Université de Toulouse
IUT de Blagnac
Département Informatique

Doc. Technique

Application JavaFX IOT

SAE 3.01

Nom : BOULOUIHA Yassir, RUIZ Nicolas, IBRAHIM Marwane, LOVIN Alex

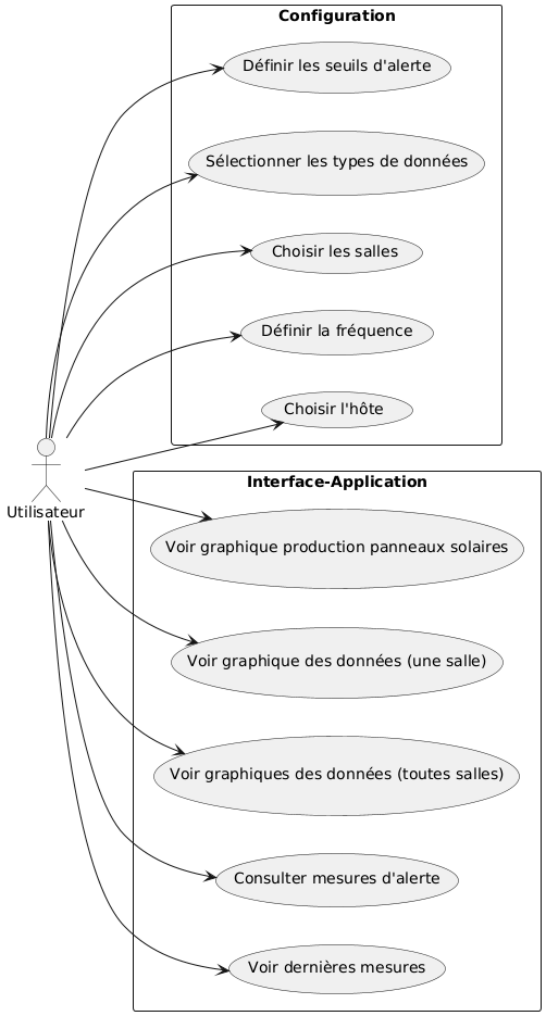
Informatique 2ème Année

Table des matières :

1.Présentation de l'application

Cette application a pour but de récolter les données des différents capteurs des salles de l'IUT de blagnac et panneaux solaires a l'aide d'un script python et de les visualiser sous forme de graphiques, l'application permets à travers une interface de configuration de choisir les salles auxquelles on souhaite s'abonner, la fréquence de lecture de données depuis le serveur, l'hôte, les types de données voulues (température CO2 etc..) et de paramétrer des seuils d'alerte pour chaque type de donnée.

2.Use case Global



Le diagramme de cas d'utilisation décrit les interactions entre un utilisateur et deux composants principaux de l'application : Configuration et Interface-Application.

Dans la section Configuration, l'utilisateur peut choisir l'hôte, définir la fréquence des mises à jour, choisir les salles auxquelles s'abonner, sélectionner les types de données souhaités et définir les seuils d'alerte.

Dans la section Interface-Application, l'utilisateur peut voir les dernières mesures enregistrées, consulter les mesures d'alerte, visualiser les graphiques des données pour toutes les salles, voir les graphiques des données pour une salle spécifique, et visualiser l'évolution de la production des panneaux solaires.

3.Architecture

3.1 Architecture générale

L'application JavaIoT est structurée en plusieurs sous-systèmes, chacun ayant un rôle bien défini pour assurer un fonctionnement efficace et modulaire. Voici une description détaillée de l'architecture générale en termes de sous-systèmes et le rôle de chaque élément :

3.1.1 Sous-Systèmes de l'Application

Config (Configuration)

- **AppConstants** : Cette classe contient les constantes utilisées à travers l'application. Elle centralise les valeurs fixes pour une maintenance facilitée.
- **ConfigManager** : Gère la lecture et l'écriture des fichiers de configuration. Il permet de charger les paramètres de l'application, tels que les informations sur l'hôte, la fréquence des mises à jour, les salles abonnées, etc.

Control (Contrôle)

- **RoomsPane** : Gère la vue des salles en chargeant un fichier FXML.

Tasks (Tâches)

- **PythonScriptRunner** : Runnable qui représente l'exécution du script Python nécessaire pour collecter les données.

Tools (Outils)

- **SensorType** : Définit les différents types de capteurs disponibles, avec leurs clés, abréviations et unités de mesure. Il facilite l'identification et l'utilisation cohérente des capteurs dans l'application.
- **StageManagement** : Permet de centrer deux fenêtres entre elles.

View (Vues)

- **AccueilViewController** : Contrôleur pour la vue d'accueil.
- **AlertPageViewController** : Contrôleur pour la page des alertes, affichant les mesures qui dépassent les seuils définis.
- **ConfigViewController** : Contrôleur pour la page de configuration.
- **GraphiquesController** : Contrôleur qui gère l'affichage des graphiques des différentes données pour toutes les salles ou pour une salle spécifique.
- **RoomsViewController** : Contrôleur pour la vue des salles, affichant toutes les salles auxquelles l'utilisateur est abonné avec leurs dernières mesures.
- **SalleDetailsController** : Contrôleur, Affiche sous forme de graphique les dernières données d'une salle.
- **SeuilsViewController** : Contrôleur d'une page qui fait partie de la configuration (Seuils d'Alerte).
- **SolarPanelController** : Contrôleur, Affiche les graphiques de l'évolution de la production des panneaux solaires.
- **LaunchApp** : Point d'entrée de l'application. Initialise et lance l'application.

- **SyncData** : Singleton qui gère la récupération des données aux intervalles définis dans les dossiers et gère les threads Python. À partir de ce singleton, les différentes classes de l'application peuvent lui demander les données.

Model.Data (Modèle de Données)

- **Measure** : Représente une mesure collectée par un capteur. Contient les valeurs associées aux différents types (ex “temp” : 18, “co2” : 400).
- **Room** : Représente une salle contenant des **Measure**. Stocke les informations sur la salle et ses mesures.
- **Solar** : Représente une mesure recoltée de production des panneaux solaires, incluant les informations sur la production actuelle et les historiques.

2. Rôle des Éléments

- **Structure de Données** **(model.data)** : Stocke les mesures collectées par les capteurs et les informations historiques. Ils sont généralement gérés par les classes du modèle de données (Measure, Room, Solar).
- **Configuration** **(application.config)** : Contient les paramètres de configuration de l'application, tels que l'hôte, la fréquence des mises à jour, et les seuils d'alerte. Gérés par ConfigManager.

- **Parties de l'Application** :
  - **Vue** (application.view) : Comprend les différents contrôleurs de vue responsables de l'interaction avec l'utilisateur et l'affichage des données.
  - **Contrôle** (application.control) : Assure la logique de gestion des données et la communication entre le modèle et la vue.
  - **Tâches** (application.tasks) : Gère les tâches d'arrière-plan comme l'exécution de scripts Python pour la collecte de données.
  - **Outils** (application.tools) : Fournit des utilitaires et des fonctions auxiliaires pour l'application.

3.2 Ressources externes utilisées

3.2.1 Jackson DataBind

Dépendance :

```
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>2.18.2</version>
</dependency>
```

Rôle : Utilisé pour la conversion entre les objets Java et les données JSON. Il permet la sérialisation des objets Java en JSON et la désérialisation des JSON en objets Java.

3.2.2 ini4j

Dépendance :

```
<dependency>
   <groupId>org.ini4j</groupId>
   <artifactId>ini4j</artifactId>
   <version>0.5.4</version>
</dependency>
```

Rôle : Permet la lecture et l'écriture de fichiers de configuration au format INI. Il est utilisé pour lire la configuration dans certaines classes en dehors de ConfigManager.

