

SAE 3.01 Partie IOT

(Groupe 12)

Documentation Java

Sommaire :

Présentation de l'application	3
Contexte général	3
Description du besoin	3
Cas d'utilisation	4
Tutoriel d'installation	5
Installer Python	5
Installer Java	6
Lancer l'application	7
Structure du projet	8
Architecture général	8
Ressources Utilisées	9
Packages et classes Java	9
Structure des autres fichiers	16
Spécificités du code	16
Fonctionnalités développées	20
Paramétrer le fichier de configuration	20
Visualiser l'historique de logs	25
Visualiser les données en temps réel	28
Recevoir des alertes	32

Présentation de l'application

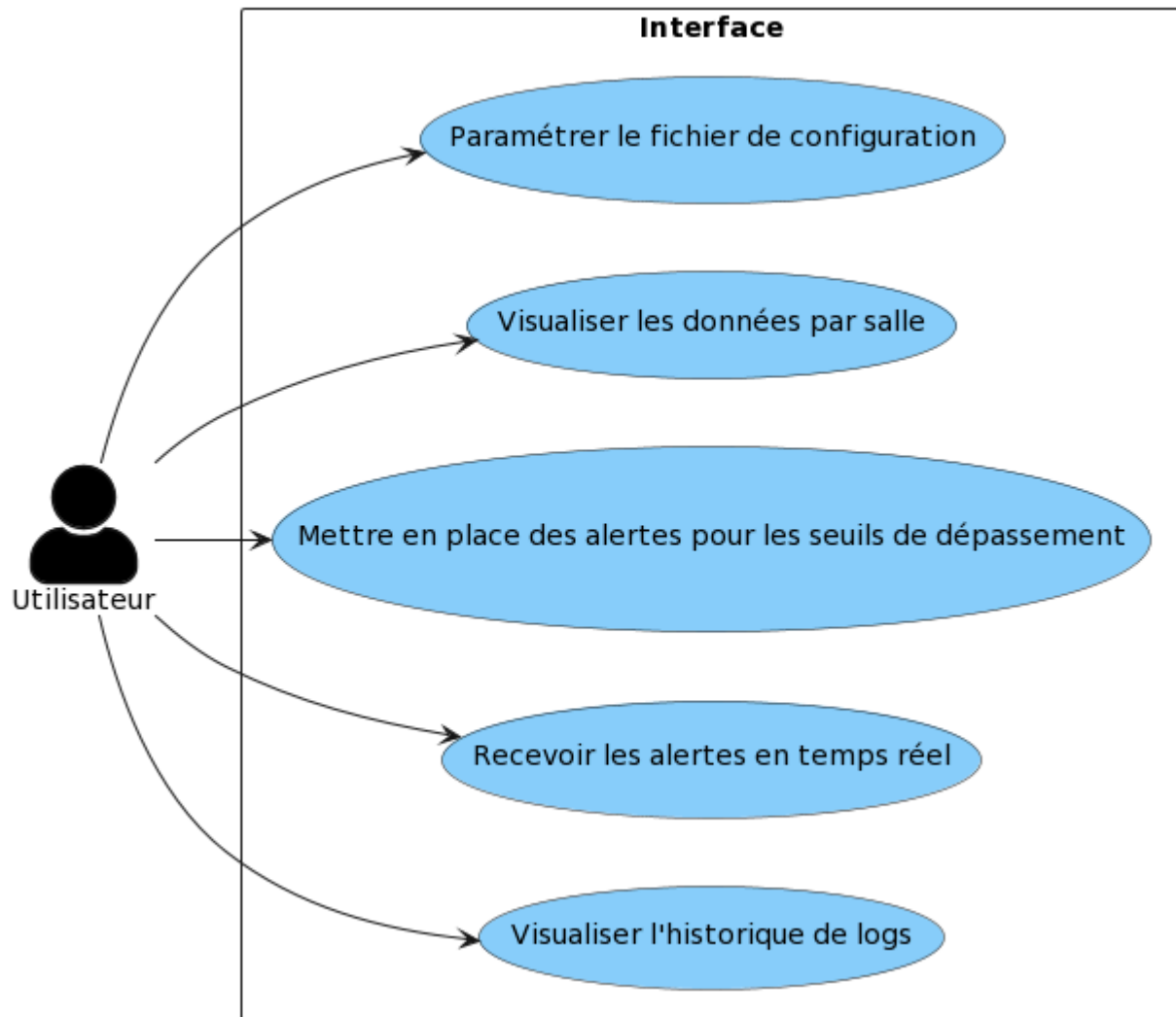
Contexte général

L'entreprise "La Bonne Note", spécialisée dans la vente d'instruments de musique, ambitionne d'améliorer la gestion de son entrepôt par le biais d'une application Java. Cette dernière sera conçue pour optimiser l'accès aux données des capteurs, l'écriture de ces informations dans des fichiers, et pour paramétrer le fonctionnement d'une application Python associée. L'objectif est de permettre une gestion précise et efficace des capteurs déployés à l'IUT, diffusant des données relatives au CO2, à la température et à l'humidité via le bus MQTT. L'application s'adapte automatiquement à tous les formats d'écrans et plusieurs options de tri / recherches sont possibles, il est également possible d'ouvrir et d'isoler un certain type de données dans une nouvelle fenêtre.

Description du besoin

L'application Java de "La Bonne Note" vise à fournir une interface ergonomique et fonctionnelle pour gérer les capteurs et les données qu'ils génèrent. Elle propose différentes fonctionnalités telles que la configuration des types de données récupérées, la fréquence de lecture des capteurs, les seuils d'alerte, et les noms de fichiers associés. Cette application Java servira également à afficher en temps réel les données captées, différenciées par salle pour offrir une représentation terrain réaliste. Elle permet également la visualisation graphique des fichiers logs, l'affichage des alertes de dépassement de seuils, tout en mettant l'accent sur une architecture logicielle proposée par les étudiants, justifiée et analysée en profondeur.

Cas d'utilisation



L'application propose plusieurs cas d'utilisation essentiels pour optimiser la gestion de son entrepôt. Tout d'abord, elle permet la configuration des capteurs en offrant la possibilité de sélectionner les types de données à récupérer, de définir la fréquence de lecture des capteurs et de fixer les seuils d'alerte pour chaque type de donnée. Ensuite, elle offre une interface conviviale pour afficher en temps réel les données captées par les capteurs, distinguant ces données par salle pour une représentation précise de l'état

actuel de l'entrepôt. De plus, elle facilite la visualisation graphique des fichiers logs, permettant aux utilisateurs de consulter l'historique des données captées. Enfin, elle assure une gestion proactive des alertes en affichant en permanence les dépassements de seuils détectés par les capteurs, fournissant ainsi un aperçu immédiat des situations critiques et nécessitant une attention particulière. A la demande du client, il n'y aura qu'un seul type d'utilisateur sur l'application qui aura directement tous les droits.

Tutoriel d'installation

Installer Python

Pour vérifier si votre système dispose de Python :

- ouvrir un terminal
- lancer la commande **Python -version**

Si la version de Python apparaît, vous pouvez sauter l'étape d'installation de Python.

Si Python n'est pas installé, vous pouvez le télécharger depuis <https://www.python.org/downloads/>.

Pour lancer le script Python, assurez-vous d'avoir les bibliothèques requises installées. Voici les bibliothèques nécessaires :

paho.mqtt.client : bibliothèque MQTT pour Python.

json : module Python pour travailler avec JSON.

configparser : module Python pour lire les fichiers de configuration.

os : module Python pour des fonctionnalités liées au système d'exploitation.

time (sous Windows uniquement) : module Python pour le temps.

datetime : module Python pour manipuler les dates et heures.

