de l humidite
13                     dans la base de donnees.")
14
15     # Maintenir le noeud actif
16     rospy.spin()
17
18 # Programme principal
19 if __name__ == '__main__':
20     try:
21         hum_listener()
22     except rospy.ROSInterruptException:
23         pass

```

Base de données pour la température

Création de la base de donnée

Le fonctionnement de la création de la base de données pour la température est le même que pour l'humidité, seul le chemin d'accès et le nom de la table changent, s'appelant maintenant "temperature" :

```

1  db_path=os.path.join(package_path, 'database', 'dht11_temperature.db') #chemin d'
   acces
2
3  # Creation de la base de donnees
4  def create_database():
5
6      conn = sqlite3.connect(db_path)
7      cursor = conn.cursor()
8      try :
9          cursor.execute('''
10              CREATE TABLE IF NOT EXISTS temperature (
11                  id INTEGER PRIMARY KEY AUTOINCREMENT,
12                  date_time TEXT NOT NULL,
13                  temperature REAL NOT NULL
14              )
15          ''')
16      rospy.loginfo("Creation de la table pour la temperature") #DEBUG
17  except sqlite3.Error as e:
18      rospy.logerr(f"Erreur lors de la creation : {e}")
19  finally:
20      conn.commit()
21      conn.close()

```

Lecture des données via topic ROS

Pour la lecture des informations, le principe est aussi similaire à la lecture de l'humidité, à l'exception que cette fois-ci nous lisons la partie "temperature" du message envoyé par le topic "topic_dht11".

```
1 # Callback pour traiter les messages du topic
2 def temperature_callback(msg):
3     temperature = msg.temperature # La temperature est stockee dans msg.data
4     rospy.loginfo(f"Temperature recue : {temperature} degrees")
5     insert_measurement(temperature) # Enregistrer dans la base de donnees
```

Écriture la base de donnée

De même pour l'écriture, le principe est exactement identique à l'humidité.

```
1 # Insertion des mesures dans la base de donnees
2 def insert_measurement(temperature):
3     conn = sqlite3.connect(db_path)
4     cursor = conn.cursor()
5     cursor.execute('''
6         INSERT INTO temperature (date_time, temperature)
7         VALUES (?, ?)
8     ''', (datetime.now().strftime('%Y-%m-%d %H:%M:%S'), temperature))
9     rospy.loginfo(f"Écriture dans la table (temperature) : {temperature} degrees") #
10    DEBUG
11    conn.commit()
12    conn.close()
```

Enfin, toutes ces fonctions sont appelées par la fonction principale du code, toujours similaire à la version pour l'humidité :

```
1 # Noeud ROS pour ecouter le topic et enregistrer les donnees
2 def hum_listener():
3     # Initialisation du noeud ROS
4     rospy.init_node('temperature_listener', anonymous=True)
5
6     # S abonner au topic "topic_tempDHT11" pour recuperer les donnees de temperature
7     rospy.Subscriber('/topic_dht11', dht11, temperature_callback)
8
9     # Creer la base de donnees si elle n existe pas
10    create_database()
11
12    rospy.loginfo("Ecoute du topic 'topic_humDHT11'. Enregistrement de la
13        temperature dans la base de donnees.")
14
15    # Maintenir le noeud actif
16    rospy.spin()
17
18 # Programme principal
19 if __name__ == '__main__':
20     try:
21         hum_listener()
22     except rospy.ROSInterruptException:
23         pass
```


Base de données pour le volume sonore

Création de la base de donnée

Le fonctionnement de la création de la base de données pour la température est le même que pour l'humidité, seul le chemin d'accès et le nom de la table changent, s'appelant maintenant "son" :

```
1 db_path=os.path.join(package_path, 'database', 'volumeMicro.db') #chemin d'accès
2
3 # Creation de la base de donnees
4 def create_database():
5
6     conn = sqlite3.connect(db_path)
7     cursor = conn.cursor()
8     try :
9         cursor.execute('''
10             CREATE TABLE IF NOT EXISTS son (
11                 id INTEGER PRIMARY KEY AUTOINCREMENT,
12                 date_time TEXT NOT NULL,
13                 volSon REAL NOT NULL
14             )
15         ''')
16         rospy.loginfo("Creation de la table pour le volume sonore") #DEBUG
17     except sqlite3.Error as e:
18         rospy.logerr(f"Erreur lors de la creation : {e}")
19     finally:
20         conn.commit()
21         conn.close()
```

Lecture des données via topic ROS

A la différence des deux précédentes bases de données, celle pour le volume sonore enregistre les mesures envoyées sur le topic "topic_micro". Le message de celui-ci n'est composé que du volume sonore relevé par le micro, on peut donc enregistrer celui-ci directement dans la variable "volSon" en utilisant *msg.data* :

```
1 # Callback pour traiter les messages du topic
2 def sound_callback(msg):
3     volSon = msg.data # La temperature est stockee dans msg.data
4     rospy.loginfo(f"volume sonore recu : {volSon}")
5     insert_measurement(volSon) # Enregistrer dans la base de donnees
```

Écriture la base de donnée

De même pour l'écriture, le principe est exactement identique à l'humidité et à la température.

```
1 # Insertion des mesures dans la base de donnees
2 def insert_measurement(volSon):
3     conn = sqlite3.connect(db_path)
4     cursor = conn.cursor()
5     cursor.execute('''
6         INSERT INTO son (date_time, volSon)
7         VALUES (?, ?)
8     ''', (datetime.now().strftime('%Y-%m-%d %H:%M:%S'), volSon))
9     rospy.loginfo(f"Ecriture dans la table (son) : {volSon}") #DEBUG
10    conn.commit()
11    conn.close()
```

Enfin, toutes ces fonctions sont appelées par la fonction principale du code, toujours similaire à la version pour l'humidité et la température :

```
1     # Noeud ROS pour ecouter le topic et enregistrer les donnees
2 def sound_listener():
3     # Initialisation du noeud ROS
4     rospy.init_node('sound_listener', anonymous=True)
5
6     # S'abonner au topic "topic_micro" pour recuperer les donnees du micro
7     rospy.Subscriber('topic_micro', Float32, sound_callback)
8
9     # Creer la base de donnees si elle n'existe pas
10    create_database()
11
12    rospy.loginfo("Ecoule du topic 'topic_micro'. Enregistrement du volume sonore
13                  dans la base de donnees.")
14
15    # Maintenir le noeud actif
16    rospy.spin()
17
18 # Programme principal
19 if __name__ == '__main__':
20     try:
21         sound_listener()
22     except rospy.ROSInterruptException:
23         pass
```

Base de données pour les relevés de badge RFID

En ce qui concerne les bases de données pour le badge RFID, le fonctionnement est légèrement différent de celles que l'on a pu décrire précédemment. Il en existe 2, "infos" et "mesures" qui sont reliées et partagent des informations en commun. Tout sera expliqué dans les sections et sous-sections qui suivent :

Écriture la base de donnée

La première différence avec les codes précédents est qu'ici nous déclarons deux chemins d'accès pour les deux bases de données. Vous comprendrez pourquoi un peu plus loin.

```
1      db_path_mesures=os.path.join(package_path, 'database', 'RFID_mesures.db') #
      chemin d'accès
2 db_path_infos=os.path.join(package_path, 'database', 'RFID_infos.db') #chemin d'
      accès
3
4
5 # Creation de la base de donnees des mesures
6 def create_database_mesure():
7     """Creation de la table pour les mesures RFID."""
8     conn = sqlite3.connect(db_path_mesures)
9     cursor = conn.cursor()
10    try:
11        cursor.execute('''
12            CREATE TABLE IF NOT EXISTS mesures (
13                id INTEGER PRIMARY KEY AUTOINCREMENT,
14                date_time TEXT NOT NULL,
15                numBadge INT32 NOT NULL,
16                register TEXT NOT NULL
17            )
18        ''')
19        rospy.loginfo("Creation de la table pour les releves du badge") # DEBUG
20    except sqlite3.Error as e:
21        rospy.logerr(f"Erreur lors de la creation de la table : {e}")
22    finally:
23        conn.commit()
24        conn.close()
```

Vérification de l'enregistrement

Une fonction supplémentaire par rapport aux codes précédents est la vérification d'enregistrement des badges dans la deuxième base de données. C'est pour cela que l'on a déclaré deux chemins d'accès au début du programme. Cette fonction se connecte à la table "infos", là où sont enregistrés les badges des membres de l'entreprise avec la commande **SELECT * FROM infos WHERE numBadge = ?**. Si le numéro du badge est enregistré, la fonction retourne les informations, sinon elle retourne FALSE.

```

1  # Verification si un badge est present dans la base de donnees infos
2  def check_badge_in_infos(num_badge):
3      """Verifie si un numero de badge est dans la base de donnees infos."""
4      conn = sqlite3.connect(db_path_infos)
5      cursor = conn.cursor()
6      try:
7          cursor.execute('SELECT * FROM infos WHERE numBadge = ?', (num_badge,))
8          result = cursor.fetchone()
9          return result is not None
10     except sqlite3.Error as e:
11         rospy.logerr(f"Erreur lors de la verification du badge : {e}")
12         return False
13     finally:
14         conn.close()

```

Écriture dans la base de données

Pour écrire dans la base de données, la fonction lit le message du topic "rfid_listener", vérifie si le badge est connu avec la fonction décrite ci-dessus. Si c'est le cas, la valeur du statut d'enregistrement *register* sera YES, sinon par défaut il est défini à NO. Ce numéro de badge et le statut d'enregistrement sont ensuite écrits dans la table "mesures" avec l'heure à laquelle le badge a été placé devant le capteur.

```

1  # Fonction de callback pour les messages du topic
2  def rfid_callback(msg):
3      """Callback appelee a la reception d'un message sur le topic."""
4      num_badge = msg.data # Supposant que msg.data contient le numero du badge
5      timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
6
7      # Verifier si le badge est connu
8      is_known = check_badge_in_infos(num_badge)
9      register_value = "YES" if is_known else "NO"
10
11     # Enregistrement dans la base de donnees des mesures
12     conn = sqlite3.connect(db_path_mesures)
13     cursor = conn.cursor()
14     try:
15         cursor.execute('''
16             INSERT INTO mesures (date_time, numBadge, register)
17             VALUES (?, ?, ?)
18             ''', (timestamp, num_badge, register_value))
19         rospy.loginfo(f"Badge {num_badge} lu a {timestamp}, statut d'enregistrement
20             : {register_value} ") # DEBUG
21     except sqlite3.Error as e:
22         rospy.logerr(f"Erreur lors de l'insertion des donnees : {e}")
23     finally:
24         conn.commit()
25         conn.close()

```

Base de données pour les infos de badge RFID

Les fonctions de cette base de données ont été réparties dans 3 codes python : *ajout_badge.py*, *delete_badge.py* et *nav_database_badge.py*. Afin de ne pas surcharger ce rapport qui est déjà suffisamment important ainsi, nous ne détaillerons pas autant toutes les fonctions des différents codes mais seulement les plus complexes et importantes. La majorité des explications est lisible directement dans les commentaires des codes.

Ajout_badge.py

Ce code permet d'ajouter un badge ainsi que les informations qui lui sont reliées dans la base de données "infos". Il permet aussi la création de celle-ci et l'écriture de la première ligne, qui sont les identifiants administrateurs qui permettent de rajouter de nouveaux badges lors de la première connexion ou s'il y a une réinitialisation de la base de données.