3.3 Structuration de l'application

La structuration de l'application repose sur des principes de modularité et d'organisation claire des responsabilités. Le design adopté utilise le modèle **MVC (Modèle-Vue-Contrôleur)** pour séparer les différentes préoccupations de l'application.

Le rôle central est occupé par la classe **`SyncData`**, qui assure la récupération des données de façon périodique à chaque fréquence, en respectant le principe de Singleton de façon à ce que toutes les classes accèdent à la même instance pour la récupération de données. Voici un aperçu détaillé de la structuration en packages, ainsi que des justifications et explications des choix faits.

Structure des Packages

L'application est divisée en plusieurs packages, chacun ayant un rôle spécifique dans l'architecture globale. Voici la répartition détaillée :

1. **`application`** : Ce package contient les éléments principaux de l'application, y compris la classe de lancement (`LaunchApp`), qui initialise l'application et charge la vue principale, ainsi que la classe `SyncData`, qui est responsable de la gestion des données et de leur mise à jour.

2. **`application.tasks`** : Ce package contient la classe `PythonScriptRunner` (Runnable du script Python).

3. **`application.view`** : Ce package contient les contrôleurs JavaFX (comme `AccueilViewController`), qui gèrent les interactions avec l'interface utilisateur. Ces classes sont responsables de la logique d'affichage des données et de la gestion des événements utilisateurs.

4. **`model.data`** : Ce package regroupe les classes représentant les entités principales de l'application, comme `Room`, `Measure`, et `Solar`, qui sont des modèles de données représentant respectivement des salles, des mesures et des panneaux solaires. Ces classes sont essentielles pour la gestion des données de capteurs et panneaux solaires.

5. **`config`** : Ce package contient des fichiers de configuration et des classes liées à la configuration de l'application. Par exemple, des paramètres relatifs à l'accès aux fichiers JSON, les seuils d'alerte, et la configuration de connexion au service externe (MQTT) peuvent y être définis. Cela permet de centraliser toute la configuration de l'application pour simplifier sa gestion.

6. **`control`** : Ce package regroupe les classes responsables de la gestion des interactions entre la logique de l'application et l'interface utilisateur. Il contient des médiateurs qui coordonnent la mise à jour des vues et les actions métier. Par exemple, **RoomsPane** gère l'affichage et la gestion de la vue des salles, en se chargeant de l'initialisation des fenêtres et de la coordination avec les contrôleurs de la vue. Ce package permet de séparer la logique de présentation de l'application de la logique de gestion des fenêtres et interactions utilisateur.

7. **`tools`** : Ce package contient des classes utilitaires qui fournissent des fonctionnalités générales utilisées dans l'application. Il inclut des outils comme **SensorType**, qui définit les types de capteurs et leurs unités de mesure, et **`StageManagement`**, qui facilite la gestion de la disposition des fenêtres, notamment en permettant de centrer une fenêtre sur une autre. Ces classes ne sont pas directement liées à la logique métier, mais offrent des services pratiques pour l'application.

Principe de Structuration : Le Singleton au Centre

La classe **`SyncData`** occupe une place centrale dans l'application. Elle est un exemple de la mise en œuvre du pattern **Singleton**, garantissant qu'une seule instance de cette classe existe dans l'application. Cette instance est utilisée pour centraliser la gestion des données relatives aux salles et aux panneaux solaires.

Justification du Singleton : Le modèle **Singleton** est particulièrement adapté dans ce cas, car l'application nécessite un point d'accès unique aux données partagées à travers toute l'application. Utiliser une seule instance de **`SyncData`** permet de garantir la cohérence des données et de simplifier l'accès aux informations partagées. Cela évite des problèmes de duplication ou de gestion d'état à travers différentes parties de l'application.

La classe **`SyncData`** assure plusieurs responsabilités :

- **Gestion des données** : Elle récupère les données des fichiers JSON (salles, mesures, panneaux solaires) et les structure en objets accessibles via des listes observables et des maps.
- **Communication avec l'interface utilisateur** : Elle met à jour les objets **ObservableList** (comme `roomsOlist`) afin que l'interface utilisateur puisse afficher des informations en temps réel.
- **Gestion des alertes** : Lorsqu'une mesure dépasse un seuil critique, **`SyncData`** envoie une alerte via des pop-ups sur l'interface. Grâce à l'utilisation de **`Platform.runLater`**, la mise à jour de l'interface pour les alertes se fait sur le thread de l'UI
- **Exécution périodique** : Elle déclenche des tâches de fond, comme l'exécution du script Python pour la sauvegarde des données via sa classe privée qui lui est intégrée **`PeriodicPythonTask`**.

Le Pattern MVC (Modèle-Vue-Contrôleur)

L'application suit également le pattern **MVC**, où chaque composant est bien séparé selon son rôle spécifique.

1. **Modèle (`Model`)** :
   - Le modèle est constitué des classes **`Room`**, **`Measure`**, et **`Solar`**, qui représentent les entités de l'application. Ces classes contiennent les données de base (par exemple, les mesures de température ou d'énergie pour les panneaux solaires).
   - **`SyncData`** est également un modèle, puisqu'il centralise et structure toutes les données de l'application et les rend accessibles via des méthodes comme `getRoomsMap()` et `getRoomsOlist()`.
2. **Vue (`View`)** :
   - La vue est principalement gérée par les fichiers FXML et les contrôleurs JavaFX situés dans le package **`application.view`**. Le contrôleur **`RoomsViewController`** interagit avec la vue **`RoomsPane`** pour afficher les données récupérées par **`SyncData`**.
   - Cette séparation permet à la vue de se concentrer uniquement sur la présentation des données, tandis que la logique métier et de gestion des données est isolée dans **`SyncData`**.
3. **Contrôleur (`Controller`)** :
   - Les contrôleurs sont responsables de la gestion des interactions utilisateur et de la logique de présentation. Par exemple, **`AccueilViewController`** utilise **`SyncData`** pour récupérer les données des salles et des panneaux solaires et