Pour installer les bibliothèques Python, ouvrez une invite de commande ou un terminal et saisissez les commandes suivantes :

- **pip install paho-mqtt**
- **pip install jsonlib-python3**
- **pip install configparser**
- **pip install datetime**

Installer Java

Pour exécuter l'application Java, vérifiez d'abord que Java est installé sur votre ordinateur en ouvrant une invite de commande et en saisissant la commande suivante :

- **java -version**

Si la version de Java apparaît et qu'il s'agit bien de la version 17, vous pouvez sauter l'étape d'installation de Java.

Si JAVA n'est pas installé, vous pouvez le télécharger ici, veillez à bien choisir la version 17 ainsi que la version adaptée pour votre système d'exploitation.

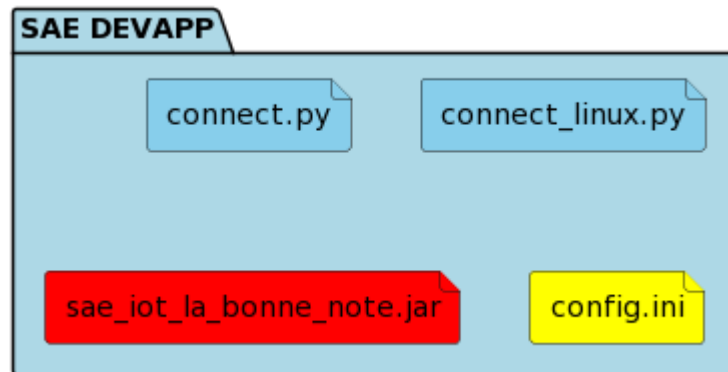
Pour pouvoir modifier et compiler le code, il vous faudra installer Maven en ayant ajouté la variable d'environnement comme dans ce [tutoriel](#).

Lancer l'application

Pour lancer l'application :

Télécharger tous les fichiers du dossier de [l'application finale](#) comprenant le .JAR (exécutable), les fichiers Python et le fichier de configuration. Ces fichiers sont disponibles au même endroit que le dépôt de cette documentation sur WebEtud.

Après avoir téléchargé les fichiers, vérifier que tous les fichiers soient au même endroit dans l'arborescence.



(En bleu les script python, en rouge l'exécutable de l'application et en jaune le fichier de configuration)

Deux méthodes pour lancer l'application :

Ouvrez une invite de commande et exécutez la commande suivante :

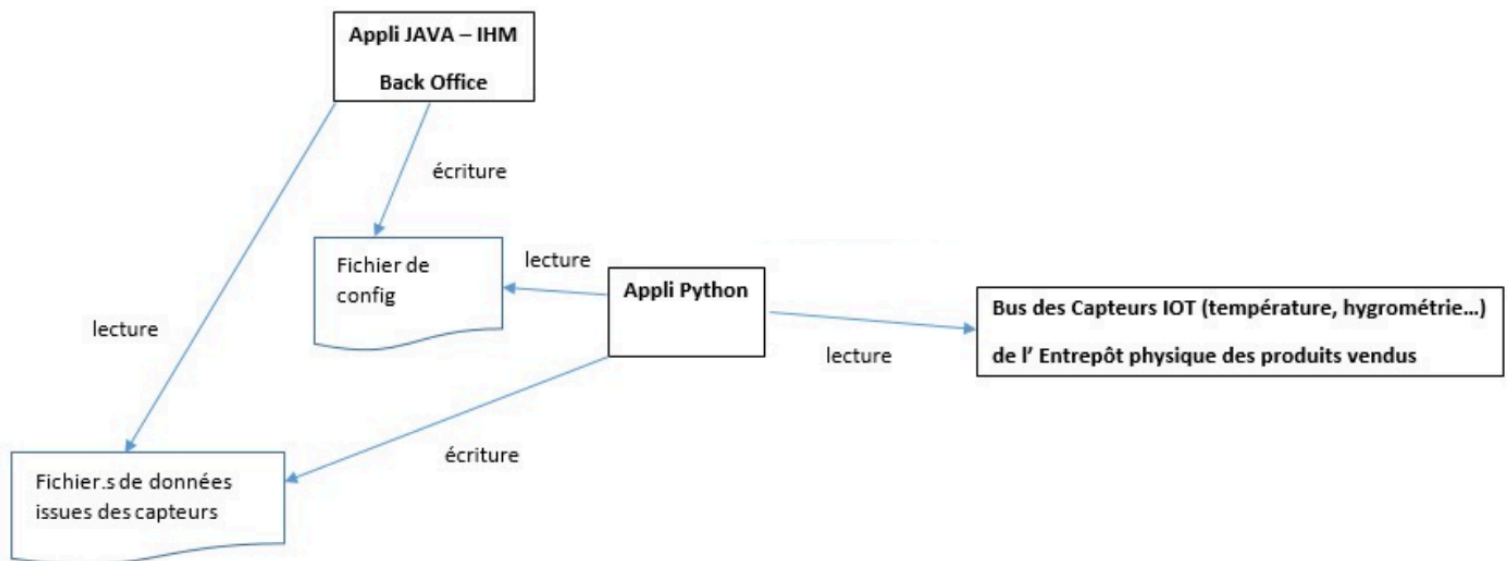
- **java -jar sae_iot_la_bonne_note.jar**

Double-cliquez sur le fichier exécutable (sae_iot_la_bonne_note.jar)
Si vous préférez exécuter le projet à partir d'Eclipse, veuillez installer le JDK 17. De plus, l'installation de JavaFX depuis Eclipse Marketplace est nécessaire (version recommandée : 3.8.0).

Si vous êtes sur Linux, il faut supprimer le script Python **connect.py**, renommer le fichier **connect_linux.py** en **connect.py**.

Structure du projet

Architecture général



Le fichier Python récupère la configuration qui sera stockée dans un fichier de configuration qui lui sera modifiable à travers l'application Java, le script Python récupère les données des capteurs et les écrit dans des fichiers de données qui seront eux lus par l'application Java.

Ressources Utilisées

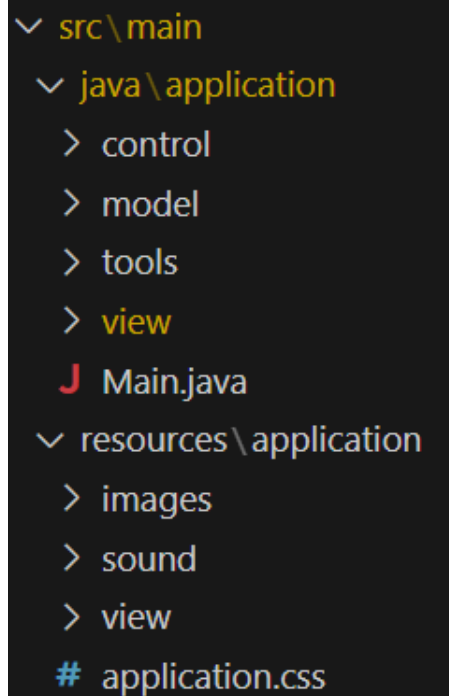
Maven : Utilisé pour la compilation, la création du JAR, la génération de la JavaDoc et la gestion des dépendances via le fichier pom.xml. Lien vers [Maven](#).

JDK : Version 17 nécessaire pour l'exécution du projet. Site du [JDK](#).

Fichiers FXML : Créés avec SceneBuilder pour les scènes de l'application. Téléchargeable [ici](#).

L'application a été développée avec [Visual Studio Code](#).