```
1 db_path=os.path.join(package_path, 'database', 'RFID_infos.db') #chemin d'accès
2
3 # Variable globale pour stocker le numero de badge
4 global_badge = None
5
6 def create_database_infos():
7     """Creation de la table pour les infos du badge."""
8     conn = sqlite3.connect(db_path)
9     cursor = conn.cursor()
10    try:
11        cursor.execute('''
12            CREATE TABLE IF NOT EXISTS infos (
13                id INTEGER PRIMARY KEY AUTOINCREMENT,
14                numBadge INTEGER NOT NULL,
15                user TEXT NOT NULL,
16                prenom TEXT NOT NULL,
17                nom TEXT NOT NULL,
18                age INTEGER NOT NULL,
19                mail TEXT NOT NULL,
20                mdp TEXT NOT NULL,
21                poste TEXT NOT NULL
22            )
23        ''')
24        rospy.loginfo("Creation de la table pour les infos du badge") # DEBUG
25    except sqlite3.Error as e:
26        rospy.logerr(f"Erreur lors de la creation de la table : {e}")
27    finally:
28        hashed_password_admin = generate_password_hash("admin", method='pbkdf2:
29            sha256')
30        cursor.execute('''
31            INSERT INTO infos (numBadge,user, prenom, nom, age, mail, mdp, poste)
32            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
33            ''', (999999,"admin", "admin", "admin", 99, "admin@admin.com",
34                hashed_password_admin, "admin"))
35        conn.commit()
36        conn.close()
```

```

1 def ajout_badge_base(req):
2     """Ajoute les informations du badge dans la base de donnees."""
3     global global_badge
4     rospy.loginfo(f"DEBUG: Valeur actuelle de global_badge: {global_badge}")
5
6     if global_badge is None:
7         rospy.logerr("Aucun badge detecte !")
8         return ajout_badgeResponse(False)
9
10    rospy.loginfo(f"Infos utilisateur recues : {req.prenom} {req.nom},username:{req.
11        username}, Age: {req.age}, mail: {req.mail}, MDP : {req.password}, Role: {
12        req.poste}")
13
14    hashed_password = generate_password_hash(req.password, method='pbkdf2:sha256')
15
16    # Enregistrement dans la base de donnees
17    conn = sqlite3.connect(db_path)
18    cursor = conn.cursor()
19    try:
20        rospy.loginfo("DEBUG: Insertion en cours dans la base de donnees...")
21        cursor.execute('''
22            INSERT INTO infos (numBadge,user, prenom, nom, age, mail, mdp, poste)
23            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
24        ''', (global_badge, req.username, req.prenom, req.nom, req.age, req.mail,
25            hashed_password, req.poste))
26
27        rospy.loginfo(f"Badge {global_badge} enregistre avec succes")
28        success = True
29    except sqlite3.Error as e:
30        rospy.logerr(f"Erreur lors de l'insertion des donnees : {e}")
31        success = False
32    finally:
33        conn.commit()
34        conn.close()
35
36    return ajout_badgeResponse(success)

```

delete_badge.py

Ce code permet de supprimer un badge de la table "infos", ce qui correspond à ne plus considérer un numéro de badge spécifique comme enregistré dans la liste de ceux connus.

```
1 db_path=os.path.join(package_path, 'database', 'RFID_infos.db') #chemin d'accès
2 def del_badge_base(req):
3     """Supprime un badge de la base de donnees."""
4     global global_badge
5
6     print(f"[DEBUG] Valeur de global_badge avant suppression : {global_badge}")
7     sys.stdout.flush()
8
9     if global_badge is None:
10        rospy.logwarn("Aucun badge recu.")
11        print("[ERROR] Aucun badge recu.")
12        sys.stdout.flush()
13        return suppr_badgeResponse(validation=False)
14
15    conn = None # Initialise conn pour eviter l'erreur
16
17    try:
18        conn = sqlite3.connect(db_path)
19        cursor = conn.cursor()
20
21        print(f"[DEBUG] Suppression du badge en cours : {global_badge}")
22        sys.stdout.flush()
23
24        cursor.execute("DELETE FROM infos WHERE numBadge = ?", (global_badge,))
25        conn.commit()
26
27        print(f"[DEBUG] total_changes apres suppression : {conn.total_changes}")
28        sys.stdout.flush()
29
30        if conn.total_changes > 0:
31            rospy.loginfo(f"Badge {global_badge} supprime.")
32            print(f"[SUCCESS] Badge {global_badge} supprime.")
33            sys.stdout.flush()
34            return suppr_badgeResponse(validation=True)
35        else:
36            rospy.logwarn(f"Aucune entree pour le badge : {global_badge}")
37            print(f"[WARNING] Aucune entree trouvee pour le badge {global_badge}.")
38            sys.stdout.flush()
39            return suppr_badgeResponse(validation=False)
```

Nav_database_badge.py

Ce code est utile pour l'extraction d'informations de la table "infos" pour le service de login de l'interface web. Pour un nom d'utilisateur précis, ce code va chercher l'id, le mot de passe et le poste relatifs à celui-ci. En cas d'erreur ou d'utilisateur introuvable, toutes ses informations sont retournées avec None, ce qui concrètement entraînera un refus d'accès aux fonctionnalités de la page web.

```
1 db_path=os.path.join(package_path, 'database', 'RFID_infos.db') #chemin d'accès
2
3 def extract_data(req):
4     username = req.username
5
6     conn = sqlite3.connect(db_path)
7     cursor = conn.cursor()
8     try:
9
10         cursor.execute('''SELECT id, mdp, poste FROM infos WHERE user = ?;''', (
11             username,))
12         data_extracted = cursor.fetchone() # Stockage des donnees dans une variable
13
14         if data_extracted:
15             id_, password, role = data_extracted # Separation des informations dans
16                 des variables distinctes
17
18             rospy.loginfo("Donnees extraites avec succes")
19             success = True
20         else:
21             rospy.logwarn(f"Aucune donnee trouvee pour l'utilisateur {username}")
22             success = False
23             id_ = None
24             username=None
25             role =None
26             password=None
27
28     except sqlite3.Error as e:
29         rospy.logerr(f"Erreur lors de l'extraction des donnees : {e}")
30         success = False
31         id_ = None
32         username=None
33         role =None
34         password=None
35
36     finally:
37         conn.commit()
38         conn.close()
39
40     return login_memberResponse(success,id_, username, password, role) # Retour des
    resultats pour le service ROS
```


Code concernant la lecture de capteurs

Le package `sensor` a pour objectif d'acquérir les données de capteurs et de les publier sur des topics ROS. Il est composé de trois nœuds :

- `node_dht11` : Acquisition de la température et de l'humidité avec un capteur DHT11.
- `node_rfid` : Lecture des badges RFID via un lecteur RC522.
- `node_micro` : Acquisition du niveau sonore en dB à l'aide d'un capteur analogique branché sur une carte Arduino.

Chaque nœud fonctionne indépendamment et publie ses mesures sur un topic ROS.

Température et humidité

Ce nœud utilise un capteur **DHT11** pour mesurer la température et l'humidité toutes les 5 secondes. Les valeurs sont publiées dans un message ROS personnalisé `dht11.msg`.

Fonctionnement :

1. Le capteur est connecté sur le GPIO 2 de la Raspberry Pi.
2. Une boucle périodique tente de lire les valeurs.
3. Si la lecture est réussie, les données sont publiées sur le topic `/topic_dht11`.

Pour commencer, on initialise et on démarre le nœud :

```
1 # Demarrer le noeud
2 def main():
3     # Initialise le noeud
4     rospy.init_node('node_dht11')
5     serial_node = Node_dht11()
6     # Lance le noeud
7     rospy.spin()
8
9 if __name__ == '__main__':
10    main()
```

On initialise le type de capteur, le numéro de pin. On crée un topic publisher `/topic_dht11`, puis on démarre une fonction qui lit la valeur sur le capteur et publie sur le topic à intervalle régulier :

```
1 class Node_dht11:
2     def __init__(self):
3         self.sensor = Adafruit_DHT.DHT11
4         self.pin = 2
5
6         # Initialisation des publisher (nom du topic, type ,taille)
7         self.pub=rospy.Publisher('/topic_dht11', dht11, queue_size=10)
8
9         # Lancer une fonction a intervalle regulier (5 sec):
10        rospy.Timer(rospy.Duration(5), self.pub_donne_dht11)
11
12        # Ecrit dans le terminal pour debugage
13        rospy.loginfo("Demarrage du node pour DHT11")
```

Pour récupérer les données d'un capteur dht11, on a besoin de la librairie `Adafruit_DHT` et de cette fonction :

```

1 import Adafruit_DHT
2 humidity, temperature = Adafruit_DHT.read_retry(self.sensor, self.pin)

```

Pour publier les données sur le topic, on a créé un message personnalisé comme décrit précédemment de type dht11. On peut ainsi publier ces deux données sur le même topic :

```

1 # Publier sur un topic
2 def pub_donne_dht11(self, event):
3     # Lit les données du capteurs
4     humidity, temperature = Adafruit_DHT.read_retry(self.sensor, self.pin)
5
6     if humidity is not None and temperature is not None:
7         # Créer un message de type dht11 (message personnalisé) pour la température
8         # et l'humidité
9         msg_dht11=dht11()
10
11        # On enregistre les données acquises dans le message
12        msg_dht11.temperature=temperature
13        msg_dht11.humidity=humidity
14
15        # Publier les messages
16        self.pub.publish(msg_dht11)
17    else:
18        # Message pour debug en cas d'erreur
19        rospy.logwarn("Erreur de lecture du capteur DHT11")

```

Lecture du badge RFID

Ce nœud permet la lecture d'un badge RFID via le module **RC522**, qui communique en SPI avec la Raspberry Pi.

Fonctionnement :

1. Le lecteur détecte un badge RFID.
2. Il récupère l'UID du badge et le convertit en un entier.
3. L'UID est publié sur le topic /topic_rfid.

Pour commencer, on initialise et on démarre le nœud de la même manière que pour la température et l'humidité :