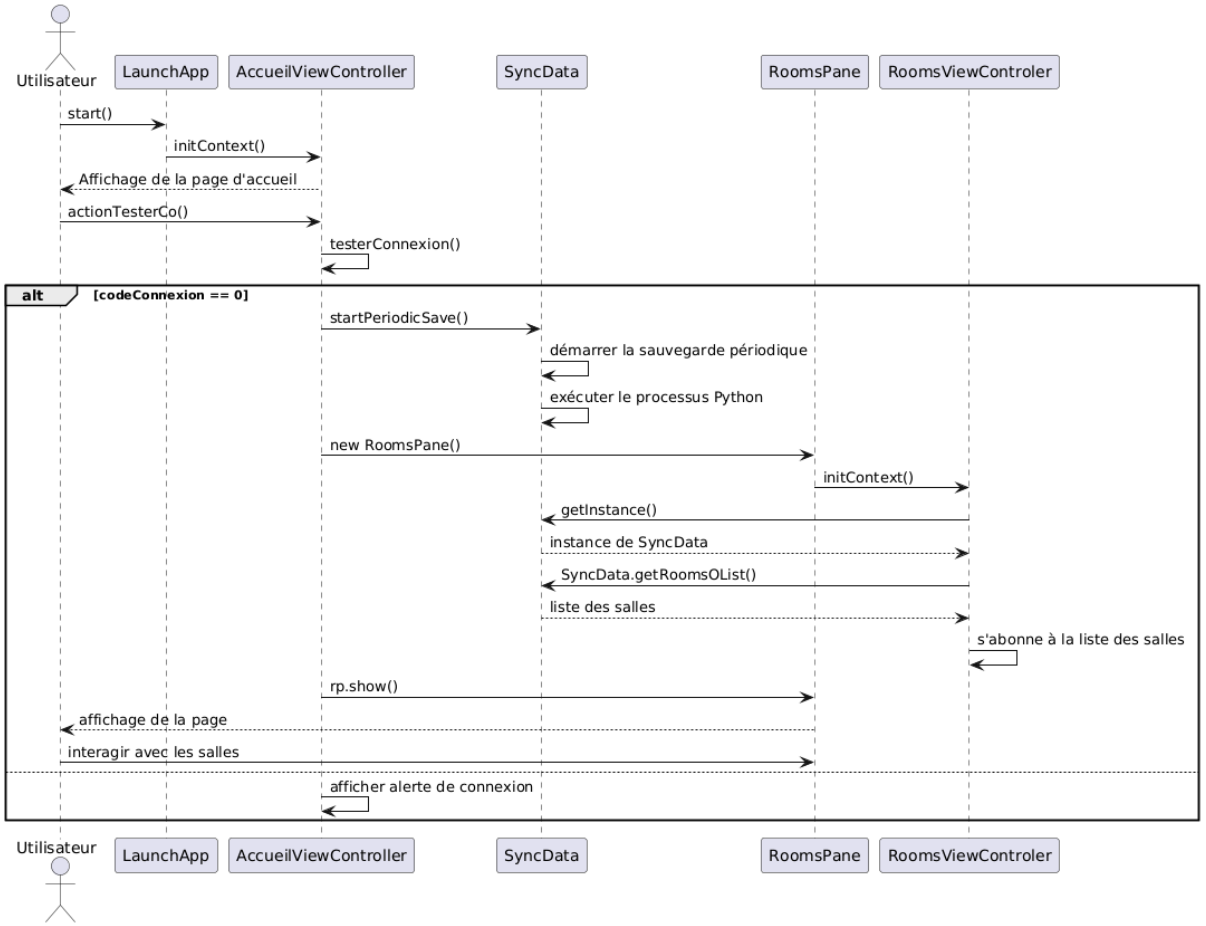
les afficher. Il interagit avec la vue et est le point de liaison entre le modèle (les données) et la vue (l'interface utilisateur).
- Ce découplage garantit que la logique métier (comme la gestion des données de capteurs) est séparée de la logique de présentation (comme l'affichage dans l'UI).

Gestion des Données Asynchrones et Périodiques

Une partie essentielle de l'application réside dans la gestion des données asynchrones. Les données des capteurs et panneaux solaires sont mises à jour périodiquement, et ces mises à jour doivent être effectuées en arrière-plan sans bloquer l'interface utilisateur. La classe **`PeriodicPythonTask`**, qui gère l'exécution périodique d'un script Python, est un exemple de gestion asynchrone dans l'application. Cela permet de récupérer et de sauvegarder les données de manière régulière sans perturber l'expérience utilisateur.

Le **`ScheduledExecutorService`** est utilisé pour planifier l'exécution périodique des tâches. Cette approche garantit que les mises à jour des données sont effectuées de manière autonome en arrière-plan, tout en permettant à l'interface utilisateur de rester réactive. Le script Python peut être exécuté à une fréquence définie (par exemple, toutes les 15 minutes) et récupère les dernières données des capteurs. Lors de la principale exécution du script python, on définit l'heure à laquelle il a été exécuté +3 secondes et on applique la fréquence sur cette heure modifiée de cette façon le script laissera assez de temps au script python d'écrire les données et lire périodiquement les dossiers de façon correcte sans doubler la fréquence (si on faisait simplement fréquence +3 secondes)

3.4 Diag Séquence Lancement de l'application



Ce diagramme de séquence décrit le processus de lancement et d'affichage des salles de l'application. L'utilisateur démarre l'application en l'exécutant donc implicitement avec `start()`, ce qui initialise le contexte de la vue d'accueil via `initContext()`. La page d'accueil est alors affichée. L'utilisateur effectue une action pour tester la connexion en interagissant avec l'interface ce qui produit l'exécution de `actionTesterCo()`. Ensuite, la méthode `testerConnexion()` dans `actionTesterCo()` vérifie le code de connexion.

Si le code de connexion est égal à 0 (connexion réussie), le processus suivant est déclenché : la sauvegarde périodique commence avec `startPeriodicSave()` et le processus Python est exécuté. Un nouvel objet `RoomsPane` est créé, son contexte est initialisé et les informations sur les salles sont récupérées via `SyncData`. Une fois les salles récupérées, `RoomsViewControler` s'abonne à la liste des salles. La page de salle est affichée à l'utilisateur, qui peut interagir avec les salles. Si la connexion échoue (codeConnexion != 0), une alerte de connexion est affichée à l'utilisateur.

4. Fonctionnalités

4.1 Affichages Graphiques

Les classes `SolarPanelController`, `GraphiquesController` et `SalleDetailsController` sont responsables de l'affichage graphique des données collectées par les capteurs et les panneaux solaires. Elles sont utilisées pour fournir à l'utilisateur une vue en temps réel de l'énergie produite par les panneaux solaires et des mesures des salles (température, pression, luminosité, etc.). Voici un aperçu détaillé de chacune de ces classes :

4.1.1 SolarPanelController

Description Générale :

Le contrôleur **`SolarPanelController`** est conçu pour afficher les données énergétiques des panneaux solaires sous forme de graphique linéaire interactif. Chaque série de données représente les mesures énergétiques d'un panneau solaire sur une période donnée. Ce graphique fournit une

visualisation claire et informative de la production d'énergie des panneaux solaires pour une analyse rapide et efficace.

Fonctionnement des Méthodes Principales :

- **`initialize()`** :
  Cette méthode configure le graphique à son lancement :
  1. Elle initialise une série de données appelée `energySeries`, qui regroupera les valeurs de production énergétique.
  2. Elle appelle la méthode `loadSolarData` pour récupérer les données des panneaux solaires via le singleton **`SyncData`**.
  3. Une fois les données chargées, la série est ajoutée au graphique `energyChart` pour afficher les résultats.
     Utilité : garantir que le graphique est prêt à afficher des informations dès le démarrage de l'interface.

- **`loadSolarData()`** :
  Cette méthode charge les données des panneaux solaires à partir de la structure centrale `SyncData`.
  1. Elle récupère une liste d'objets **`Solar`**, chacun représentant un panneau solaire.
  2. Si aucune donnée n'est disponible, un message d'avertissement est affiché.
  3. Pour chaque panneau solaire, la méthode appelle `updateEnergyChart` pour actualiser le graphique avec ses données.
     Utilité : assurer que toutes les informations disponibles sur les panneaux solaires sont affichées dans l'interface utilisateur.

- **`updateEnergyChart(Solar solar)`** :
  Cette méthode met à jour le graphique avec les données d'un panneau solaire spécifique :
  1. Elle commence par effacer les anciennes données pour éviter tout conflit avec les nouvelles valeurs.
  2. Elle parcourt la carte des données énergétiques (**`energyMap`**) du panneau solaire pour ajouter chaque mesure au graphique.
  3. Si aucune donnée n'est ajoutée, un message est affiché pour en informer l'utilisateur.
     Utilité : maintenir un graphique précis et à jour pour chaque panneau solaire, en gérant dynamiquement les modifications des données.

4.1.2 GraphiquesController

Description Générale :

Cette fonctionnalité permet de visualiser les données collectées des capteurs installés dans différentes salles sous forme de graphiques interactifs. Les données mesurées incluent des paramètres tels que la température, la pression, et d'autres caractéristiques environnementales. Les graphiques générés peuvent être des graphiques à barres ou des graphiques linéaires en fonction de la nature des données.