Packages et classes Java



```

  ✓ src\main
    ✓ java\application
      > control
      > model
      > tools
      > view
      J Main.java
    ✓ resources\application
      > images
      > sound
      > view
      # application.css

```

Vue globale sur l'arborescence de l'application.

Justification du Choix du Pattern MVC :

Le choix du pattern Modèle-Vue-Contrôleur (MVC) pour le développement de cette application repose sur sa capacité à fournir une structure organisée et modulaire. En adoptant MVC, nous séparons clairement la logique métier (Modèle) de l'interface utilisateur (Vue) et des interactions utilisateur (Contrôleur). Cette séparation des responsabilités améliore la maintenabilité du code, facilite les modifications et extensions futures, et encourage la réutilisabilité des composants. En outre, MVC offre une clarté architecturale, simplifiant le processus de développement et permettant une évolution plus fluide de l'application au fil du temps. Ce choix vise à créer une base solide pour garantir la robustesse, la flexibilité et la pérennité de l'application.

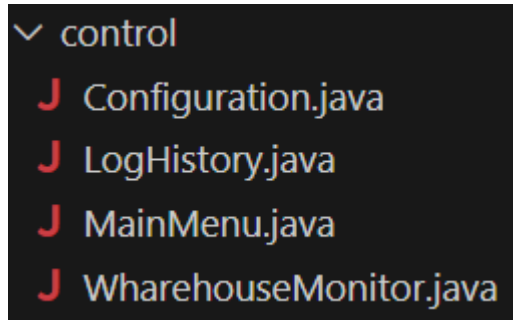
application : comporte le main permettant de lancer l'application ainsi que la classe qui permet de connaître l'état de l'application

Classes :

Main.java : Classe principale permettant de lancer l'application.

application.control : contient les contrôleurs et l'accès aux données, regroupés dans le package "application.control", sont responsables de la gestion des fonctionnalités de l'application

Classes :



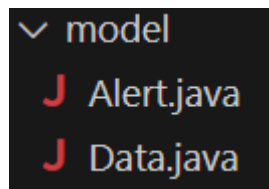
Configuration.java : gère la récupération et l'affichage de la scène de configuration

LogHistory.java : gère la récupération et l'affichage de la scène de l'historique

MainMenu.java : gère la récupération et l'affichage de la scène du menu principal

WharehouseMonitor.java : gère la récupération et l'affichage de la scène des données en temps réel

application.model :



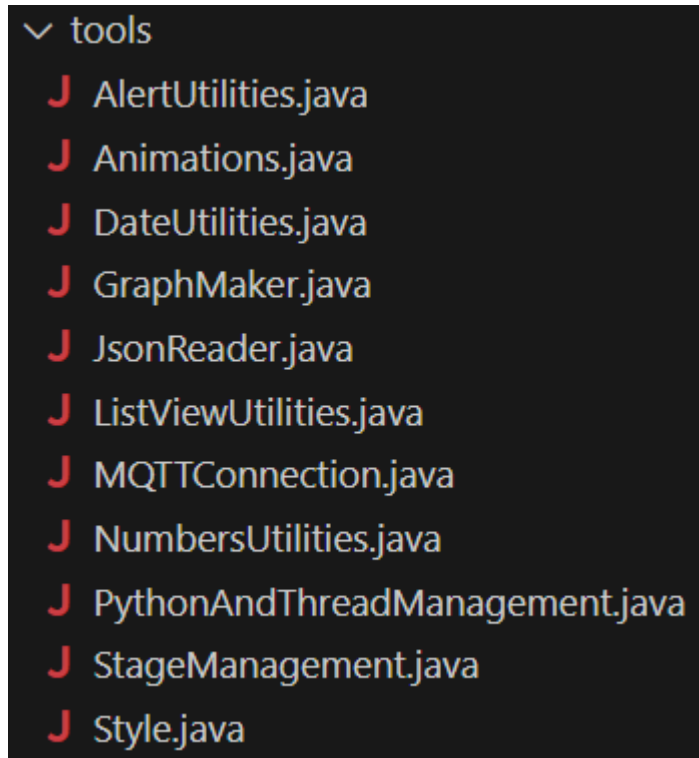
Classes :

Alert.java : représente une alerte en fonction des seuils définis pour la température, l'humidité, l'activité et le CO2

Data.java : représente les données associées à une mesure spécifique.

application.tools : fournit des outils d'aide supplémentaires facilitant ainsi le développement et la maintenance de l'application

Classes :



AlertUtilities.java : fournit des méthodes pour afficher différents types d'alertes

Animations.java : permet de créer des animations sur des Node javafx

DateUtilities.java : fournit des méthodes utilitaires pour manipuler les dates et les formats de date

GraphMaker.java : fournit des méthodes pour manipuler et gérer les graphiques JavaFX

JsonReader.java : utilitaire pour la lecture et la manipulation de fichiers JSON

ListViewUtilities.java : utilitaire pour la configuration des cellules dans une ListView

MQTTConnection.java : utilitaire pour la gestion des connexions MQTT

NumbersUtilities.java : utilitaire pour la conversion des chaînes en nombres

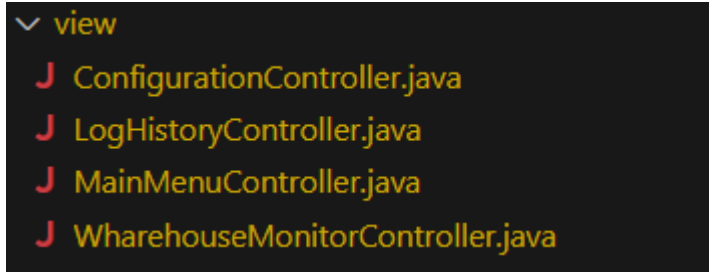
PythonAndThreadManagement.java : gère le démarrage, l'arrêt et la surveillance des processus Python et la mise à jour des données via des threads

StageManagement.java : utilitaire pour centrer automatiquement une fenêtre sur une autre

Style.java : fournit des méthodes pour gérer et appliquer des styles personnalisés ainsi que pour manipuler des icônes dans une interface utilisateur JavaFX

application.view : contient les vues de l'application, gérée par le package "application.view", est basée sur des fichiers FXML et leurs contrôleurs associés. Chaque page FXML possède sa propre vue, garantissant une interface utilisateur claire et intuitive

Classes :



```
view
├── ConfigurationController.java
├── LogHistoryController.java
├── MainMenuController.java
└── WharehouseMonitorController.java
```
































ConfigurationController.java : gère les actions de la scène de configuration

LogHistoryController.java : gère les actions de la scène de l'historique

MainMenuController.java : gère les actions de la scène du menu principal

WharehouseMonitorController.java : gère les actions de la scène des données en temps réel

Le dossier ressources contient tous les éléments nécessaires à l'application (images, sons, scènes FXML et le CSS, un seul fichier css est utilisé pour toute l'application).

- ▼ resources\application
 - ▼ images
 -  alert_data_icon.png
 -  alerte_icon.png
 -  bar-chart_icon.png
 -  choix_icon.png
 -  connection_fail.png
 -  connexion_icon.png
 -  deleteFileIcon.png
 -  entrepot_fond.jpg
 -  entrepot_logo.png
 -  failed_icon.png
 -  frequency_icon.png
 -  historique_logo.png
 -  info_icon.jpg
 -  la_bonne_note_logo.png
 -  leave_icon.png
 -  line-chart_icon.png
 -  listview_icon.png
 -  loading_icon.jpg
 -  menu_icon.png
 -  reset_icon.png
 -  save_icon.png
 -  search_icon.png
 -  settings_icon.png
 -  sound_icon.png
 -  success_icon.png
 -  undefined_icon.png
 - ▼ sound
 -  alert_sound.mp3
 - ▼ view
 -  Configuration.fxml
 -  LogHistory.fxml
 -  MainMenu.fxml
 -  WharehouseMonitor.fxml
 - # application.css

Structure des autres fichiers

Fichiers Python :

- **connect_linux.py** : version Linux du script.
- **connect.py** : version Windows du script.