```

1 def main():
2     rospy.init_node('node_rfid', anonymous=True)
3     rfid_node = Node_RFID()
4     rfid_node.read_rfid()
5     rospy.spin()
6
7
8 if __name__ == '__main__':
9     main()

```

Ensuite, on initialise les ports GPIOs, le lecteur RFID et on crée un topic publisher pour publier l'uid du badge :

```

1 import RPi.GPIO as GPIO
2
3 class Node_RFID:
4     def __init__(self):
5         # Initialisation des GPIOs
6         GPIO.setmode(GPIO.BOARD)
7         GPIO.setwarnings(False)
8
9         # Initialisation du lecteur RFID
10        self.rc522 = RFID()
11
12        # Initialisation du publisher
13        self.pub_rfid = rospy.Publisher('/topic_rfid', Int32, queue_size=10)
14
15        rospy.loginfo("En attente d'un badge")

```

Pour détecter et lire un badge RFID, c'est la librairie pirc522 qui nous fournit les fonctions nécessaires :

```

1 from pirc522 import RFID
2
3 # Initialisation du lecteur RFID
4 rc522 = RFID()
5
6 # Attente d'un badge
7 rc522.wait_for_tag()
8 (error, tag_type) = rc522.request()
9
10 # Lire l'UID du badge
11 (error, uid) = rc522.anticoll()

```

Ainsi, notre code attend qu'un badge soit passé, et une fois qu'il est scanné, on récupère l'uid, on converti l'uid en entier et on le publie sur le topic avec un message de type Float32

```

1 def read_rfid(self, test_mode=False):
2     """Lit un badge RFID, s'arrete apres une lecture si test_mode=True"""
3
4     while not rospy.is_shutdown():
5         # Attend qu'un badge soit scanne
6         self.rc522.wait_for_tag()
7         (error, tag_type) = self.rc522.request()
8         if error:
9             rospy.logwarn("Erreur lors de la detection du badge RFID") # Ajout du
                log d'erreur
10
11         # Uniquement pour sortir de la boucle en cas de test unitaire
12         if test_mode:
13             return
14         # on recup l'uid
15         (error, uid) = self.rc522.anticoll()
16
17         if error:
18             rospy.logwarn("Erreur lors de la recuperation de l'UID du badge RFID")
                # Ajout du log d'erreur
19         if test_mode:
20             return
21
22         rfid_id = int(''.join(map(str, uid))) # Convertir l'UID en entier
23         rospy.loginfo(f"Badge detecte avec l'ID : {rfid_id}")
24
25         # Publier le message sur le topic
26         self.pub_rfid.publish(rfid_id)
27
28         if test_mode: # Sortir immediatement apres une seule lecture en mode test
29             return
30
31         time.sleep(1) # Evite une lecture trop rapide en boucle

```

Ecoute du micro

Le capteur sonore est analogique, mais la Raspberry Pi ne dispose pas d'entrée analogique. Pour contourner cette limitation, nous avons utilisé une **carte Arduino** reliée via le protocole de communication **Firmata** à la Raspberry Pi.

Fonctionnement :

1. L'Arduino lit la tension analogique sur A0.
2. La valeur est convertie en **décibels (dB)** grâce à une fonction logarithmique.
3. Le résultat est publié sur le topic `/topic_micro` toutes les 0.5 secondes.

Comme pour tous les autres noeuds, la première étape est d'initialiser le noeud et de le démarrer.

```

1 # Demarrer le noeud
2 def main():
3     # Initialise le noeud
4     rospy.init_node('node_micro')
5     serial_node = Node_micro()
6     # Lance le noeud
7     rospy.spin()
8
9 if __name__ == '__main__':
10     main()

```

Ensuite, on initialise le port USB sur lequel est branchée la carte arduino, on crée un topic publisher pour publier un message de type Float32 contenant la valeur en décibel du son relevé par le micro.

```

1 class Node_micro:
2     def __init__(self):
3         # Définir le port serie de l'Arduino
4         self.board = Arduino('/dev/ttyUSB0') # Remplacez '/dev/ttyUSB0' par le port
          approprié
5         it = util.Iterator(self.board)
6         it.start()
7
8         self.board.analog[0].enable_reporting()
9
10        # Initialisation des publisher (nom du topic, type, taille)
11        self.topic_micro = rospy.Publisher('/topic_micro', Float32, queue_size=10)
12
13        # Lancer une fonction a intervalle regulier
14        rospy.Timer(rospy.Duration(0.5), self.pub_donne_micro)
15
16        # Ecrit dans le terminal
17        rospy.loginfo("Publication micro")

```

Maintenant que tout est initialisé, nous pouvons lire la valeur du micro. Pour ce faire, on utilise la librairie pyfirmata pour lire la donnée du micro sur la carte Arduino :

```

1 from pyfirmata import Arduino, util
2
3 # Définir le port serie de l'Arduino
4 self.board = Arduino('/dev/ttyUSB0') # Remplacez '/dev/ttyUSB0' par le port
          approprié
5
6 # Cree un iterateur qui surveille en arriere-plan les donnees recues de l'Arduino.
7 it = util.Iterator(self.board)
8
9 # Demarre l'iterateur pour permettre a PyFirmata de recuperer les valeurs
          analogiques.
10 it.start()
11
12 # Active la lecture continue de la broche A0 (entree analogique 0).
13 self.board.analog[0].enable_reporting()
14
15 # Recupere la valeur analogique lue sur la broche A0.
16 analog_value = self.board.analog[0].read()

```

Voici plus en détail comment on utilise ces fonctions :

Pour commencer, on lit la valeur analogique lue sur la broche A0, ensuite on convertit cette valeur en tension de 0 à 5V. Ce qu'il faut savoir, c'est qu'avec Firmata, la valeur lue est comprise entre 0 et 1. Donc on multiplie par 5 pour revenir à une valeur entre 0 et 5V. Ensuite, on convertit cette valeur en décibel avant de publier un message de type Float32 sur le topic.

```
1 # Convertir les valeurs en decibel
2 def voltage_to_db(self, voltage, v_ref=1.0):
3     if voltage > 0 :
4         return 20*math.log10(voltage/v_ref)
5     else :
6         return -float('inf')
7
8 # Publier sur un topic
9 def pub_donne_micro(self, event):
10     analog_value = self.board.analog[0].read() # Lire la valeur analogique de la
11         broche A0
12
13     if analog_value is None:
14         rospy.logwarn("Valeur analogique None, aucune publication") # Ajout d'un
15         warning
16         return # Arrete l'execution pour eviter une publication
17
18     # Conversion de la valeur analogique en tension (de 0 a 5V)
19     voltage = analog_value * 5.0
20     db_value = self.voltage_to_db(voltage, 1)
21     # Initialisation d'un message de type Float32
22     msg = Float32()
23     # On enregistre la valeur dans le message
24     msg.data = db_value
25     # On publie le message sur le topic
26     self.topic_micro.publish(msg)
```

Code pour les serveurs de service ROS

Dans notre projet, nous utilisons les services ROS pour la gestions du badge dans la base de données. Nous avons trois services :

1. Le service ROS `ajout_badge` permet d'ajouter un badge RFID dans la base de données.
2. Le service `del_badge` permet de supprimer un badge de la base de données.
3. Le service `login_serv` permet de récupérer les informations d'un utilisateur enregistré à partir de son nom d'utilisateur.

Dans cette partie, nous allons décrire le fonctionnement du service ROS.

Un service ROS fonctionne selon un modèle client-serveur. Contrairement aux topics, qui diffusent des messages en continu, un service permet d'exécuter une action ponctuelle avec une requête et une réponse.

Le serveur de service (ROS Service Server) est un nœud qui attend des requêtes et exécute une action lorsque le client envoie une demande.

Le client de service (ROS Service Client) est un nœud qui envoie une requête au serveur et attend une réponse.

Le serveur exécute la tâche demandée (ajout d'un badge, suppression, etc.) et ensuite il envoie une réponse au client avec le résultat de l'opération.

Voici un exemple montrant comment mettre en place un service ROS :

Côté serveur de service :

```
1 import rospy
2 from my_package.srv import MonService, MonServiceResponse
3
4 # Fonction qui execute l'action et renvoie la reponse au client
5 def handle_request(req):
6     return MonServiceResponse("Reponse du serveur")
7
8 # Initialisation du nœud du serveur de service
9 rospy.init_node('serveur')
10 # Initialisation du service
11 service = rospy.Service('mon_service', MonService, handle_request)
12 rospy.spin()
```

Côté client de service :

```
1 import rospy
2 from my_package.srv import MonService
3
4 # On attend que le serveur de service soit disponible
5 rospy.wait_for_service('mon_service')
6 # On initialise le client de service
7 client = rospy.ServiceProxy('mon_service', MonService)
8 # On envoie la requete et on attend la reponse
9 response = client()
```

Ainsi, un client demande une action et le serveur l'exécute avant de renvoyer une réponse.

Le service pour ajouter un badge

Sur notre application web, un administrateur peut ajouter un nouveau badge si nécessaire. Pour cela, il doit d'abord placer le badge sur le lecteur RFID, puis saisir les informations de l'utilisateur associé, telles que le nom, le prénom, le nom d'utilisateur, l'âge, l'email, le mot de passe et son rôle dans l'entreprise. Une fois ces données renseignées, notre code les collecte et les envoie à un service ROS de la manière suivante :

```
1 # Service ROS pour ajouter un badge
2 def send_user_info(prenom, nom, username, age, email, mdp, job_title):
3     # En attente de trouver le service ROS ajout_badge
4     rospy.wait_for_service('ajout_badge')
5     try:
6         # On initialise le service
7         ajout_badge_service = rospy.ServiceProxy('ajout_badge', ajout_badge)
8         # On envoie les donnees sur le service et on attend de recevoir la reponse
9         response = ajout_badge_service(prenom, nom, username, age, email, mdp,
10                                         job_title)
11         # On affiche le resultat de la reponse. Si success est a True alors les
12         # donnees ont bien ete ajoute
13         rospy.loginfo(f"Service response: success = {response.success}")
14         return response.success
15     except rospy.ServiceException as e:
16         rospy.logerr(f"Service call failed: {e}")
17         return False
```

Côté serveur de service, on reçoit la requête et on ajoute à la base de données. Pour commencer, on initialise le noeud, le topic subscriber se chargeant de lire la valeur de l'uid du badge scanné et on initialise le serveur de service.