Les graphiques sont affichés dans des onglets spécifiques, chaque onglet étant associé à un type de donnée mesurée (par exemple, température ou

humidité). Ces visualisations permettent une interprétation rapide des données par les utilisateurs.

Fonctionnement des Méthodes Principales :

- **`initialize()`:**
  Cette méthode initialise le contrôleur en charge de la gestion des graphiques. Elle effectue les étapes suivantes :
  1. Récupère les données des salles à partir de l'objet singleton `SyncData`.
  2. Parcourt les données pour organiser les valeurs mesurées en fonction des salles et des types de capteurs.
  3. Crée des onglets pour chaque type de donnée mesurée (par exemple, température, humidité), et y associe un graphique correspondant.
  4. Les graphiques sont générés dynamiquement en appelant les méthodes `createBarChart` ou `createLineChart`.

Cette méthode permet d'assurer une liaison entre les données brutes et leur affichage graphique.

- **`createChart(String key, Map<String, Double> data)`:**
  Cette méthode détermine le type de graphique à créer en fonction du type de données (`key`).
  - Si les données correspondent à des paramètres continus comme la température ou la pression, un graphique linéaire est généré via `createLineChart`.
  - Sinon, un graphique à barres est généré via `createBarChart`.

Cette méthode joue le rôle de "distributeur" de graphiques, s'assurant que chaque type de donnée est affiché de manière optimale.

- **`createBarChart(String key, Map<String, Double> data):`**
  Cette méthode crée un graphique à barres pour afficher les données :
  1. Définit un axe des catégories (x) et un axe numérique (y).
  2. Ajoute les valeurs des données dans une série de données (`XYChart.Series`), en associant chaque salle (axe x) à sa valeur mesurée (axe y).
  3. Retourne un graphique à barres contenant les données.

Cette méthode est utilisée pour représenter des données discrètes ou catégoriques.

- **`createLineChart(String key, Map<String, Double> data):`**
  Cette méthode crée un graphique linéaire pour afficher des données continues :
  1. Définit un axe des catégories (x) et un axe numérique (y).
  2. Ajoute les valeurs des données dans une série de données (`XYChart.Series`), en associant chaque salle (axe x) à sa valeur mesurée (axe y).
  3. Retourne un graphique linéaire contenant les données.

Ce type de graphique est utilisé pour des données montrant une progression ou des relations continues, comme la pression ou la température.

4.1.3 SalleDetailsController

Description Générale :

Cette fonctionnalité permet d'afficher des détails spécifiques à une salle sélectionnée, tels que ses graphiques de données, tout en surveillant activement les alertes et les nouvelles données. L'interface se compose d'un onglet unique regroupant les graphiques associés à cette salle. Les données affichées incluent les mesures capturées par les capteurs dans cette salle (par exemple, température, humidité, CO_2).

La surveillance des fichiers de données et des alertes garantit que les graphiques sont mis à jour automatiquement dès qu'un changement est détecté dans les répertoires correspondants.

Fonctionnement des Méthodes Principales :

- **`setRoom(Room room)`** :
  Cette méthode initialise la vue pour une salle spécifique :
  1. Elle configure les onglets contenant les graphiques pour la salle en appelant `createRoomTab`.
  2. Elle démarre deux mécanismes de surveillance :
     - La surveillance des fichiers d'alertes dans le répertoire spécifié.
     - La surveillance des nouvelles données dans le répertoire général des données.
       Cette méthode garantit que les graphiques affichent les informations pertinentes pour la salle sélectionnée.

- **`createRoomTab(Room room)`** :
  Cette méthode crée un onglet interactif contenant un graphique qui représente les données spécifiques de la salle.
  1. Elle génère un graphique pour les données de la salle en appelant `createRoomChart`.
  2. Elle vide les anciens onglets pour s'assurer qu'un seul onglet, correspondant à la salle actuelle, est affiché.

Cette méthode est essentielle pour une organisation claire et une présentation propre des données de la salle.

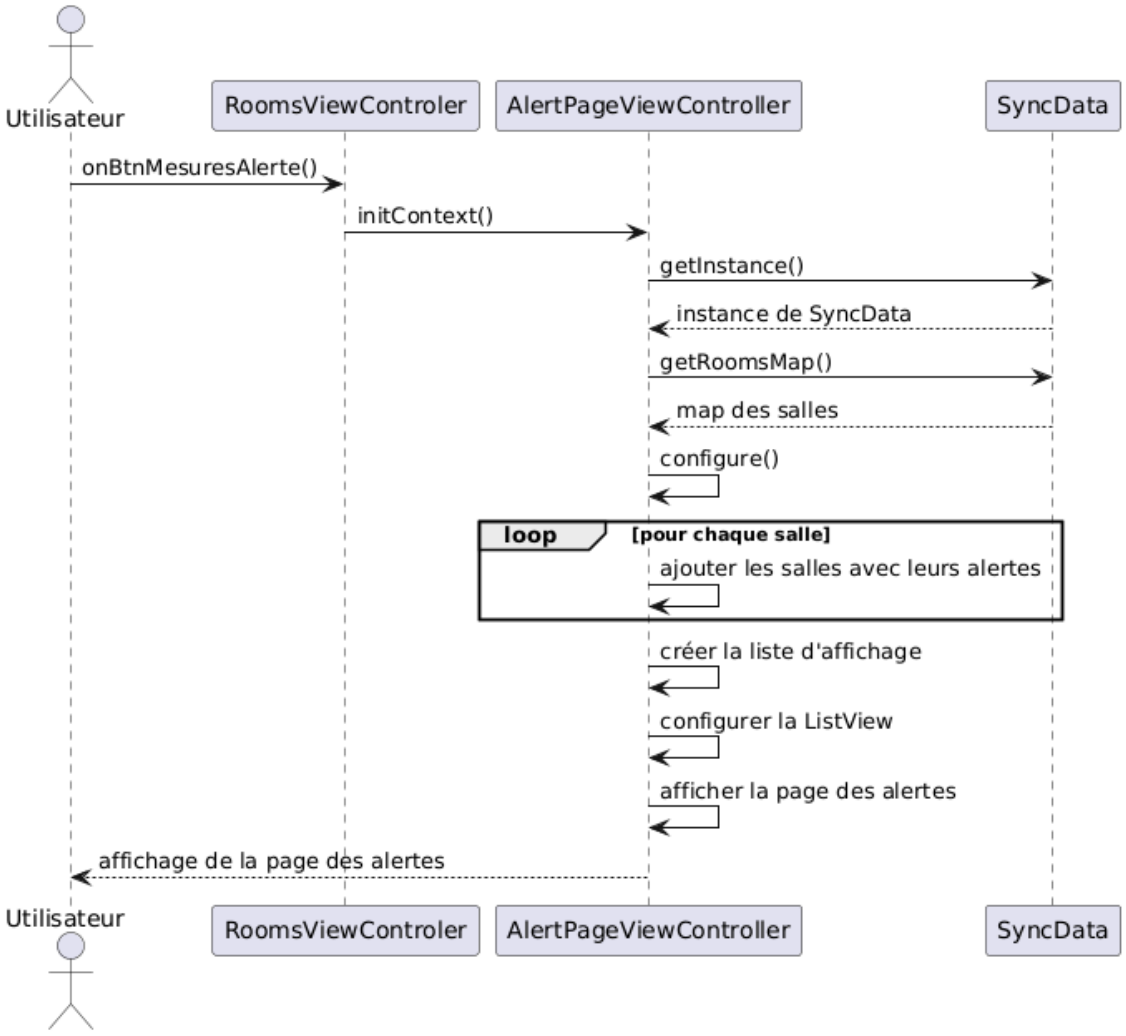
- **`createRoomChart(Room room)`** :
  Cette méthode génère un graphique à barres pour afficher les données d'une salle donnée :
  1. Un graphique à barres est configuré avec des axes pour représenter les types de mesures (axe x) et leurs valeurs (axe y).
  2. Les données de la salle sont parcourues et ajoutées au graphique sous forme de séries, chaque mesure étant associée à son type et sa valeur.
  3. Le graphique est retourné pour être affiché dans l'onglet de la salle.