Fichiers de données JSON :

- un fichier Json stockant tout l'historique des alertes
- un fichier Json stockant tout l'historique des données
- un fichier Json stockant juste les données récupérées en temps réel (fichier qui est remis à vide à chaque lancement du script)

Spécificités du code

3 Threads sont utilisées dans différentes parties de l'application :

1) Thread pour tester la connexion au serveur MQTT :

```
171 // Thread de test de connexion
172 private Thread connexionTestThread;
```



```

596  /**
597   * Initialise le thread de test de connexion MQTT.
598   * Ce thread vérifie la connexion au serveur MQTT à l'aide des paramètres
599   * d'hôte et de port.
600   * Il met à jour l'IHM en conséquence, affichant une icône et des alertes en
601   * cas de succès ou d'échec de connexion.
602   */
603  private void initConnexionTestThread() {
604      // A l'intérieur du thread
605      connexionTestThread = new Thread(() -> {
606          try {
607              isTestRunning = true;
608              // Désactive les éléments de configuration minimale pendant le test
609              disableMinConfWhileTest(_disable:true);
610              Style.setNewIcon(imgConnexion, _imageName:"loading_icon.jpg");
611              loadingIconAnimation = Animations.startLoadingAnimation(imgConnexion);
612              // Effectue le test de connexion MQTT
613              isConnected = MQTTConnection.testMQTTConnection(host, port);
614
615              // Réactive les éléments de configuration minimale après le test
616              disableMinConfWhileTest(_disable:false);
617
618              // Mise à jour de l'IHM après la fin du test (à l'intérieur de
619              // Platform.runLater())
620              Platform.runLater(() -> {
621                  loadingIconAnimation.stop();
622                  if (isConnected) {
623                      Style.setNewIcon(imgConnexion, _imageName:"success_icon.png");
624                      AlertUtilities.showAlert(primaryStage, _title:"Connexion établie.", _message:"Connexion réussie !",
625                          _content:"La connexion au serveur MQTT a été établie.", AlertType.INFORMATION);
626                  } else {
627                      Style.setNewIcon(imgConnexion, _imageName:"failed_icon.png");
628                      AlertUtilities.showAlert(primaryStage, _title:"Échec de la connexion.", _message:"Échec de la connexion !",
629                          _content:"Veuillez saisir les bons paramètres du serveur MQTT.", AlertType.ERROR);
630                  }
631                  isTestRunning = false;
632              });
633          }

```

- dans la classe **ConfigurationController.java**, ce thread permet de tester si la connexion au serveur MQTT est réussie donc si les paramètres du serveur rentrés sont bons, ce thread s'arrête lors de la fin du test. Lancé avec la méthode **initConnexionTestThread()**.

2) Thread pour la mise à jour des données en temps réel :

```

165      private Thread updatesDataThread;

```

```

205  /**
206   * Initialise et lance un thread pour récupérer de nouvelles données à partir
207   * d'un fichier JSON.
208   * Ce thread vérifie périodiquement si le fichier JSON a été modifié et met à
209   * jour les données si nécessaire.
210   */
211  private void initGetNewDatasThread() {
212      updatesDataThread = new Thread(() -> {
213          long lastTime = 0;
214          long lastAlert = System.currentTimeMillis();
215          while (!Thread.currentThread().isInterrupted()) {
216              try {
217                  File jsonFile = new File(fileDataPath + ".json");
218                  long currentTime = jsonFile.lastModified();
219                  if (currentTime > lastTime) {
220                      Platform.runLater(() -> {
221                          if (jsonFile.exists()) {
222                              JsonReader.updateHistoryFromFile(primaryStage, _isCurrentDatas:true, _isAlertFile:false, obsList,
223                                  listAllRoomsDatas, _listAllRoomsAlerts:null, comboBoxRooms);
224                              if (currentTime > lastAlert) {
225                                  checkAlertForLastData();
226                              }
227                              updateDatasHistory();
228                          }
229                      });
230                      lastTime = currentTime;
231                  }
232              } catch (Exception e) {
233                  Thread.currentThread().interrupt();
234              }
235          }
236      });
237      updatesDataThread.setName(name:"getNewDatasThread");
238      updatesDataThread.start();
239  }

```

- dans la classe **WarehouseMonitorController.java**, ce Thread permet de mettre à jour les données à chaque nouvelle modification du fichier de données, il met ensuite à jour les graphiques (méthode **updateDatasHistory()**). Il gère aussi les alertes avec la méthode **checkAlertsForLastData()** qui vérifie chaque nouvelle donnée et s'il faut afficher une alerte ou non.

```

194      // Arrête et relance le thread de mise à jour des données pour éviter les
195      // doublons de ce Thread lors des changements de scène
196      PythonAndThreadManagement.stopThreadByName(_threadName:"getNewDatasThread");
197      initGetNewDatasThread();

```

Thread lancé à l'ouverture de la scène "Temps réel" avec la méthode **initGetNewDatasThread()**. Avant cet appel, on vérifie si ce Thread n'a pas déjà été lancé avec l'appel à **stopThreadByName()** qui arrête le Thread s'il est déjà lancé pour le relancer.

3) Thread pour lancer / arrêter le script Python :

```
19     private static Thread pythonThread;
20     private static Process pythonProcess;
21     private static ImageView imgConnexionState;
22     private static FadeTransition fdAnim;
```

```
188     PythonAndThreadManagement.initImgConnexionState(imgConnexionState);
189     if (!PythonAndThreadManagement.isPythonRunning()) {
190         PythonAndThreadManagement.startPythonThread(primaryStage);
191     }
192     PythonAndThreadManagement.updateImgConnexionState();
```

```
57     /**
58      * Démarre un nouveau thread pour exécuter le script Python.
59      *
60      * @param _primaryStage La fenêtre principale de l'application.
61      */
62     public static void startPythonThread(Stage _primaryStage) {
63         pythonThread = new Thread(() -> {
64             String scriptPath = "connect.py";
65             ProcessBuilder processBuilder = new ProcessBuilder(...command:"python", scriptPath);
66             try {
67                 pythonProcess = processBuilder.start();
68                 updateImgConnexionState();
69                 try {
70                     pythonProcess.waitFor();
71                 } catch (InterruptedException e) {
72                 }
73             } catch (IOException e) {
74                 Platform.runLater(() -> {
75                     AlertUtilities.showAlert(_primaryStage, _title:"Erreur",
76                                             _message:"Lancement du script Python impossible.",
77                                             "Une erreur est survenue lors du lancement du script Python."
78                                             + "...\nCode d'erreur : " + e,
79                                             AlertType.ERROR);
80                 });
81             }
82             updateImgConnexionState();
83         });
84         pythonThread.start();
85     }
```

- stockée dans la classe **PythonAndThreadManagement.java**, il est lancé au lancement de la scène “Temps réel” par la méthode

startPythonThread() qui prend le stage en paramètre pour afficher des alertes en cas d'exceptions rencontrées. Cette méthode est appelée seulement si le Thread Python n'est pas déjà lancé. La méthode va initialiser le processus Python et le lancer. Ce Thread sera à partir du moment où il est lancé actif sur toutes les scènes, donc l'affichages des alertes sera fait dans toute l'application.

```
719      /**
720       * Méthode de fermeture de la fenêtre par la croix.
721       *
722       * @param _e L'événement de fermeture de fenêtre.
723       */
724     private void closeWindow(WindowEvent _e) {
725         if (AlertUtilities.confirmYesCancel(primaryStage, _title:"Quitter l'application ?",
726             _message:"Voulez-vous vraiment quitter l'application ?", _content:null,
727             AlertType.CONFIRMATION)) {
728             closeLargeGraphsStages();
729             PythonAndThreadManagement.stopPythonThread();
730             primaryStage.close();
731             System.exit(status:0);
732         } else {
733             _e.consume();
734         }
735     }
```

Le Thread sera arrêté lors de la fermeture de la fenêtre en faisant appel à **stopPythonThread()**. Il sera relancé lors d'une nouvelle sauvegarde du fichier de configuration dans le but que la nouvelle configuration sauvegardée soit prise en compte par le script Python.