```
1 def listener():
2     """Initialise le noeud ROS et les services."""
3     rospy.init_node('ajout_badge', anonymous=True)
4
5     # Creer la base de donnees si elle n'existe pas
6     create_database_infos()
7
8     # Souscrire au topic pour recupere le numero du badge
9     rospy.Subscriber("topic_rfid", Int32, lecture_badge)
10
11     # Initialisation du serveur de service
12     rospy.Service('ajout_badge', ajout_badge, ajout_badge_base)
13
14     rospy.loginfo("Noeud ajout_badge demarre et en attente de messages...")
15     rospy.spin()
16
17 if __name__ == '__main__':
18     try:
19         listener()
20     except rospy.ROSInterruptException:
21         rospy.loginfo("Noeud ajout_badge arrete.")
```

Les infos sont envoyées dans la fonction "ajout_badge_base" dont son paramètre "req" contient toutes les infos de la requête. Les données sont ensuite ajoutées à la database et pour finir, le serveur renvoie une réponse "success" à True si l'opération a réussi ou "False" si l'opération a échoué.


```

1 def ajout_badge_base(req):
2     """Ajoute les informations du badge dans la base de donnees."""
3     global global_badge
4     rospy.loginfo(f"DEBUG: Valeur actuelle de global_badge: {global_badge}")
5
6     if global_badge is None:
7         rospy.logerr("Aucun badge detecte !")
8         return ajout_badgeResponse(False)
9
10    rospy.loginfo(f"Infos utilisateur recues : {req.prenom} {req.nom},username:{req.
11        username}, Age: {req.age}, mail: {req.mail}, MDP : {req.password}, Role: {
12        req.poste}")
13
14    # Suppression du hachage pour eviter l'erreur dans le test
15    hashed_password = generate_password_hash(req.password, method='pbkdf2:sha256')
16
17    # Enregistrement dans la base de donnees
18    conn = sqlite3.connect(db_path)
19    cursor = conn.cursor()
20    try:
21        rospy.loginfo("DEBUG: Insertion en cours dans la base de donnees...")
22        cursor.execute('''
23            INSERT INTO infos (numBadge,user, prenom, nom, age, mail, mdp, poste)
24            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
25            ''', (global_badge, req.username, req.prenom, req.nom, req.age, req.mail,
26                hashed_password, req.poste))
27
28        rospy.loginfo(f"Badge {global_badge} enregistre avec succes")
29        success = True
30    except sqlite3.Error as e:
31        rospy.logerr(f"Erreur lors de l'insertion des donnees : {e}")
32        success = False
33    finally:
34        conn.commit()
35        conn.close()
36
37    return ajout_badgeResponse(success)

```

Le service pour supprimer un badge

De la même manière que pour ajouter un badge, un administrateur peut également supprimer un badge. Pour ce faire, un service ROS est utilisé. Cette fois-ci, on envoie simplement une requête à True si on veut supprimer un badge, le serveur se charge de lire le numéro de badge scanné et de le supprimer de la database.

```
1 # Service ROS pour ajouter un badge
2 def del_badge_serv():
3     try:
4         rospy.wait_for_service('del_badge', timeout=2) # Timeout de 2 secondes
5     except rospy.ROSException:
6         rospy.logerr("Timeout : Service 'del_badge' non disponible.")
7         return False
8
9     rospy.loginfo("Request Del badge")
10    try:
11        del_badge_service = rospy.ServiceProxy('del_badge', suppr_badge)
12        response = del_badge_service(True)
13        rospy.loginfo(f"Service response: success = {response.validation}")
14        return response.validation
15    except rospy.ServiceException as e:
16        rospy.logerr(f"Service call failed: {e}")
17        return False
```

Coté serveur, on se charge de lire le numéro de badge à supprimer et de le supprimer si la requête envoie True. Pour commencer, comme pour ajouter un badge on initialise le noeud, le subscriber et le service :

```
1 def listener():
2     """Initialise le noeud ROS et les services."""
3     rospy.init_node('Del_badge', anonymous=True)
4     rospy.Subscriber("topic_rfid", Int32, lecture_badge)
5     rospy.Service('del_badge', suppr_badge, del_badge_base)
6
7     rospy.loginfo("Noeud Del_badge démarre et en attente d'instructions...")
8     rospy.spin()
9
10 if __name__ == '__main__':
11     try:
12         listener()
13     except rospy.ROSInterruptException:
14         rospy.loginfo("Noeud Del_badge arrete.")
```

Ensuite on analyse la requête et on renvoie la réponse au client en cas de succès ou d'échec :

```
1 def del_badge_base(req):
2     """Supprime un badge de la base de donnees."""
3     global global_badge
4     sys.stdout.flush()
5
6     if global_badge is None:
7         rospy.logwarn("Aucun badge reçu.")
8         sys.stdout.flush()
9         return suppr_badgeResponse(validation=False)
10
11     conn = None # Initialise conn pour eviter l'erreur
12
13     try:
14         conn = sqlite3.connect(db_path)
15         cursor = conn.cursor()
16
17         sys.stdout.flush()
18
19         cursor.execute("DELETE FROM infos WHERE numBadge = ?", (global_badge,))
20         conn.commit()
21
22         sys.stdout.flush()
23
24         if conn.total_changes > 0:
25             rospy.loginfo(f"Badge {global_badge} supprime.")
26             sys.stdout.flush()
27             return suppr_badgeResponse(validation=True)
28         else:
29             rospy.logwarn(f"Aucune entree pour le badge : {global_badge}")
30             sys.stdout.flush()
31             return suppr_badgeResponse(validation=False)
32
33     except sqlite3.Error as e:
34         rospy.logerr(f"Erreur SQLite : {e}")
35         sys.stdout.flush()
36         return suppr_badgeResponse(validation=False)
37
38     finally:
39         if conn: # Verifie que la connexion a bien ete creee avant de la fermer
40             conn.close()
```

Le service pour récupérer un identifiant de la database

Quand un utilisateur souhaite se connecter, il entre son nom d'utilisateur et son mot de passe. Pour valider la connexion, on a besoin de récupérer ces infos de la database pour comparer le mot de passe qu'il a entré associé à son nom d'utilisateur. C'est pourquoi on utilise de nouveau un service pour récupérer ces informations. On envoie une requête contenant le nom d'utilisateur, le serveur se charge d'aller récupérer son mot de passe et son rôle attribué dans l'entreprise ainsi on peut par la suite comparer le mot de passe qu'il a tapé et lui définir quels accès il a le droit d'avoir sur le site Web.

```
1 @login_manager.user_loader
2 def load_user(user_id):
3     if user_id in users_cache:
4         return users_cache[user_id] # Retourne l'utilisateur en cache sans appeler
5         ROS
6
7     rospy.loginfo(f"Trying to load user with ID: {user_id}")
8
9     rospy.wait_for_service('login_serv')
10    try:
11        login_service = rospy.ServiceProxy('login_serv', login_member)
12        response = login_service(user_id)
13
14        if response.success:
15            user = User(id=response.id, username=response.username, password=
16                response.password, role=response.role)
17            users_cache[user_id] = user # Stocke l'utilisateur en cache
18            return user
19        else:
20            return None
21    except rospy.ServiceException as e:
22        rospy.logerr(f"Service call failed: {e}")
23        return None
```

Côté serveur, je passe le côté initialisation car c'est exactement la même manière que pour les deux précédents services, cependant on analyse la requête de la même manière et on renvoie les infos demandées.

```
1 def extract_data(req):
2     username = req.username
3
4     conn = sqlite3.connect(db_path)
5     cursor = conn.cursor()
6     try:
7         # Correction : utilisation de la variable username dans la requete SQL
8         cursor.execute('''SELECT id, mdp, poste FROM infos WHERE user = ?;''', (
9             username,))
10        data_extracted = cursor.fetchone() # Stockage des donnees dans une variable
11
12        if data_extracted:
13            id_, password, role = data_extracted # Separation des informations dans
14            des variables distinctes
15
16            rospy.loginfo("Donnees extraites avec succes")
17            success = True
18        else:
19            rospy.logwarn(f"Aucune donnee trouvee pour l'utilisateur {username}")
20            success = False
21            id_ = None
22            username=None
23            role =None
24            password=None
25
26        except sqlite3.Error as e:
27            rospy.logerr(f"Erreur lors de l'extraction des donnees : {e}")
28            success = False
29            id_ = None
30            username=None
31            role =None
32            password=None
33
34        finally:
35            conn.commit()
36            conn.close()
37
38        return login_memberResponse(success,id_, username, password, role) # Retour des
39        resultats pour le service ROS
```

VIII. Tests et résultats

Dans cette section, nous présentons les tests unitaires et d'intégration réalisés afin de garantir le bon fonctionnement des différents nœuds de notre projet ROS. Les tests couvrent les aspects critiques de notre système, notamment :

- L'acquisition des données depuis les capteurs.
- Le traitement et l'analyse des données collectées.
- L'intégration avec la base de données.
- La communication entre les différents nœuds.

Pour ce faire, nous avons utilisé `unittest` et `rosunit` pour écrire et exécuter les tests unitaires, en nous assurant de couvrir plusieurs scénarios :

- **Cas normaux** : vérification du bon fonctionnement en conditions standard.
- **Cas limites** : tests sur des valeurs extrêmes ou inhabituelles.
- **Cas d'erreur** : simulation de défaillances pour tester la robustesse du code.

Tous les tests ont été sauvegardés dans un dossier `test` propre à chaque package, et un fichier de test a été écrit pour chaque nœud.

Tests unitaires dans le package `sensors`

Tests du nœud `badge_sensor_node.py`

Le nœud `badge_sensor_node.py` est chargé de lire les badges RFID et de publier leur identifiant sur le topic `/topic_rfid`. Pour valider son fonctionnement, nous avons écrit un fichier de test `test_badge.py` qui couvre les cas suivants :

- Lecture et publication correcte d'un badge RFID.
- Gestion d'une erreur de lecture du badge.

Le test est basé sur l'utilisation de `unittest` et `rosunit`, ainsi que de `unittest.mock` pour simuler le capteur RFID. La structure du test est la suivante :

- **Initialisation** : création d'un mock pour le module RFID et lancement du nœud en test.
- **Test de lecture réussie** : simulation d'un badge détecté et vérification que l'ID est correctement publié.
- **Test de gestion d'erreur** : simulation d'une erreur et vérification que le système gère le cas correctement.

Le test principal est décrit dans le code suivant :

```

1 @patch('badge_sensor_node.RFID')
2 def setUp(self, mock_rfid):
3     rospy.init_node('test_rfid', anonymous=True)
4     self.mock_rfid_instance = mock_rfid.return_value
5     self.mock_rfid_instance.request.return_value = (False, "TAG_TYPE")
6     self.mock_rfid_instance.anticoll.return_value = (False, [12, 34, 56, 78, 90])
7     self.node = Node_RFID()
8     self.received_rfid = None
9     self.sub_rfid = rospy.Subscriber('/topic_rfid', Int32, self.callback_rfid)
10    rospy.sleep(1)

```

Le test de lecture d'un badge valide est réalisé avec :

```

1 def test_rfid_read_success(self):
2     self.node.read_rfid(test_mode=True)
3     timeout = time.time() + 5
4     while self.received_rfid is None and time.time() < timeout:
5         rospy.sleep(0.1)
6     self.assertIsNotNone(self.received_rfid, "L'ID RFID n'a pas été publié")
7     self.assertEqual(self.received_rfid, 1234567890)

```

Enfin, un test d'erreur est ajouté pour vérifier la gestion des défaillances du capteur :

```

1 def test_rfid_read_error(self):
2     self.mock_rfid_instance.request.return_value = (True, None)
3     with self.assertLogs(level="INFO") as log:
4         self.node.read_rfid(test_mode=True)
5     self.assertTrue(any("Badge détecté" not in message for message in log.output))

```

Tous les tests ont été exécutés avec succès, validant ainsi le bon fonctionnement du nœud RFID.