Cette méthode assure une représentation visuelle intuitive des données mesurées dans la salle.

- **`updateCharts()`** :
  Cette méthode permet de rafraîchir les graphiques de manière asynchrone (via la plateforme JavaFX). Elle recrée les onglets et leurs graphiques lorsque de nouvelles données sont disponibles.

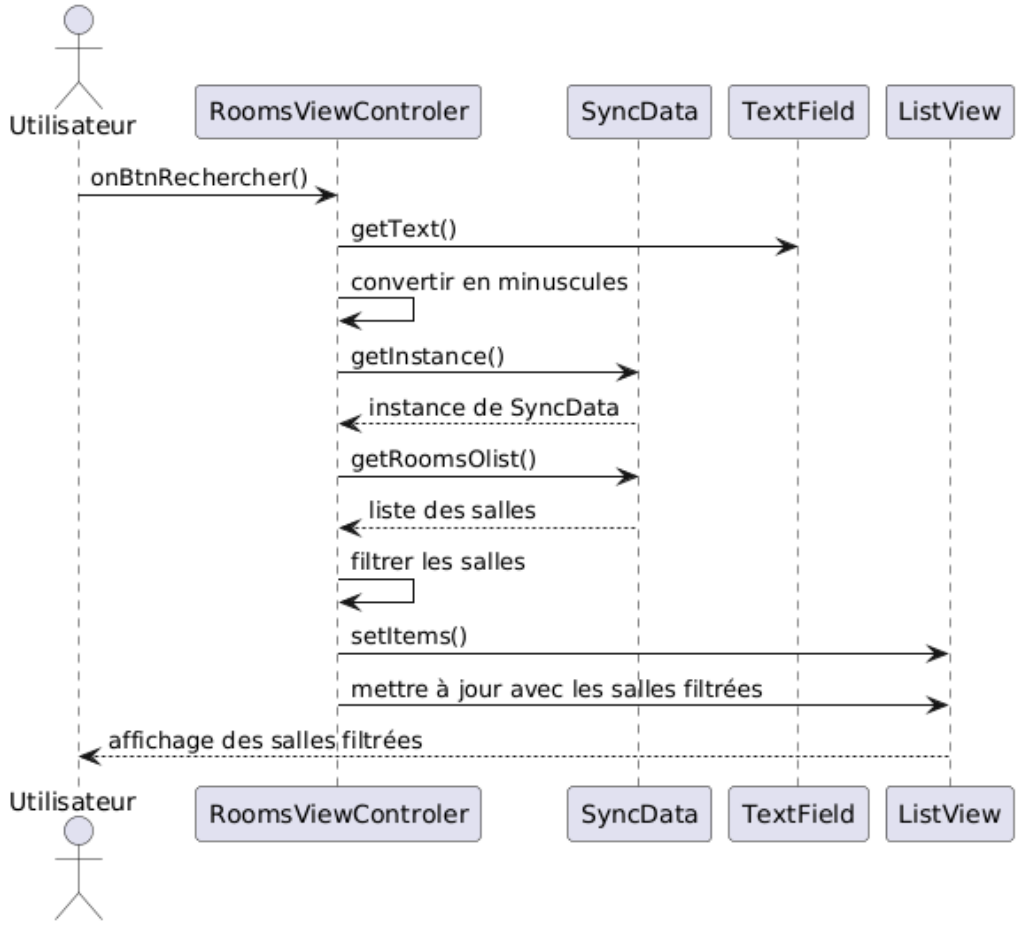
Les contrôleurs `SolarPanelController`, `GraphiquesController`, et `SalleDetailsController` sont des éléments clés dans l'interface graphique de l'application. Ils gèrent l'affichage des données en temps réel collectées à partir des capteurs et des panneaux solaires. Chaque contrôleur est conçu pour une fonctionnalité spécifique : le suivi de l'énergie des panneaux solaires, l'affichage des mesures des salles, et la gestion des alertes. Ces contrôleurs utilisent des threads pour surveiller en continu les nouvelles données et mettre à jour l'affichage sans interrompre l'expérience utilisateur.

4.2 Affichage des mesures d'alerte



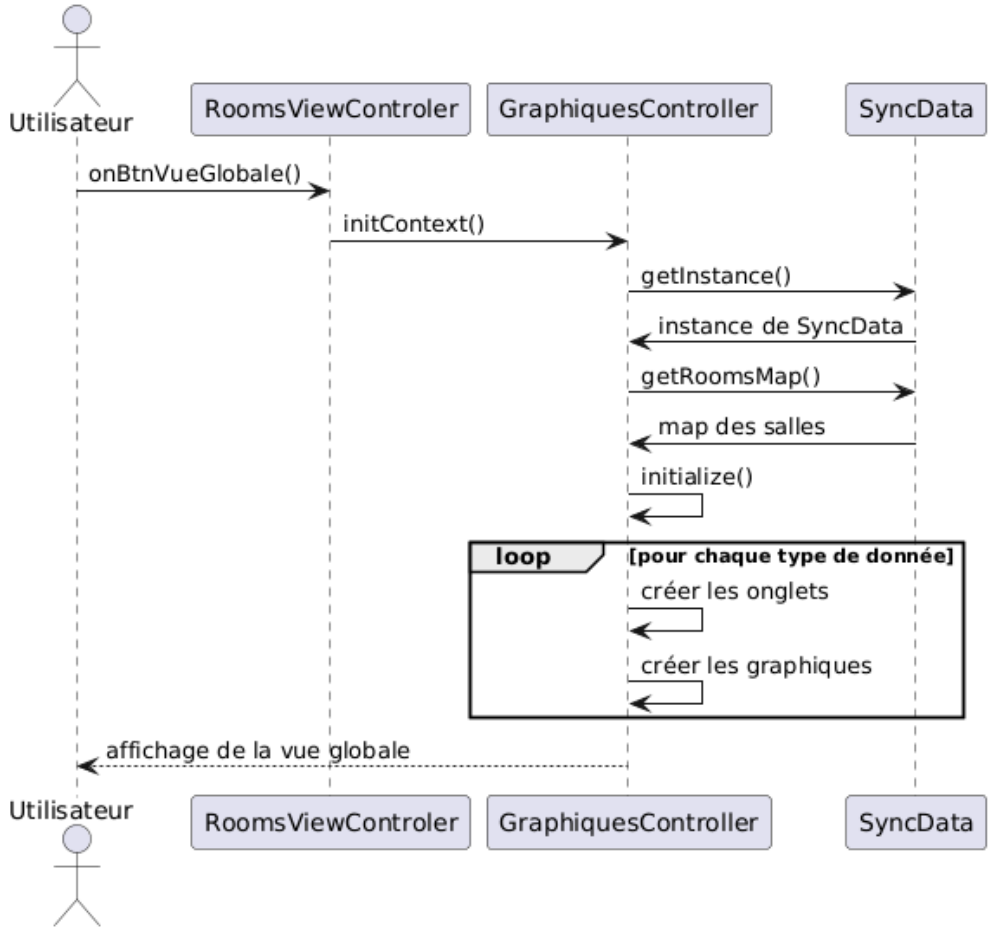
Ce diagramme de séquence illustre la fonctionnalité "Affichage mesures d'alertes" dans l'application. Lorsque l'utilisateur clique sur le bouton "Mesures d'Alerte", le contrôleur RoomsViewControler initialise le contrôleur AlertPageViewController. Ce dernier récupère les données des salles via SyncData, configure les alertes pour chaque salle, crée une liste d'affichage, configure la ListView, et affiche finalement la page des alertes à l'utilisateur. Le processus inclut une boucle pour traiter chaque salle et ses alertes associées, assurant ainsi une mise à jour complète des informations affichées.