Fonctionnalités développées

Paramétrer le fichier de configuration

Vérifie les données captées par les capteurs pour chaque salle surveillée. Si une donnée dépasse le seuil prédéfini, l'application affiche instantanément une alerte correspondante.

Configuration

Menu

Temps Réel

Historique

Configuration

Host : chirpstack.iut-blagnac.fr

Port : 1883

Tester la connexion

Topic : #

Nom des fichiers : Alertes : alerte

Données : donnees

Logs : logs

Choix des données :

Température

Humidité

Activité

Co2

Fréquence d'affichage :

seconde(s)

0.0

Son des alertes :

Activé

Volume : 70.0

Seuils d'alertes :

Température : 10.0

Humidité : 50.0

Activité : 2.0

Co2 : 0.0

Réinitialiser

Sauvegarder

Partie du UseCase :

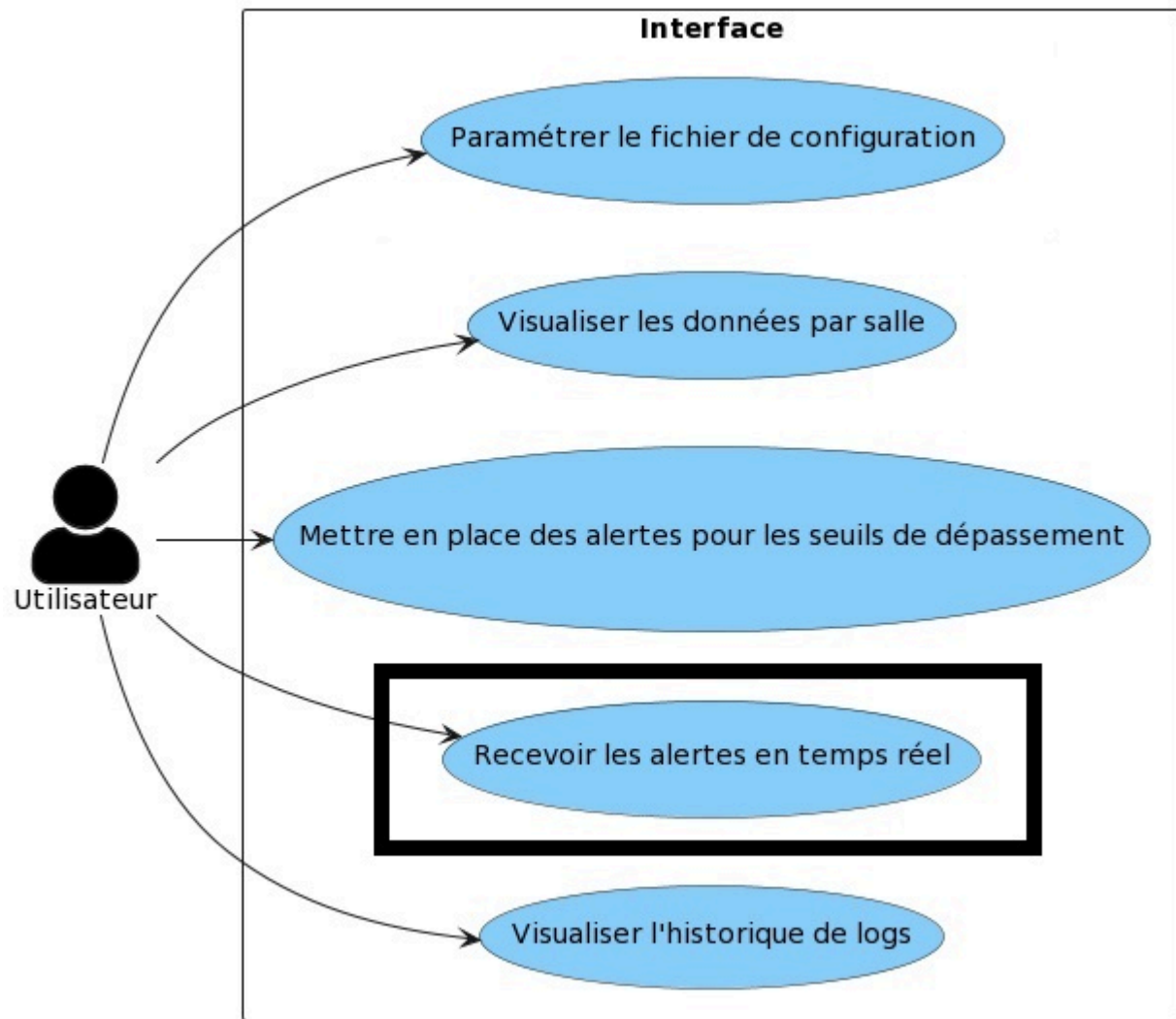
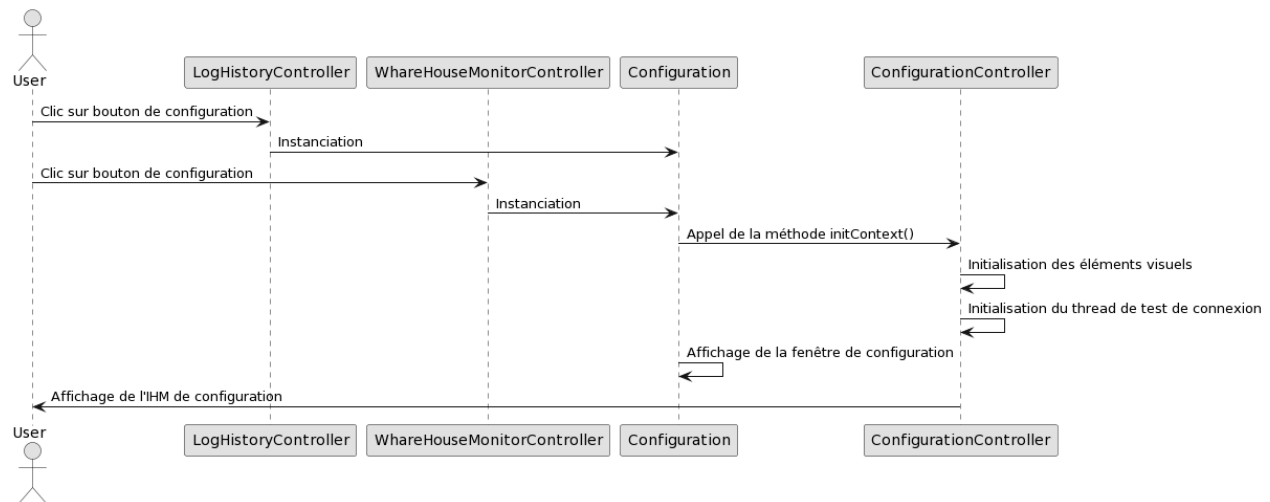


Diagramme de séquence :



Lorsqu'un utilisateur déclenche l'action en cliquant sur le bouton de configuration, le contrôleur de l'historique des logs (LogHistoryController) ou celui d'entrepôt (WhareHouseMonitorController) interagissent pour instancier la classe Configuration. Cette classe déclenche alors le contrôleur de configuration (ConfigurationController) via la méthode `initContext()`, permettant ainsi l'initialisation des éléments visuels de l'IHM et la mise en place des actions associées aux différents éléments graphiques de la fenêtre de configuration. Enfin, cette fenêtre de configuration est affichée, fournissant à l'utilisateur une interface pour configurer l'application.