```

root@41584ca7261b:~/Projet_it# rosrn sensors test_badge.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/sensors/rosunit-test_badge.xml
[Testcase: test_rfid_read_error] ... ok
[Testcase: test_rfid_read_success] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 2
* ERRORS: 0 []
* FAILURES: 0 []

```

FIGURE 22 – Résultat du test du nœud lisant le badge

Test du nœud DHT11

Le nœud `dht11_sensor_node.py` est responsable de l'acquisition des données de température et d'humidité depuis un capteur DHT11, puis de leur publication sur le topic `/topic_dht11` sous forme d'un message personnalisé `sensors/msg/dht11`. Pour vérifier son bon fonctionnement, nous avons mis en place un fichier de test `test_dht11.py` qui couvre les scénarios suivants :

- Publication correcte des valeurs de température et d'humidité.
- Gestion des erreurs lorsque le capteur ne fournit pas de données valides.

Le test repose sur `unittest`, `rosunit` et l'utilisation de `unittest.mock` pour simuler le comportement du capteur. Voici l'initialisation du test :

```

1 def setUp(self):
2     rospy.init_node('test_dht11', anonymous=True)
3     self.node = Node_dht11()
4     self.received_msg = None
5     self.sub_dht11 = rospy.Subscriber('/topic_dht11', dht11, self.callback_dht11)
6
7 def callback_dht11(self, msg):
8     self.received_msg = msg

```

Le test de publication de données correctes est réalisé comme suit :

```

1 @patch('Adafruit_DHT.read_retry', return_value=(60.0, 22.5))
2 def test_pub_donne_dht11(self, mock_dht):
3     self.node.pub_donne_dht11(event=None)
4     timeout = time.time() + 5
5     while self.received_msg is None and time.time() < timeout:
6         rospy.sleep(0.1)
7     self.assertIsNotNone(self.received_msg, "Le message DHT11 n'a pas ete recu")
8     self.assertEqual(self.received_msg.temperature, 22.5)
9     self.assertEqual(self.received_msg.humidity, 60.0)

```

Enfin, un test d'erreur est implémenté pour vérifier la gestion des lectures défaillantes du capteur :

```

1 @patch('Adafruit_DHT.read_retry', return_value=(None, None))
2 def test_pub_donne_dht11_erreur(self, mock_dht):
3     with self.assertLogs(level="WARN") as log:
4         self.node.pub_donne_dht11(event=None)
5     self.assertTrue(any("Erreur de lecture du capteur DHT11" in message for message
6                         in log.output))

```

Tous les tests ont été exécutés avec succès, confirmant la bonne gestion des mesures et des erreurs pour le capteur DHT11.

```

root@41584ca7261b:~/Projet_it# rosrund sensors test_dht11.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/sensors/rosunit-test_dht11.xml
[Testcase: test_pub_donne_dht11] ... ok
[Testcase: test_pub_donne_dht11_erreur] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 2
* ERRORS: 0 []
* FAILURES: 0 []

```

FIGURE 23 – Résultat du test du noeud DHT11

Test du noeud micro

Le noeud `micro_sensor_node.py` est chargé d'acquérir les niveaux sonores à partir d'un capteur analogique via une carte Arduino, puis de les publier sous forme de valeurs en décibels sur le topic `/topic_micro`. Un fichier de test `test_micro.py` a été écrit pour vérifier les éléments suivants :

- La conversion correcte d'une tension en décibels.
- La publication correcte des niveaux sonores sur le topic.
- La gestion des erreurs en cas de lecture invalide du capteur.

L'initialisation du test se fait en simulant une carte Arduino et en lançant le noeud sous test :


```

1 @patch('micro_sensor_node.Arduino')
2 def setUp(self, mock_arduino):
3     rospy.init_node('test_micro', anonymous=True)
4     self.mock_arduino_instance = mock_arduino.return_value
5     self.mock_analog_pin = MagicMock()
6     self.mock_arduino_instance.analog = [self.mock_analog_pin]
7     self.mock_analog_pin.read.return_value = 0.5
8     self.node = Node_micro()
9     self.received_msg = None
10    self.sub_micro = rospy.Subscriber('/topic_micro', Float32, self.callback_micro)
11    rospy.sleep(1)

```

Un test de conversion tension -> dB est défini comme suit :

```

1 def test_voltage_to_db(self):
2     self.assertAlmostEqual(self.node.voltage_to_db(1.0, 1.0), 0.0) # 1V / 1V -> 0
3     self.assertAlmostEqual(self.node.voltage_to_db(0.5, 1.0), -6.02, places=2) #
4     self.assertAlmostEqual(self.node.voltage_to_db(2.0, 1.0), 6.02, places=2) # 2V
5     self.assertEqual(self.node.voltage_to_db(0, 1.0), -float('inf')) # Cas ou
    voltage = 0

```

Le test de publication du niveau sonore s'écrit ainsi :

```

1 def test_pub_donne_micro(self):
2     self.node.pub_donne_micro(event=None)
3     timeout = time.time() + 5
4     while self.received_msg is None and time.time() < timeout:
5         rospy.sleep(0.1)
6     expected_db_value = self.node.voltage_to_db(2.5, 1.0)
7     self.assertIsNotNone(self.received_msg, "La valeur n'a pas ete publiee")
8     self.assertAlmostEqual(self.received_msg, expected_db_value, places=2)

```

Enfin, un test vérifie la gestion des erreurs lorsque la valeur analogique est absente :

```

1 def test_pub_donne_micro_none(self):
2     self.mock_analog_pin.read.return_value = None
3     with self.assertLogs(level="WARN") as log:
4         self.node.pub_donne_micro(event=None)
5     self.assertTrue(any("Valeur analogique None, aucune publication" in message for
        message in log.output))

```

Ces tests ont confirmé que le nœud microphone fonctionne correctement et gère bien les valeurs incorrectes.

```

root@41584ca7261b:~/Projet_it# rosrund sensors test_micro.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/sensors/rosunit-test_micro.xml
[Testcase: test_pub_donne_micro] ... ok
[Testcase: test_pub_donne_micro_none] ... ok
[Testcase: test_voltage_to_db] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 3
* ERRORS: 0 []
* FAILURES: 0 []

```

FIGURE 24 – Résultat du test du nœud écoutant le micro

Tests des nœuds du package database

Les nœuds du package `database` sont responsables de l'interaction avec les bases de données pour stocker et récupérer les mesures issues des capteurs et des badges RFID. Pour s'assurer de leur bon fonctionnement, nous avons mis en place des tests unitaires visant à vérifier :

- La création correcte des bases de données SQLite.
- L'insertion correcte des mesures dans les bases de données.
- La récupération et le traitement corrects des données.
- L'intégration correcte des callbacks recevant les messages des capteurs.

Les tests suivants ont été réalisés.

Test de la base de données d'humidité

Le fichier de test `test_database_humd.py` vérifie les opérations de stockage des mesures d'humidité collectées par le capteur DHT11.

Les tests effectués incluent :

- **Création de la base de données** : Vérifie que la table pour l'humidité est correctement créée.
- **Insertion d'une mesure** : Simule l'insertion d'une humidité et vérifie l'exécution correcte de la requête.
- **Réception des données via ROS** : Teste que le callback récupère bien l'humidité reçue depuis le topic `/topic_dht11` et l'insère correctement dans la base.

Exemple de test de création de la base :

```
1 @patch('database_humdht11.sqlite3.connect')
2 def test_create_database(self, mock_sqlite):
3     mock_conn = mock_sqlite.return_value
4     mock_cursor = mock_conn.cursor.return_value
5     create_database()
6     mock_cursor.execute.assert_called()
7     mock_conn.commit.assert_called_once()
8     mock_conn.close.assert_called_once()
```

```
root@41584ca7261b:~/Projet_it# rosrn database test_database_humd.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/sensors/rosunit-test_humidite_listener.xml
[Testcase: test_create_database] ... ok
[Testcase: test_hum_callback] ... ok
[Testcase: test_insert_measurement] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 3
* ERRORS: 0 []
* FAILURES: 0 []
```

FIGURE 25 – Résultat du test du noeud database humidite

Test de la base de données sonore

Le fichier `test_database_micro.py` teste le stockage des niveaux sonores capturés. Il s'assure que :

- La base de données est bien créée.

- Les mesures sont correctement enregistrées.
 - Le callback `sound_callback` récupère bien les données du topic `/topic_micro`.
- Exemple de test de réception d'un message ROS et insertion dans la base :

```

1 @patch('database_sonore.insert_measurement')
2 def test_sound_callback(self, mock_insert):
3     msg = Float32()
4     msg.data = 80.0 # Simuler un volume sonore de 80 dB
5     sound_callback(msg)
6     mock_insert.assert_called_with(80.0)

```