4.3 Barre de Recherche



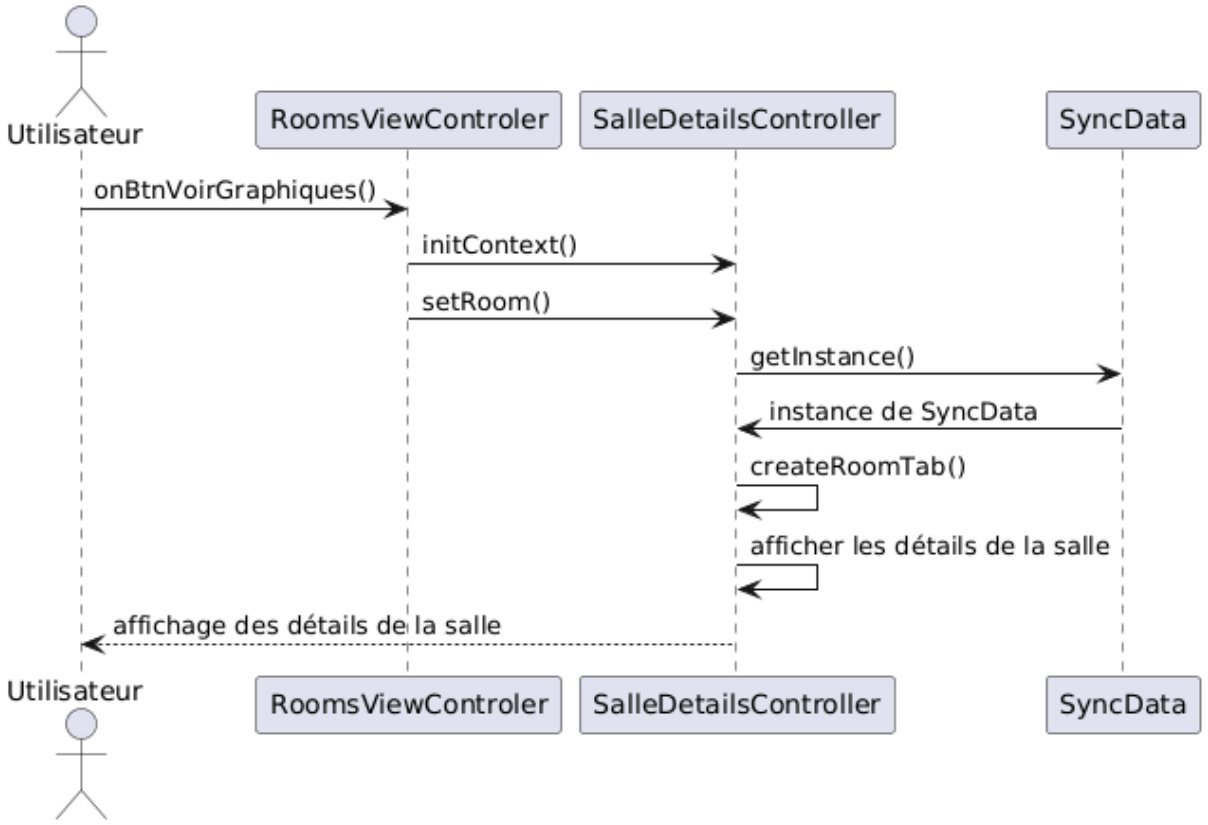
Ce diagramme de séquence illustre la fonctionnalité "Rechercher une salle" dans l'application. Lorsque l'utilisateur clique sur le bouton "Rechercher", le contrôleur RoomsViewControler récupère le texte saisi dans le champ de recherche (TextField). Ce texte est ensuite converti en minuscules pour une recherche non sensible à la casse. Le contrôleur obtient ensuite l'instance de SyncData pour accéder à la liste des salles. Après avoir récupéré cette liste, RoomsViewControler filtre les salles en fonction du texte de recherche. Enfin, il met à jour la ListView avec les salles filtrées, permettant à l'utilisateur de voir uniquement les salles correspondant à sa recherche.

4.4 Affichage Graphiques Vue globale



Ce diagramme montre les interactions principales entre les composants de l'application pour la fonctionnalité "Vue Globale". Lorsque l'utilisateur clique sur le bouton "Vue Globale", le contrôleur RoomsViewControler initialise le contrôleur GraphiquesController. Ce dernier récupère les données des salles via SyncData, initialise les données en créant des onglets pour chaque type de donnée et en générant les graphiques correspondants. Enfin, la vue globale des données sous forme de graphiques est affichée à l'utilisateur.

4.5 Affichage Graphiques d'une salle



Ce diagramme de séquence illustre la fonctionnalité "Affichage des données d'une salle" dans l'application. Lorsque l'utilisateur clique sur le bouton "Voir Graphiques", le contrôleur RoomsViewControler initialise le contrôleur SalleDetailsController et lui transmet la salle sélectionnée. SalleDetailsController récupère ensuite l'instance de SyncData pour accéder aux données nécessaires. Il crée un onglet pour afficher les graphiques de la salle. Ce processus permet de visualiser les informations spécifiques à une salle, y compris les graphiques et les alertes associées.

4.6 Page d'accueil

La page d'accueil est la première page que l'utilisateur voit lorsqu'il lance l'application.

Ses fonctionnalités sont les suivantes :

- Test de connexion à MQTT
- Possibilité de lancer l'application
- Manuel d'utilisation
- Crédits

4.6.1 Test de connexion à MQTT

Le test de connexion à MQTT permet de vérifier si l'application est bien connectée au serveur MQTT. Pour cela, l'application va essayer de se connecter au serveur MQTT. Si la connexion est réussie, un message de succès apparaît. Sinon, un message d'erreur apparaît.

Le code de la méthode testerConnexion() consiste à exécuter le script python principal avec un certain paramètre (conTest). Quand le script python est exécuté avec ce paramètre il renvoie tout de suite 0 si la connexion MQTT fonctionne et 1 si ce n'est pas le cas.

Ensuite on a juste à vérifier dans la méthode actionTesterCo() quelle est la valeur de retour et afficher une alerte en fonction de cette dernière. Cette deuxième méthode est reliée à un bouton utilisateur.

4.6.2 Possibilité de lancer l'application

La méthode actionLancer() permet de lancer l'application. En cliquant sur le bouton "Lancer l'application", l'utilisateur est redirigé vers la page de configuration. Pour que l'application se lance il faut que le test de connexion soit effectué.

4.6.3 Manuel d'utilisation

La méthode actionManuel permet d'ouvrir la fenêtre du manuel d'utilisation qui permet de comprendre l'application et ses fonctionnalités.

Il est expliqué qu'il faut tester la connexion avant de pouvoir lancer l'application, comment tester la connexion, ou encore certaines contraintes à respecter pour ne pas entraver la connexion (ne pas être connecté à eduroam par exemple).

4.6.4 Crédits

La méthode actionCredits() ouvre une fenêtre des crédits qui donne simplement les noms des auteurs de cette application.

4. Pages de configuration

Afin de parler de cette page nous allons parler des 4 classes qui sont reliées au bon fonctionnement de ces 2 pages.