Classes utilisées :

LogHistoryController.java : Contrôleur pour la gestion des historiques de logs.

WhareHouseMonitorController.java : Contrôleur pour surveiller l'entrepôt.

Configuration.java : Classe responsable de la fenêtre de configuration.

ConfigurationController.java : Contrôleur pour la fenêtre de configuration, gère les interactions et la logique.

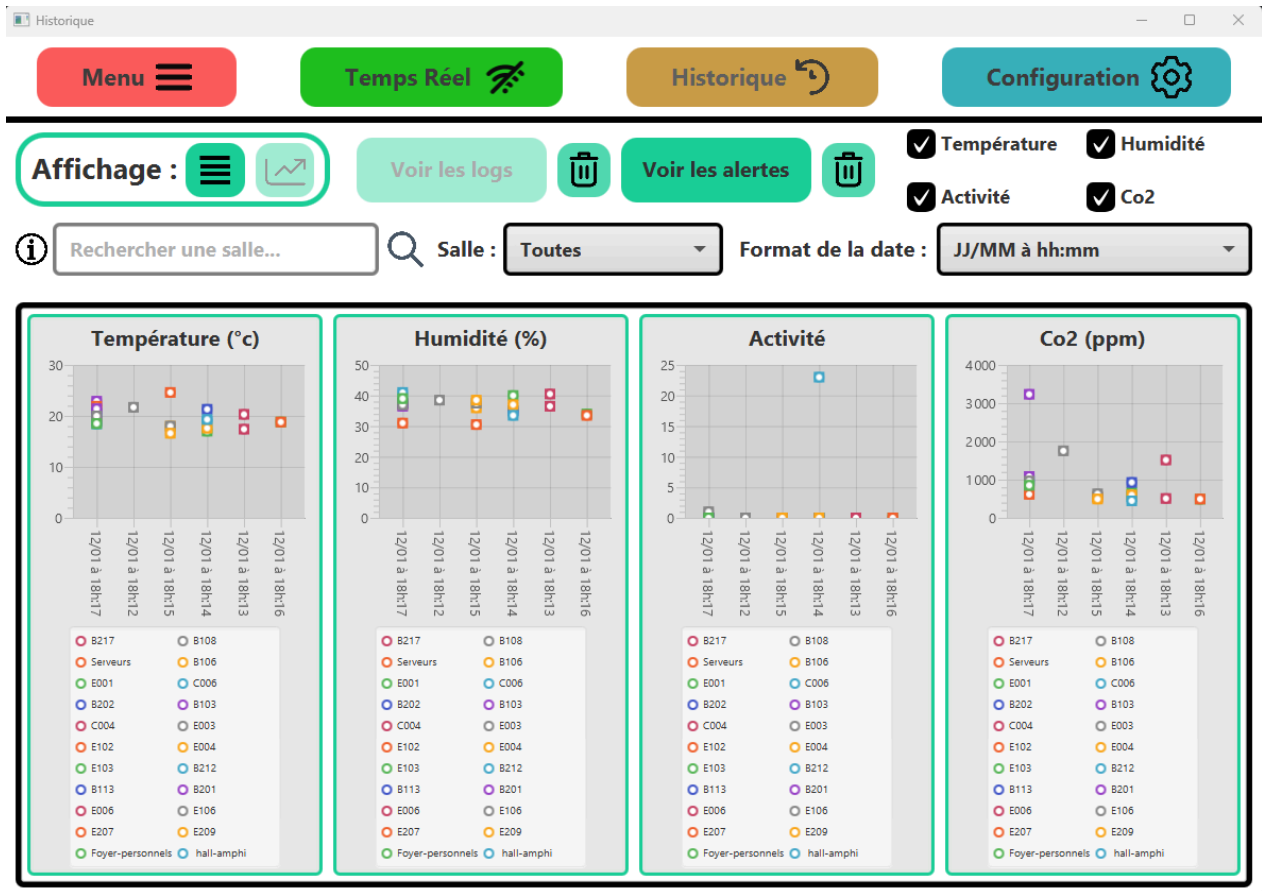
Extrait de code commenté :

```
363     writer.write("topic=" + topic + "\n");
364     writer.write(str:"[CONFIG]\n");
365     writer.write("fichier_alerte=" + alertFile + "\n");
366     writer.write("fichier_donnees=" + dataFile + "\n");
367     writer.write("fichier_logs=" + logsFile + "\n");
368     String choixDonnee = "";
369     if (cbTemperature.isSelected()) {
370         choixDonnee += "temperature,";
371     }
372     if (cbHumidity.isSelected()) {
373         choixDonnee += "humidity,";
374     }
375     if (cbActivity.isSelected()) {
376         choixDonnee += "activity,";
377     }
378     if (cbCo2.isSelected()) {
379         choixDonnee += "co2,";
380     }
381     // Suppression de la dernière virgule, si présente
382     if (choixDonnee.endsWith(suffix:",")) {
383         choixDonnee = choixDonnee.substring(beginIndex:0, choixDonnee.length() - 1);
384     }
385     writer.write("choix_donnees=" + choixDonnee + "\n");
386
387     tpTemps = cbTimeUnit.getValue();
388     if (tpTemps == "minute(s)") {
389         frequency = NumbersUtilities.getDoubleFromString(txtFrequency.getText()) * 60;
390     }
391     if (tpTemps == "heure(s)") {
392         frequency = NumbersUtilities.getDoubleFromString(txtFrequency.getText()) * 3600;
393     }
394     if (tpTemps == "jour(s)") {
395         frequency = NumbersUtilities.getDoubleFromString(txtFrequency.getText()) * 86400;
396     }
397     writer.write("typeTemps=" + tpTemps + "\n");
398     writer.write("frequence_affichage=" + frequency + "\n");
```

Ici, dans la méthode FXML "doSave" relié au bouton "Sauvegarder" permettant de sauvegarder la configuration, on écrit les nouvelles données saisies par l'utilisateur dans le fichier de configuration donc ici les noms des fichiers, le topic, les données choisies, la fréquence en convertissant en fonction de l'unité de temps choisit par l'utilisateur etc.

Visualiser l'historique de logs

L'application JavaFX permet de visualiser graphiquement les données des fichiers de logs.