```

root@41584ca/261b:~/Projet_it# rosrn database test_database_micro.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/database/rosunit-test_sound_listener.xml
[Testcase: test_create_database] ... ok
[Testcase: test_insert_measurement] ... ok
[Testcase: test_sound_callback] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 3
* ERRORS: 0 []
* FAILURES: 0 []

```

FIGURE 26 – Résultat du test du noeud de database du micro

Test de la base de données RFID

Le fichier `test_database_rfid_infos.py` teste les interactions avec la base de données contenant les badges RFID enregistrés. Les tests portent sur :

- La création correcte de la base.
- La vérification de l'existence d'un badge.
- La gestion correcte des badges inconnus.
- L'intégration avec le callback ROS du nœud.

Test de la vérification de l'existence d'un badge :

```

1 @patch('database_RFID.sqlite3.connect')
2 def test_check_badge_in_infos(self, mock_sqlite):
3     mock_conn = mock_sqlite.return_value
4     mock_cursor = mock_conn.cursor.return_value
5
6     # Simuler un badge existant
7     mock_cursor.fetchone.return_value = (123456, "John Doe", "Engineer")
8     result = check_badge_in_infos(123456)
9     self.assertTrue(result)
10
11     # Simuler un badge inexistant
12     mock_cursor.fetchone.return_value = None
13     result = check_badge_in_infos(999999)
14     self.assertFalse(result)

```

```

root@41584ca7261b:~/Projet_it# rosrund database test_database_rfid_info.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/database/rosunit-test_rfid_listener.xml
[Testcase: test_check_badge_in_infos] ... ok
[Testcase: test_create_database_mesure] ... ok
[Testcase: test_rfid_callback_known_badge] ... ok
[Testcase: test_rfid_callback_unknown_badge] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 4
* ERRORS: 0 []
* FAILURES: 0 []

```

FIGURE 27 – Résultat du test du noeud de database des infos du badge

Test de la base de données de température

Le fichier `test_database_temp.py` teste les fonctionnalités liées à l'enregistrement des températures.

Les tests effectués :

- Vérification de la création de la base de données.
- Test de l'insertion des mesures.
- Vérification de la bonne réception des données via ROS et leur stockage.

Exemple de test sur la réception d'une mesure et son enregistrement :

```

1 @patch('database_tempdht11.insert_measurement')
2 def test_temperature_callback(self, mock_insert):
3     msg = dht11()
4     msg.temperature = 25.0 # Simuler une temperature de 25 degre
5     temperature_callback(msg)
6     mock_insert.assert_called_with(25.0)

```

```

root@41584ca7261b:~/Projet_it# rosrund database test_database_temp.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/database/rosunit-test_temperature_listener.xml
[Testcase: test_create_database] ... ok
[Testcase: test_insert_measurement] ... ok
[Testcase: test_temperature_callback] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 3
* ERRORS: 0 []
* FAILURES: 0 []

```

FIGURE 28 – Résultat du test du noeud de database de la température

Conclusion des tests des bases de données

Tous les tests ont été exécutés avec succès, confirmant que :

- Les bases de données sont correctement créées et gérées.
- Les mesures issues des capteurs sont bien stockées et accessibles.
- Les callbacks ROS fonctionnent bien et traitent correctement les messages reçus.

Ces tests garantissent une intégration fluide entre les modules ROS et le stockage des données.

Tests des nœuds du package badge_rfid

Le package `badge_rfid` est responsable de la gestion des badges RFID, notamment leur ajout et leur suppression dans la base de données. Les tests effectués dans cette section s'assurent que :

- L'ajout d'un badge fonctionne correctement.
 - La suppression d'un badge est bien prise en compte.
 - Les erreurs de base de données sont correctement gérées.
 - Les services interagissent correctement avec les topics et la base de données.
- Les fichiers de tests suivants ont été écrits.

Test du service d'ajout de badge

Le fichier `test_ajout_badge.py` teste le fonctionnement du service `ajout_badge.srv`. Il couvre les cas suivants :

- La création correcte de la base de données des badges.
- La bonne réception d'un numéro de badge depuis un topic ROS.
- L'ajout réussi d'un badge dans la base.
- La gestion du cas où aucun badge n'a été scanné.

Exemple de test de création de la base :

```
1 @patch('ajout_badge.sqlite3.connect')
2 def test_create_database_infos(self, mock_sqlite_connect):
3     mock_conn = MagicMock()
4     mock_cursor = MagicMock()
5     mock_sqlite_connect.return_value = mock_conn
6     mock_conn.cursor.return_value = mock_cursor
7
8     ajout_badge.create_database_infos()
9
10    mock_cursor.execute.assert_called()
11    mock_conn.commit.assert_called()
12    mock_conn.close.assert_called()
```

Test de lecture d'un badge et mise à jour de la variable globale :

```
1 def test_lecture_badge(self):
2     ajout_badge.global_badge = None
3     msg = Int32()
4     msg.data = 123456
5     ajout_badge.lecture_badge(msg)
6
7     self.assertEqual(ajout_badge.global_badge, 123456, "Le numero de badge n'a pas
    ete correctement mis a jour.")
```

Test d'ajout d'un badge avec succès :
Le mot de passe étant haché dans la base de données, il faut le traiter séparément.

```
1 @patch('ajout_badge.sqlite3.connect')
2 def test_ajout_badge_base_success(self, mock_sqlite_connect):
3     mock_conn = mock_sqlite_connect.return_value
4     mock_cursor = mock_conn.cursor.return_value
5
6     ajout_badge.global_badge = 123456 # Assurer que le badge est bien defini
7     print(f"DEBUG TEST - Valeur de global_badge juste avant appel: {ajout_badge.
8         global_badge}") # Debug
9
10    response = ajout_badge.ajout_badge_base(self.mock_service_req) #
11
12    # Verifier que execute() a bien ete appele
13    mock_cursor.execute.assert_called()
14
15    # Verifier si la requete est correcte
16    expected_sql = "INSERT INTO infos (numBadge,user, prenom, nom, age, mail, mdp,
17        poste) VALUES (?, ?, ?, ?, ?, ?, ?, ?)"
18    actual_sql, actual_values = mock_cursor.execute.call_args[0] # Recupere 1'
19        argument SQL
20
21    # Normalisation des espaces et suppression des sauts de ligne
22    actual_sql = " ".join(actual_sql.split())
23
24    self.assertEqual(expected_sql, actual_sql, "La requete SQL executee ne
25        correspond pas a celle attendue.")
26
27    # Recuperer le mot de passe reellement insere
28    actual_hashed_password = actual_values[6] # Position du mdp dans le tuple
29    expected_password = "securepassword"
30
31    # Verifier que le mot de passe hache correspond bien au mot de passe d'
32        origine
33    self.assertTrue(check_password_hash(actual_hashed_password, expected_password),
34        "Le mot de passe stocke ne correspond pas au mot de passe
35        attendu.")
36
37    # Verifier que les autres valeurs sont correctes
38    self.assertEqual(
39        actual_values[:6] + actual_values[7:], # Exclut le mot de passe
40        (123456, "johndoe", "John", "Doe", 25, "john.doe@example.com", "Ingenieur"),
41        "Les autres valeurs inserees ne correspondent pas."
42    )
43
44    # Verifier que commit et close sont appeles
45    mock_conn.commit.assert_called_once()
46    mock_conn.close.assert_called_once()
47
48    # Verifier que la reponse est bien un succes
49    self.assertTrue(response.success)
```

Test d'ajout échoué sans badge scanné :

```
1 def test_ajout_badge_base_failure_no_badge(self):
2     ajout_badge.global_badge = None
3     response = ajout_badge.ajout_badge_base(self.mock_service_req)
4     self.assertFalse(response.success)
```

```
root@41584ca7261b:~/Projet_it# rosrun badge_rfid test_ajout_badge.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/badge_rfid/rosunit-test_ajout_badge.xml
[Testcase: test_ajout_badge_base_failure_no_badge] ... ok
[Testcase: test_ajout_badge_base_success] ... ok
[Testcase: test_create_database_infos] ... ok
[Testcase: test_lecture_badge] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 4
* ERRORS: 0 []
* FAILURES: 0 []
```

FIGURE 29 – Résultat du test du noeud d'ajout de badge

Tous les tests ont été exécutés avec succès, confirmant que l'ajout d'un badge est bien pris en charge et que les erreurs sont correctement gérées.

Test du service de suppression de badge

Le fichier `test_del_badge.py` teste le service `suppr_badge.srv` et couvre les cas suivants :

- La bonne réception du numéro de badge depuis le topic.
- La suppression correcte d'un badge existant.
- La gestion du cas où le badge n'existe pas en base.
- La gestion des erreurs SQL lors de la suppression.

Exemple de test de lecture d'un badge :

```
1 def test_lecture_badge(self):
2     delete_badge.global_badge = None
3     msg = Int32()
4     msg.data = 123456
5     delete_badge.lecture_badge(msg)
6
7     self.assertEqual(delete_badge.global_badge, 123456, "Le numero de badge n'a pas
    ete correctement mis a jour")
```

Test de suppression réussie d'un badge existant :

```
1 @patch('delete_badge.sqlite3.connect')
2 def test_del_badge_base_success(self, mock_sqlite):
3     delete_badge.global_badge = 123456
4     mock_conn = mock_sqlite.return_value
5     mock_cursor = mock_conn.cursor.return_value
6     mock_conn.total_changes = 1
7
8     req = suppr_badgeRequest()
9     response = delete_badge.del_badge_base(req)
10
11     self.assertTrue(response.validation, "Le badge aurait du etre supprime")
12     mock_cursor.execute.assert_called_with("DELETE FROM infos WHERE numBadge = ?", (
13         delete_badge.global_badge,))
14     mock_conn.commit.assert_called_once()
15     mock_conn.close.assert_called_once()
```

Test de suppression d'un badge inexistant :

```
1 @patch('delete_badge.sqlite3.connect')
2 def test_del_badge_base_not_found(self, mock_sqlite):
3     delete_badge.global_badge = 999999
4     mock_conn = mock_sqlite.return_value
5     mock_cursor = mock_conn.cursor.return_value
6     mock_conn.total_changes = 0
7
8     req = suppr_badgeRequest()
9     response = delete_badge.del_badge_base(req)
10
11     self.assertFalse(response.validation, "Le service aurait du renvoyer False car
12         le badge n'existe pas")
```

Test de gestion d'une erreur SQL lors de la suppression :

```
1 @patch('delete_badge.sqlite3.connect', side_effect=sqlite3.Error("Erreur SQL"))
2 def test_del_badge_base_sql_error(self, mock_sqlite):
3     delete_badge.global_badge = 123456
4     req = suppr_badgeRequest()
5     response = delete_badge.del_badge_base(req)
6
7     self.assertFalse(response.validation, "Le service aurait du renvoyer False en
8         cas d'erreur SQL")
```

```
root@41584ca7261b:~/Projet_it# rosrun badge_rfid test_del_badge.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/badge_rfid/rosunit-test_del_badge.xml
[Testcase: test_del_badge_base_not_found] ... ok
[Testcase: test_del_badge_base_sql_error] ... ok
[Testcase: test_del_badge_base_success] ... ok
[Testcase: test_lecture_badge] ... ok
=====
SUMMARY:
* RESULT: SUCCESS
* TESTS: 4
* ERRORS: 0 []
* FAILURES: 0 []
```

FIGURE 30 – Résultat du test du noeud de suppression de badge

Test du service de navigation dans database pour le login

Le fichier `test_nav_database_badge.py` teste le service `login_member.srv`, qui permet d'extraire les informations d'un utilisateur à partir de la base de données des badges RFID. Ce service est utilisé pour l'authentification des utilisateurs.

Les objectifs des tests sont les suivants :

- Vérifier que l'extraction des données fonctionne pour un utilisateur existant.
- Vérifier que le service renvoie une erreur lorsqu'un utilisateur n'est pas trouvé.

Test de connexion réussie

Ce test simule la connexion d'un utilisateur existant et vérifie que les données retournées sont correctes.