4.7.1 configViewController

- **Package** : `application.view`

La classe **configViewController** gère l'interface de configuration principale de l'application. C'est l'endroit où l'utilisateur peut ajuster les paramètres essentiels tels que l'hôte MQTT, la fréquence de sauvegarde, et les abonnements aux capteurs. Elle facilite également la navigation vers d'autres parties de l'application, comme la gestion des seuils d'alerte.

Rôle principal

Cette classe connecte **l'interface utilisateur** à la logique métier. Elle s'assure que **les paramètres saisis ou modifiés sont enregistrés correctement** et vérifie la cohérence des données.

Ce qu'elle fait

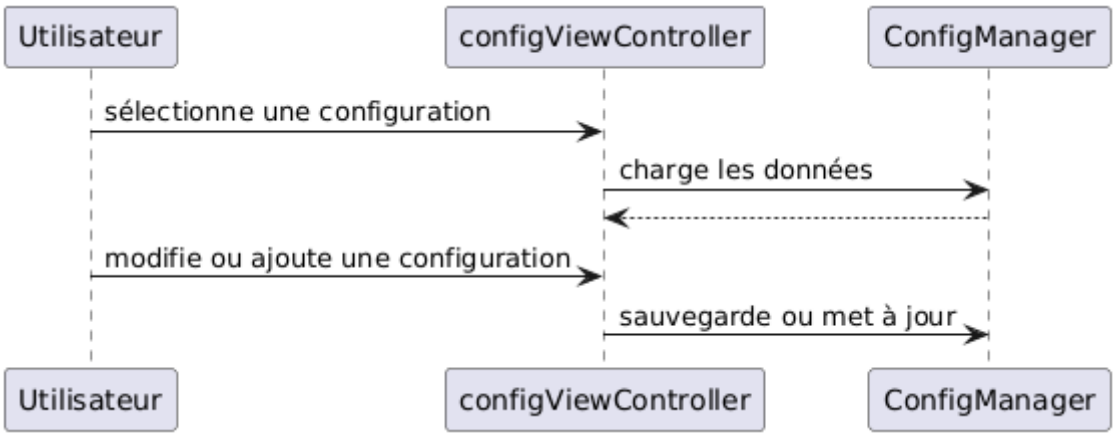
1. **Initialisation** :
    - Charge les données existantes grâce à **ConfigManager**.
    - **Pré-remplit** les champs (hôte, fréquence, etc.) avec les valeurs actuelles ou par défaut.
    - Prépare les listes de sélection pour **les salles** et **les types de données**.
2. **Gestion des abonnements** :
    - Active ou désactive **l'abonnement global aux capteurs** ou **panneaux solaires**.
    - Permet une sélection fine des **salles** ou des **types de données spécifiques**.
3. **Validation et navigation** :
    - Vérifie que **toutes les données nécessaires sont présentes et valides** avant de permettre **l'enregistrement** ou **la navigation**.

- Permet de **sauvegarder les modifications** ou de revenir à une vue précédente avec des avertissements en cas de données non sauvegardées.

Pourquoi elle est importante

Elle centralise **la gestion de la configuration** dans une interface intuitive. De plus, grâce à sa structure, elle est facilement extensible si de nouveaux paramètres doivent être ajoutés.

Diagramme de séquence :



4.7.2 seuilsViewController

- **Package** : `application.view`

La classe **seuilsViewController** est conçue pour gérer les seuils d'alerte associés aux capteurs de Ludorama. Elle permet de personnaliser les limites minimum et maximum pour des données comme le CO2 ou la température.

Rôle principal

Elle offre à l'utilisateur **un contrôle précis sur les alertes générées par les capteurs**, afin de répondre aux besoins spécifiques des environnements surveillés.

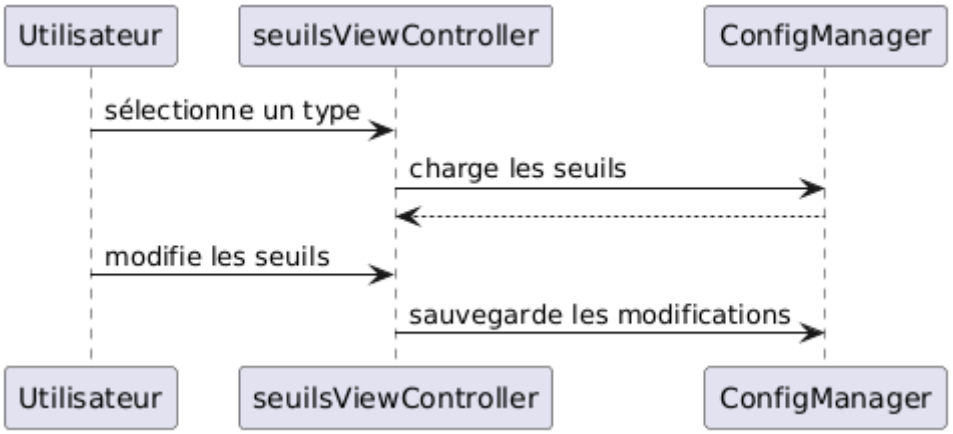
Ce qu'elle fait

1. **Chargement des seuils** :
    - Récupère **les valeurs actuelles depuis le fichier de configuration** pour les afficher dans l'interface.
2. **Mise à jour des seuils** :
    - Vérifie que **les valeurs saisies sont cohérentes** (par exemple, le seuil maximum doit être supérieur ou égal au seuil minimum).
    - Enregistre **les nouvelles valeurs dans le fichier de configuration**.
3. **Navigation fluide** :
    - Permet de **revenir à la vue de configuration** tout en gérant les éventuelles **modifications non sauvegardées**.

Pourquoi elle est importante

En permettant une gestion fine des seuils, cette classe contribue à **la réactivité de l'application** en cas de dépassement des valeurs limites.

Diagramme de séquence :



4.7.3 ConfigManager

- **Package** : `application.config`

La classe **ConfigManager** est **le chef d'orchestre** des interactions avec le fichier de configuration. Elle est essentielle pour lire, écrire, et structurer les données qui font fonctionner l'application.

Rôle principal

Centraliser **la gestion de la configuration** dans un module autonome et réutilisable, déconnecté de l'interface utilisateur.

Ce qu'elle fait

1. **Chargement et sauvegarde** :
    - Lit **les données depuis le fichier `config.ini`**.
    - **Sauvegarde** les modifications dans une structure claire, avec des commentaires pour faciliter la lecture humaine.
2. **Gestion des valeurs par défaut** :

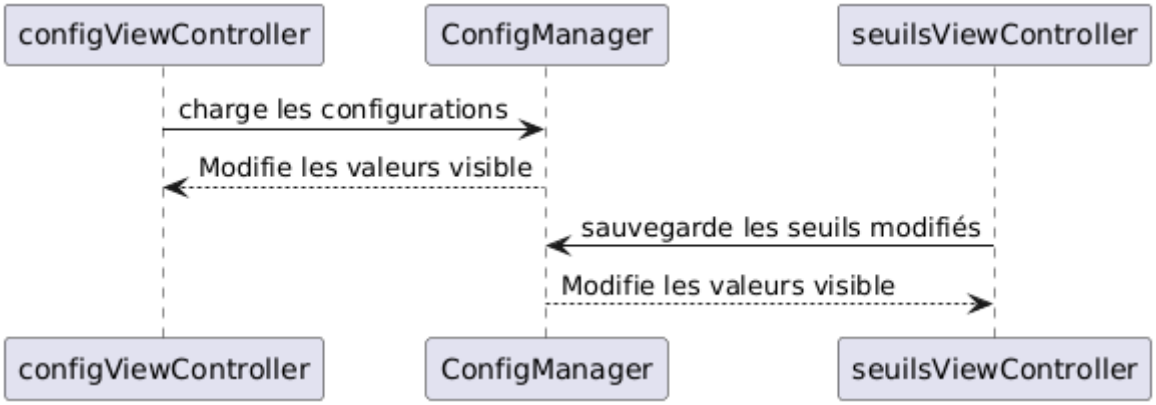
- Crée un fichier de configuration avec des valeurs initiales si celui-ci n'existe pas.