Partie du UseCase :

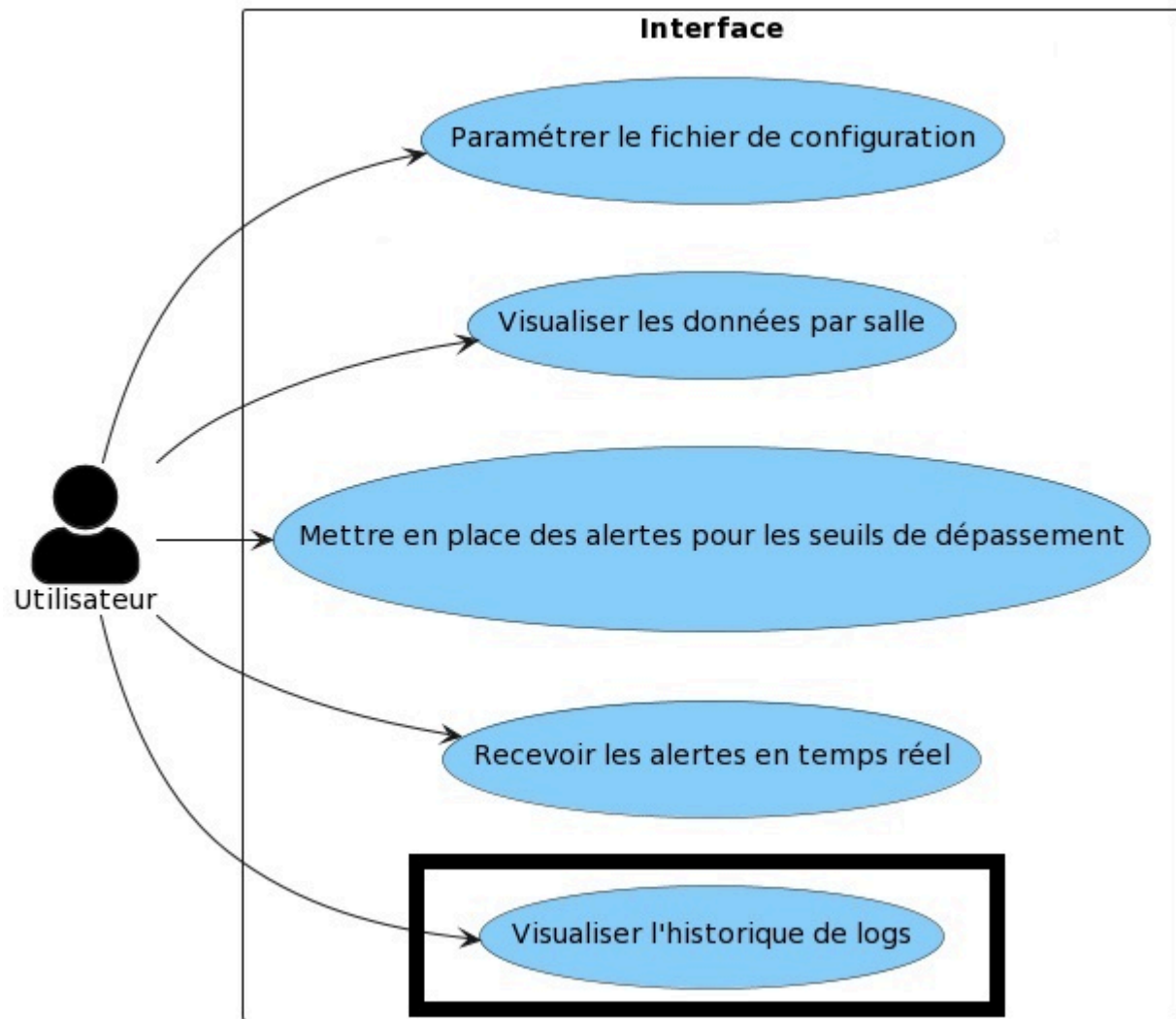
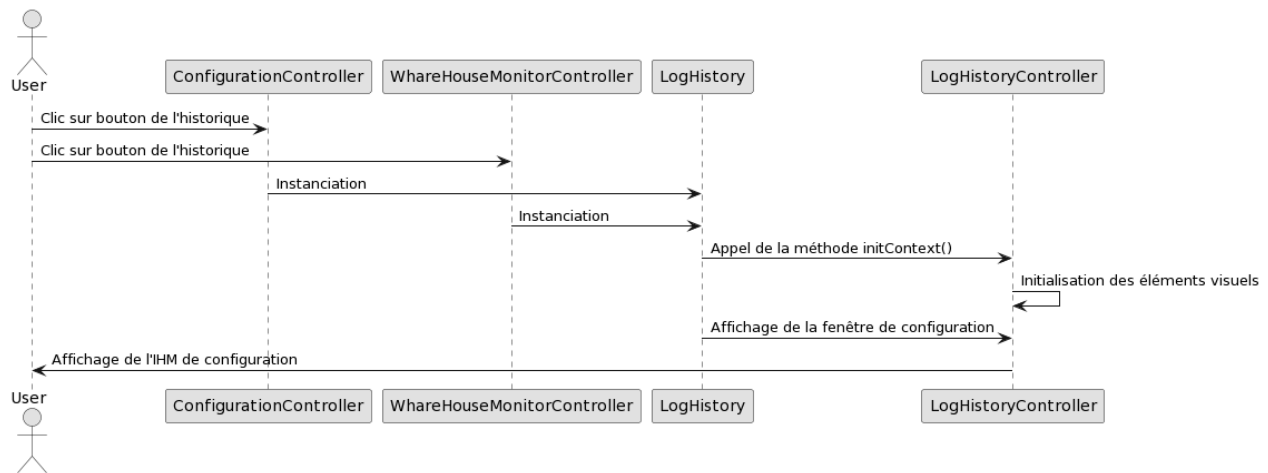


Diagramme de séquence :



Ce diagramme de séquence illustre l'interaction lorsqu'un utilisateur déclenche l'action de visualiser l'historique des logs. L'utilisateur peut initier cette action depuis le contrôleur de configuration ou celui de surveillance de l'entrepôt. Suite à cela, l'instanciation de la classe LogHistory est appelée depuis les deux contrôleurs concernés. Ensuite, la méthode `initContext()` du LogHistoryController est invoquée pour initialiser les éléments visuels de l'interface. Enfin, le contrôleur transmet la scène de l'historique des logs à l'utilisateur.

Classes utilisées :

ConfigurationController.java : Contrôleur pour la scène de configuration.

WhareHouseMonitorController.java : Contrôleur pour surveiller l'entrepôt.

LogHistory.java : Classe responsable de la fenêtre de l'historique.

LogHistoryController.java : Contrôleur pour la fenêtre de l'historique, gère les interactions et la logique.

Extrait de code commenté :

```

353 private void updateDatasHistory() {
354     listSearchedDatas.clear();
355     obsList.clear();
356     ListViewUtilities.updateSelectedElements(cbTemperature.isSelected(), cbHumidity.isSelected(),
357         cbActivity.isSelected(), cbCo2.isSelected());
358     ListViewUtilities.setCellForData(lvHistory);
359     if (currentSearch == null || currentSearch.trim().isEmpty()) {
360         listSearchedDatas.addAll(listAllRoomsDatas);
361         for (Data data : listAllRoomsDatas) {
362             obsList.add(data.toString(DateUtilities.transformDateFormat(comboBoxDateFormat.getValue())));
363         }
364     } else {
365         String[] roomsToSearch = currentSearch.split(regex:",");
366         for (Data data : listAllRoomsDatas) {
367             for (String room : roomsToSearch) {
368                 if (data.getId().toLowerCase().contains(room.trim().toLowerCase())) {
369                     listSearchedDatas.add(data);
370                     obsList.add(data.toString(DateUtilities.transformDateFormat(comboBoxDateFormat.getValue())));
371                     break;
372                 }
373             }
374         }
375     }
376 }
377

```

La fonction suivante "**updateDatasHistory**" met à vide l'observable list relié à la listview ainsi que l'ArrayList contenant les données recherchées. Elle met ensuite la scène à jour en fonction des données choisies par l'utilisateur (graphique de température seulement par exemple), puis si la variable "currentSearch" qui correspond à la recherche actuel de l'utilisateur est null ou vide, la liste des données recherchées va recevoir toute la liste contenant toutes les données. Si la recherche n'est ni null ni vide, on va remplir la liste seulement avec les données dont le nom correspond avec la recherche actuelle.