```
1 @patch('nav_database_badge.sqlite3.connect')
2 def test_extract_data_success(self, mock_sqlite_connect):
3     """Test si l'extraction des donnees fonctionne pour un utilisateur existant."""
4
5     mock_conn = MagicMock()
6     mock_cursor = MagicMock()
7     mock_sqlite_connect.return_value = mock_conn
8     mock_conn.cursor.return_value = mock_cursor
9
10    # Simuler une reponse de la base de donnees
11    mock_cursor.fetchone.return_value = (1, "hashedpassword123", "Ingenieur")
12
13    # Executer la fonction
14    response = nav_database_badge.extract_data(self.mock_service_req)
15
16    # Verifications SQL
17    mock_cursor.execute.assert_called_once_with(
18        '''SELECT id, mdp, poste FROM infos WHERE user = ?;''', ("johndoe",)
19    )
20    mock_conn.commit.assert_called_once()
21    mock_conn.close.assert_called_once()
22
23    # Verifications du retour du service
24    self.assertTrue(response.success, "Le service aurait de reussir.")
25    self.assertEqual(response.id, 1, "L'ID ne correspond pas.")
26    self.assertEqual(response.username, "johndoe", "Le username devrait etre '
27        johndoe' si trouve dans la base.")
28    self.assertEqual(response.password, "hashedpassword123", "Le mot de passe ne
29        correspond pas.")
30    self.assertEqual(response.role, "Ingenieur", "Le rele ne correspond pas.")
```

Test de connexion échouée (utilisateur inexistant)

Ce test simule une tentative de connexion avec un nom d'utilisateur qui n'existe pas en base et vérifie que le service renvoie une erreur appropriée.

```
1 @patch('nav_database_badge.sqlite3.connect')
2 def test_extract_data_failure_no_user(self, mock_sqlite_connect):
3     """Test si l'extraction echoue pour un utilisateur inexistant."""
4
5     mock_conn = MagicMock()
6     mock_cursor = MagicMock()
7     mock_sqlite_connect.return_value = mock_conn
8     mock_conn.cursor.return_value = mock_cursor
9
10    # Simuler une reponse vide (aucun utilisateur trouve)
11    mock_cursor.fetchone.return_value = None
12
13    # Executer la fonction
14    response = nav_database_badge.extract_data(self.mock_service_req)
15
16    # Verifications SQL
17    mock_cursor.execute.assert_called_once_with(
18        '''SELECT id, mdp, poste FROM infos WHERE user = ?;''', ("johndoe",)
19    )
20    mock_conn.commit.assert_called_once()
21    mock_conn.close.assert_called_once()
22
23    # Verifications du retour du service
24    self.assertFalse(response.success, "Le service aurait du echouer.")
25    self.assertEqual(response.id, 0, "L'ID aurait du etre 0.")
26    self.assertEqual(response.username, '', "Le username aurait du etre vide.")
27    self.assertEqual(response.password, '', "Le mot de passe aurait du etre vide.")
28    self.assertEqual(response.role, '', "Le role aurait du etre vide.")
```

```
root@41584ca7261b:~/Projet_it# rosrunk badge_rfid test_nav_database_badge.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/badge_rfid/rosunit-test_nav_database_badge.xml
[Testcase: test_extract_data_failure_no_user] ... ok
[Testcase: test_extract_data_success] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 2
* ERRORS: 0 []
* FAILURES: 0 []
```

FIGURE 31 – Résultat du test du noeud de navigation dans database

Conclusion des tests du package badge_rfid

Tous les tests ont été exécutés avec succès, validant les points suivants :

- L'ajout et la suppression de badges fonctionnent correctement.
- Les interactions entre les services et la base de données sont bien gérées.
- Les erreurs SQL et les cas particuliers sont correctement pris en charge.
- Un utilisateur existant peut être correctement authentifié et ses informations sont retournées.
- Une tentative de connexion avec un utilisateur inexistant est correctement gérée et renvoie une erreur.
- La base de données est bien interrogée et ses connexions sont correctement fermées.

Ces tests assurent le bon fonctionnement des services liés aux badges et leur intégration dans l'architecture globale du projet.

Tests des nœuds du package `web_interface`

Le package `web_interface` est chargé de fournir une interface web permettant d'afficher et d'interagir avec les données issues des capteurs et des badges RFID. Cette interface est basée sur `Flask` et communique avec ROS via des services et topics.

Les tests effectués dans cette section visent à valider :

- L'initialisation correcte du nœud ROS au sein de `Flask`.
- L'appel correct des services ROS pour l'ajout et la suppression de badges.
- L'authentification des utilisateurs via un service ROS.
- La récupération correcte des données depuis les bases de données `SQLite`.
- Le bon fonctionnement des API REST pour l'affichage des données capteurs et de la base de données.

Les tests ont été mis en place dans le fichier `test_web_interface.py`.

Test de l'initialisation de ROS

Le test suivant vérifie que ROS est bien initialisé dans `Flask`, et que les abonnements aux topics sont correctement effectués.

```
1 @patch('app.rospy.init_node')
2 @patch('app.rospy.Subscriber')
3 def test_init_ros(self, mock_subscriber, mock_init_node):
4     """Test si ROS est bien initialise dans Flask"""
5     init_ros()
6     mock_init_node.assert_called_with('flask_node', anonymous=True)
7     self.assertGreaterEqual(mock_subscriber.call_count, 0, "ROS peut s'abonner a
    plusieurs topics")
```

Test du service d'ajout de badge

Le service `ajout_badge` est testé pour s'assurer qu'il est bien appelé et retourne le bon résultat.

```
1 @patch('app.rospy.ServiceProxy')
2 @patch('app.rospy.wait_for_service', return_value=None)
3 def test_send_user_info(self, mock_wait, mock_service):
4     """Test si le service d'ajout de badge est bien appele"""
5
6     mock_instance = MagicMock()
7     mock_instance().success = True
8     mock_service.return_value = mock_instance
9
10    response = send_user_info("John", "Doe", "johndoe", 30, "john.doe@example.com",
11                             "password", "Engineer")
12    self.assertTrue(response, "Le service d'ajout de badge aurait du renvoyer True")
13
14    mock_service.assert_called_with('ajout_badge', mock.ANY)
```

Test du service de suppression de badge

Le service `del_badge` est testé pour vérifier qu'il est bien invoqué et fonctionne correctement.

```
1 @patch('app.rospy.ServiceProxy')
2 @patch('app.rospy.wait_for_service', return_value=None)
3 def test_del_badge_serv(self, mock_wait, mock_service):
4     """Test si le service de suppression de badge est bien appele"""
5
6     mock_instance = MagicMock()
7     mock_instance.return_value.validation = True
8     mock_service.return_value = mock_instance
9
10    response = del_badge_serv()
11    self.assertTrue(response, "Le service de suppression aurait du renvoyer True")
12
13    mock_service.assert_called_with('del_badge', mock.ANY)
```

Test de la récupération des données en base

Le test suivant s'assure que les 10 dernières valeurs d'une table SQLite sont bien récupérées.

```
1 @patch('app.sqlite3.connect')
2 def test_get_last_10_values(self, mock_sqlite):
3     """Test si la recuperation des 10 dernieres valeurs fonctionne"""
4
5     mock_conn = mock_sqlite.return_value
6     mock_cursor = mock_conn.cursor.return_value
7     mock_cursor.fetchall.return_value = [(10.5,), (20.3,)]
8
9     result = get_last_10_values("dht11_temperature.db", "temperature", "temperature"
10                                )
11    self.assertEqual(result, [10.5, 20.3])
12
13    mock_cursor.execute.assert_called()
14    mock_conn.close.assert_called_once()
```

Test de l'API /data

Ce test vérifie que l'API REST qui retourne les données des capteurs via ROS fonctionne correctement. Un utilisateur est simulé et authentifié avant l'appel.

```
1 @patch('app.get_last_10_values')
2 def test_get_database_data(self, mock_get_data):
3     """Test si l'API '/data' retourne bien les valeurs des bases SQLite"""
4
5     # Simule un utilisateur et l'ajoute au cache
6     mock_user = User(id="123", username="testuser", password="hashedpassword", role=
7         "user")
8     users_cache["123"] = mock_user
9
10    # Simule une reponse de la base de donnees
11    mock_get_data.side_effect = [
12        [22.1, 23.5], # Temperature
13        [45.0, 50.2], # Humidite
14        [70.5, 71.3] # Volume sonore
15    ]
16
17    # Simule une session utilisateur
18    with self.app.session_transaction() as session:
19        session['_user_id'] = '123'
20        session['_fresh'] = True
21
22    # Appel de l'API '/data'
23    response = self.app.get('/data')
24
25    # Verifications
26    self.assertEqual(response.status_code, 200, "L'API devrait retourner 200 et non
27        302")
28    self.assertEqual(response.json["temperature"], [22.1, 23.5])
29    self.assertEqual(response.json["humidity"], [45.0, 50.2])
30    self.assertEqual(response.json["volume"], [70.5, 71.3])
31
32    # Nettoyage
33    users_cache.pop("123", None)
```

Test de l'authentification utilisateur

Ce test vérifie que l'authentification d'un utilisateur est bien gérée via Flask-Login et la base de données.