3. **Lecture et mise à jour** :
   - Fournit des méthodes simples pour **lire ou modifier un paramètre** sans avoir à manipuler directement le fichier.

Pourquoi elle est importante

Grâce à cette classe, toutes les interactions avec le fichier de configuration sont centralisées, ce qui garantit la cohérence et facilite la maintenance.

Diagramme de séquence :



4.7.4 AppConstants

- **Package** : `application.config`

La classe **AppConstants** regroupe toutes les constantes globales nécessaires au bon fonctionnement de l'application. Elle est comme une boîte à outils statique contenant des listes et des paramètres fixes.

Rôle principal

Centraliser les données statiques et éviter les répétitions ou incohérences dans le code.

Ce qu'elle contient

1. **Liste des salles** :
    - Une liste complète des salles équipées de capteurs.
2. **Types de données** :
    - Les types de données mesurés par les capteurs (CO2, température, humidité, etc.).

Pourquoi elle est importante

En centralisant ces informations dans une seule classe, on réduit les risques d'erreurs et on rend le code plus facile à maintenir et à modifier.

En résumé

Ces quatre classes collaborent étroitement pour fournir **une application fluide et bien structurée**.

- **configViewController** et **seuilsViewController** se concentrent sur **l'interface utilisateur**.
- **ConfigManager** et **AppConstants** fournissent la base logique et les données nécessaires.

Cette organisation facilite l'ajout de nouvelles fonctionnalités et la maintenance à long terme.

5. Installation

5.1 Installation pour utilisateur

L'application JavaFX avec intégration IoT est fournie sous la forme d'un fichier exécutable .jar. Pour utiliser l'application :

Pré-requis : Aucun, Python est inclus dans les ressources internes de l'application, ce qui garantit une exécution fluide sans installation supplémentaire.

Étapes :

- Téléchargez le fichier .jar (rendu sur Webetud)
- Lancez l'application en double-cliquant sur le fichier .jar

Aucune configuration spécifique n'est nécessaire. L'ensemble des dépendances et des scripts Python sont automatiquement gérés en interne, même python est installé en interne.

5.2 Installation environnement de développement

Pour avoir accès au code développé pour analyser ou exécuter, voici les étapes et les pré-requis nécessaire :

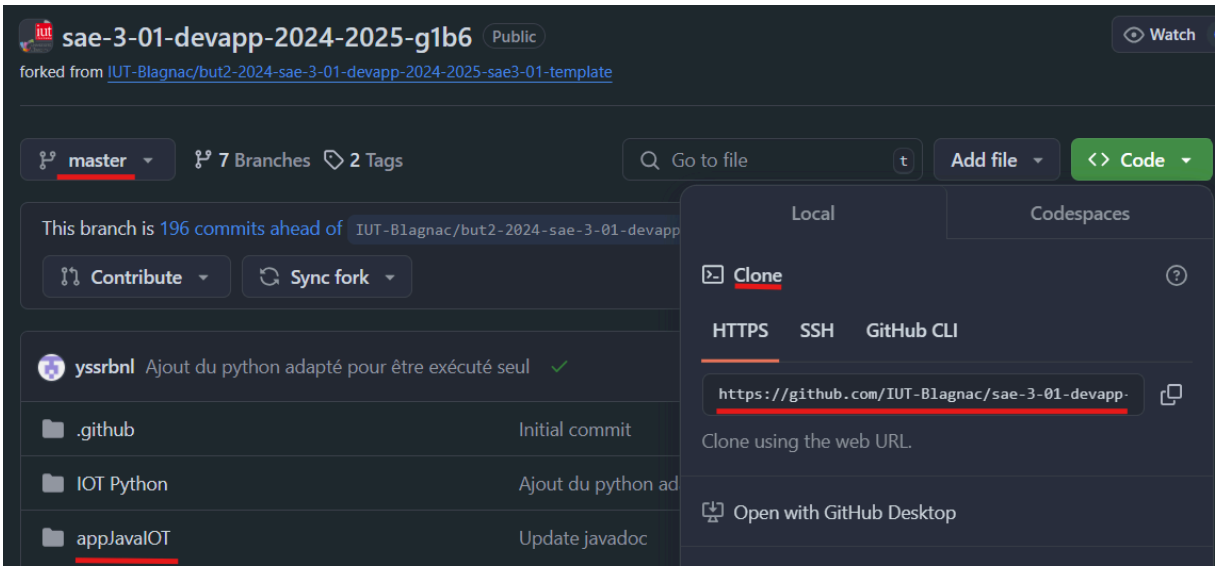
Pré-requis :

- Java Development Kit (JDK) version 11 ou supérieure.
- Un IDE compatible avec Maven, tel que :
    1. IntelliJ IDEA (recommandé) : Chargez simplement le projet en tant que projet Maven.
    2. Visual Studio Code : Assurez-vous que l'extension Maven est installée et active.
- Tout autre IDE prenant en charge Maven.

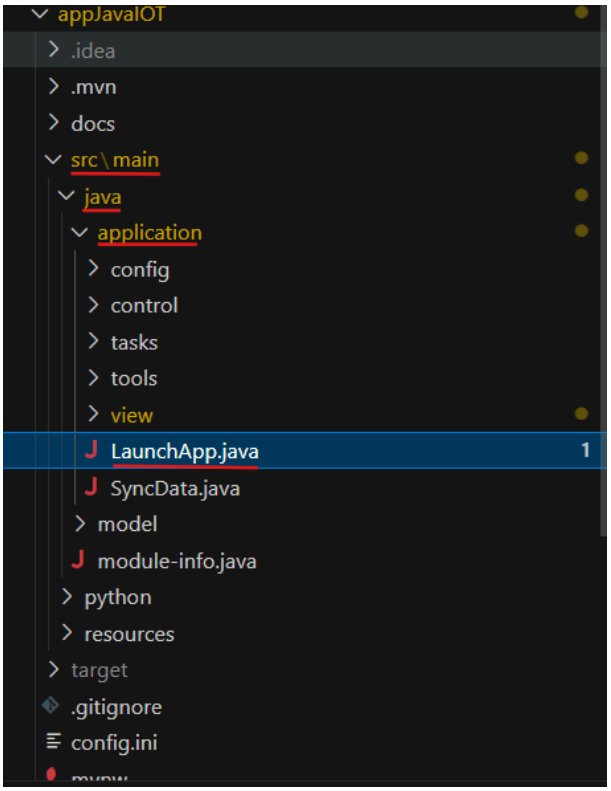
Étapes :

- Clonez ou téléchargez le projet depuis le dépôt sur la branche master.
- Ouvrez le projet appJavaIOT dans votre IDE.
- Configurez les dépendances Maven si nécessaire (la plupart des IDE le font automatiquement).
- Lancez l'application ou effectuez vos modifications selon vos besoins.

Etat du github pour le clonage :



Architecture du projet (chemin rouge jusqu'à l'application exécutable) :



Cette configuration assure une compatibilité optimale pour le développement, le débogage et l'ajout de nouvelles fonctionnalités, tout en offrant un cadre structuré pour la gestion des dépendances grâce à Maven. Ainsi, les développeurs peuvent facilement collaborer et maintenir l'intégrité du projet tout au long de son cycle de vie.