```
1 @patch('app.load_user')
2 def test_authenticate(self, mock_load_user):
3     """Test si l'authentification fonctionne sans bloquer sur le service ROS"""
4
5     from app import User, users_cache
6
7     # Genere un mot de passe hache pour simuler la base de donnees
8     hashed_password = generate_password_hash("password")
9
10    # Simule un utilisateur
11    mock_user = User(id="123", username="testuser", password=hashed_password, role="
12    user")
13    mock_load_user.return_value = mock_user
14
15    # Ajoute aussi l'utilisateur au cache
16    users_cache["testuser"] = mock_user
17
18    # Test de l'authentification
19    user = authenticate("testuser", "password")
20
21    # Verifications
22    self.assertIsNotNone(user, "L'utilisateur ne devrait pas etre None")
23    self.assertEqual(user.username, "testuser")
24    self.assertEqual(user.role, "user")
25
26    # Nettoyage apres le test
27    users_cache.pop("testuser", None)
```

```
root@41584ca7261b:~/Projet_it# rosrund webinterface test_flask_node.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/web_interface/rosunit-test_flask_node.xml
[Testcase: test_authenticate] ... ok
[Testcase: test_del_badge_serv] ... ok
[Testcase: test_get_database_data] ... ok
[Testcase: test_get_last_10_values] ... ok
[Testcase: test_init_ros] ... ok
[Testcase: test_load_user_cache] ... ok
[Testcase: test_send_user_info] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 7
* ERRORS: 0 []
* FAILURES: 0 []
```

FIGURE 32 – Résultat du test du noeud de l'application web

Conclusion des tests du package web_interface

Tous les tests ont été exécutés avec succès, validant les points suivants :

- L'initialisation ROS est bien intégrée à Flask.
- Les services ROS d'ajout et de suppression de badges fonctionnent correctement.
- L'authentification des utilisateurs est bien gérée et sécurisée.
- La récupération des données en base est correcte.

— Les API REST retournent bien les informations attendues.

Ces tests garantissent que l'interface web est opérationnelle et correctement intégrée à l'architecture globale du projet.

Tests d'intégration

Les tests d'intégration visent à vérifier le bon fonctionnement des interactions entre les différents composants du système. Contrairement aux tests unitaires qui évaluent des fonctions isolées, ces tests s'assurent que :

- Les nœuds ROS communiquent correctement via les topics et services.
- Les données sont bien stockées et récupérées depuis la base de données.
- L'interface web Flask interagit correctement avec ROS et SQLite.

Les tests d'intégration suivants ont été réalisés.

—

Test d'intégration de l'ajout de badge

Le fichier `test_integration_ajout_badge.py` vérifie que l'ensemble du processus d'ajout d'un badge fonctionne correctement, depuis la lecture du badge via un topic ROS jusqu'à son enregistrement dans la base de données.

1. Un badge est simulé et publié sur le topic `/topic_rfid`.
2. Le service `ajout_badge` est appelé pour ajouter ce badge.
3. La base de données est vérifiée pour s'assurer que l'ajout a bien eu lieu.

```

1 def test_ajout_badge_integration(self):
2     """Test d'integration de l'ajout de badge"""
3
4     # Simuler un badge scanne
5     badge_pub = rospy.Publisher("topic_rfid", Int32, queue_size=1)
6     rospy.sleep(1)
7     badge_pub.publish(Int32(999999))
8     rospy.sleep(1)
9
10    # Appeler le service 'ajout_badge'
11    request = ajout_badgeRequest()
12    request.prenom = "Alice"
13    request.nom = "Dupont"
14    request.username = "alice_dupont"
15    request.age = 30
16    request.mail = "alice.dupont@example.com"
17    request.password = "securepassword"
18    request.poste = "Developpeur"
19
20    response = self.client_ajout(request)
21    self.assertTrue(response.success, "L'ajout de badge aurait du reussir !")
22
23    # Verifier que le badge est bien dans la base
24    self.cursor.execute("SELECT * FROM infos WHERE numBadge = ?", (999999,))
25    result = self.cursor.fetchone()
26    self.assertIsNotNone(result, "Le badge devrait etre enregistre en base !")
27    self.assertEqual(result[2], "alice_dupont", "Le username est incorrect")
28    self.assertEqual(result[3], "Alice", "Le prenom est incorrect")
29    self.assertEqual(result[4], "Dupont", "Le nom est incorrect")
30    self.assertEqual(result[6], "alice.dupont@example.com", "L'email est incorrect")

```

```

root@0ec096005029:~/Projet_it# rosrund badge_rfid test_integration_ajout_badge.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/badge_rfid/rosunit-test_integration_ajout_badge.xml
[Testcase: test_ajout_badge_integration] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 1
* ERRORS: 0 []
* FAILURES: 0 []

```

FIGURE 33 – Résultat du test d'intégration d'ajout d'un badge

Test d'intégration de la suppression de badge

Le fichier `test_integration_del_badge.py` teste le processus de suppression de badge :

1. Un badge est ajouté manuellement à la base de données.
2. Le service `del_badge` est invoqué après la simulation du scan du badge.
3. La base de données est vérifiée pour s'assurer que le badge a bien été supprimé.


```

1 def test_suppression_badge_integration(self):
2     """Test de suppression d'un badge"""
3
4     # Simuler un badge scanne
5     badge_pub = rospy.Publisher("topic_rfid", Int32, queue_size=1)
6     rospy.sleep(1)
7     badge_pub.publish(Int32(888888))
8     rospy.sleep(1)
9
10    # Appeler le service 'del_badge'
11    response = self.client_suppression(True)
12    self.assertTrue(response.validation, "Le badge aurait du etre supprime !")
13
14    # Verifier que le badge n'est plus dans la base
15    self.cursor.execute("SELECT * FROM infos WHERE numBadge = ?", (888888,))
16    result = self.cursor.fetchone()
17    self.assertIsNone(result, "Le badge devrait avoir ete supprime !")

```

```

root@0ec096005029:~/Projet_it# rosrund badge_rfid test_integration_del_badge.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/badge_rfid/rosunit-test_integration_del_badge.xml
[Testcase: test_suppression_badge_integration] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 1
* ERRORS: 0 []
* FAILURES: 0 []

```

FIGURE 34 – Résultat du test d'intégration de suppression d'un badge

Test d'intégration de l'interface Flask

Le fichier `test_integration_flask.py` valide le bon fonctionnement de l'API Flask et sa communication avec ROS et SQLite.

Les points suivants sont vérifiés :

— L'API `/data` retourne correctement les données enregistrées en base.

Test de l'API `/data` :

```

1 def test_get_database_data(self):
2     """Tester si Flask recupere bien les donnees des bases SQLite."""
3     response = requests.get(f"{FLASK_URL}/get_data")
4     self.assertEqual(response.status_code, 200, "L'API /get_data devrait retourner 200")
5
6     data = response.json()
7     self.assertIn("temperature", data, "La cle temperature doit etre presente")
8     self.assertIn("humidity", data, "La cle humidite doit etre presente")
9     self.assertIn("volume", data, "La cle volume doit etre presente")

```

En raison de la nécessité de s'être "login" pour pouvoir accéder à ces infos, le code doit aussi contenir les identifiants de connexion :

```
1 def setUpClass(cls):
2     """Verifier que Flask est bien demarre et se connecter"""
3     rospy.init_node('test_integration_flask', anonymous=True)
4     time.sleep(5) # Laisser Flask se lancer
5     cls.session = requests.Session() # Creer une session persistante
6
7     # Connexion avec un utilisateur valide
8     login_data = {
9         "username": "admin",
10        "password": "admin"
11    }
12    response = cls.session.post(f"{FLASK_URL}/", data=login_data)
13
14    if response.status_code != 200:
15        print("Echec de la connexion :", response.text)
```

Voici les résultats :

```
root@096005029:~/Projet_it# rosrund webinterface test_integration_flask.py
[ROSUNIT] Outputting test results to /root/.ros/test_results/webinterface/rosunit-test_integration_flask.xml
[Testcase: test_get_database_data] ... ok
-----
SUMMARY:
* RESULT: SUCCESS
* TESTS: 1
* ERRORS: 0 []
* FAILURES: 0 []
```

FIGURE 35 – Résultat du test d'intégration de données pour les graphiques sur la page web

—

Conclusion des tests d'intégration

Tous les tests d'intégration ont été exécutés avec succès, validant ainsi :

- La chaîne complète d'ajout et de suppression de badges (ROS → base de données).
- La bonne communication entre Flask et ROS via les services.
- La récupération correcte des données bases via l'API.

Ces tests garantissent que l'ensemble des modules interagissent correctement dans un environnement ROS fonctionnel, assurant ainsi la fiabilité du système.

IX. Conclusion et perspectives

Pour conclure sur ce projet, nous dirions que ce fut un projet complexe, nous permettant à la fois de travailler nos compétences de programmation basiques, mais aussi en développer de nouvelles avec l'intégration de ROS, l'utilisation de base de données en temps réel et le développement d'une page web. Il nous a aussi permis d'enrichir nos capacités de travail de groupe et de gestion de projet.

Bien que complet, ce projet reste ouvert à de futures améliorations, notamment sur certains points :

- Accès à la page web depuis un réseau externe et fonctionnement en continu de la station de surveillance (capteurs) : Pour l'instant le site internet n'est accessible que sur le réseau privé de l'école lorsque la Raspberry est allumée et connectée à ce même réseau. Pour pousser le concept plus loin, il faudrait héberger la page web et laisser la station branchée sans interruption.
- Amélioration du matériel : Les capteurs utilisés sont ceux disponibles dans les kit bon marché disponible à la vente. Ces capteurs sont loin d'être parfait et sont souvent soumis aux erreurs de mesures et aux pannes. Une amélioration de ceux-ci peut s'envisager dans le cas d'une reprise de ce projet.
- Tests de de-bug plus intensifs et retours utilisateurs : Bien que de nombreux tests concluants aient été effectués, il est possible que nous n'ayons pas passer en revue certains cas spécifiques. Un retour de la part de plusieurs utilisateurs peut aussi être une solution à considérer pour éviter de rencontrer tout problème ou bug futur.

X. Annexes

1. Documentations du matériel

A. Capteur DHT11

Lien de la [documentation](#)

B. Capteur KY-038

Lien de la [documentation](#)

C. Capteur RFID RC522

Lien de la [documentation](#)

D. Raspberry PI 4B

Lien de la [documentation](#)

E. Câbles utilisés

Lien du [site d'achat \(Amazon\)](#)

2. Documentations des logiciels et outils utilisés

A. Visual Studio Code

Lien du tutoriel pour [Visual Studio Code](#)

B. Docker

Lien du tutoriel pour [Docker](#)

C. Sqlite3

Lien du tutoriel pour [Sqlite3](#)

D. ROS

Lien du tutoriel pour [ROS](#)

E. Flask

Lien du tutoriel pour [Flask](#)