Visualiser les données en temps réel

L'application JavaFX permet de visualiser graphiquement les données en temps réel avec différents choix de vues disponibles.



Partie du UseCase :

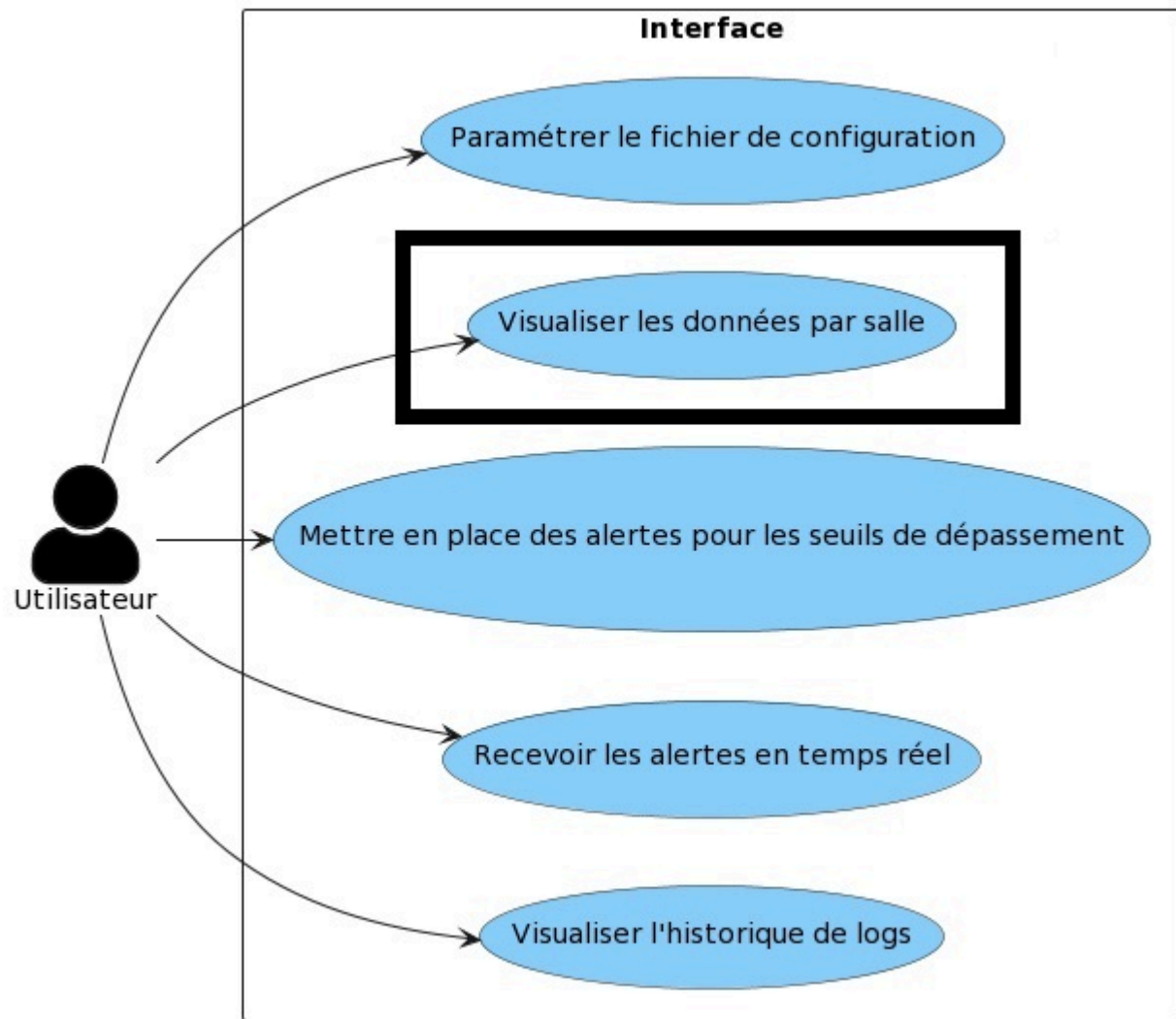
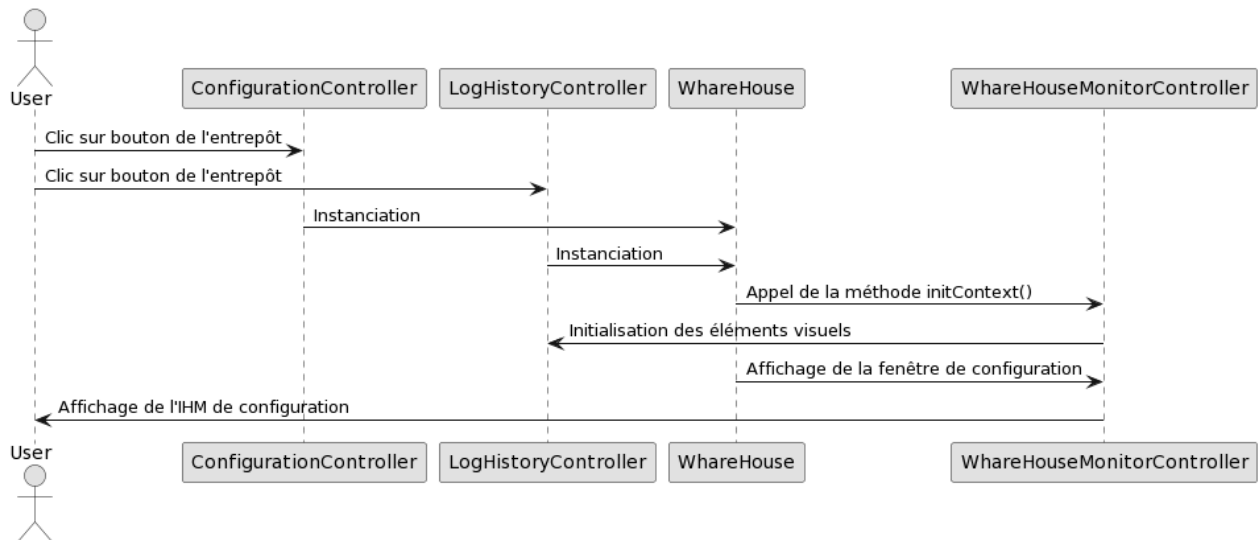


Diagramme de séquence :



Ce diagramme de séquence illustre le processus déclenché par l'utilisateur lorsqu'il clique sur le bouton de configuration à partir de deux interfaces distinctes de l'application. Lorsque l'utilisateur effectue cette action depuis l'interface gérée par le LogHistoryController, une instance de la classe Configuration est créée, suivie de l'initialisation des éléments visuels et du thread de test de connexion par le ConfigurationController. De manière similaire, le même processus est enclenché à partir de l'interface gérée par le WhareHouseMonitorController, générant une autre instance de la classe Configuration.

Classes utilisées :

ConfigurationController.java : Contrôleur pour la scène de configuration.

LogHistoryController.java : Contrôleur pour la scène de l'historique.

WhareHouseMonitor.java : Classe responsable de la fenêtre de l'entrepôt.

WhareHouseMonitorController.java : Contrôleur pour la fenêtre de l'entrepôt, gère les interactions et la logique.

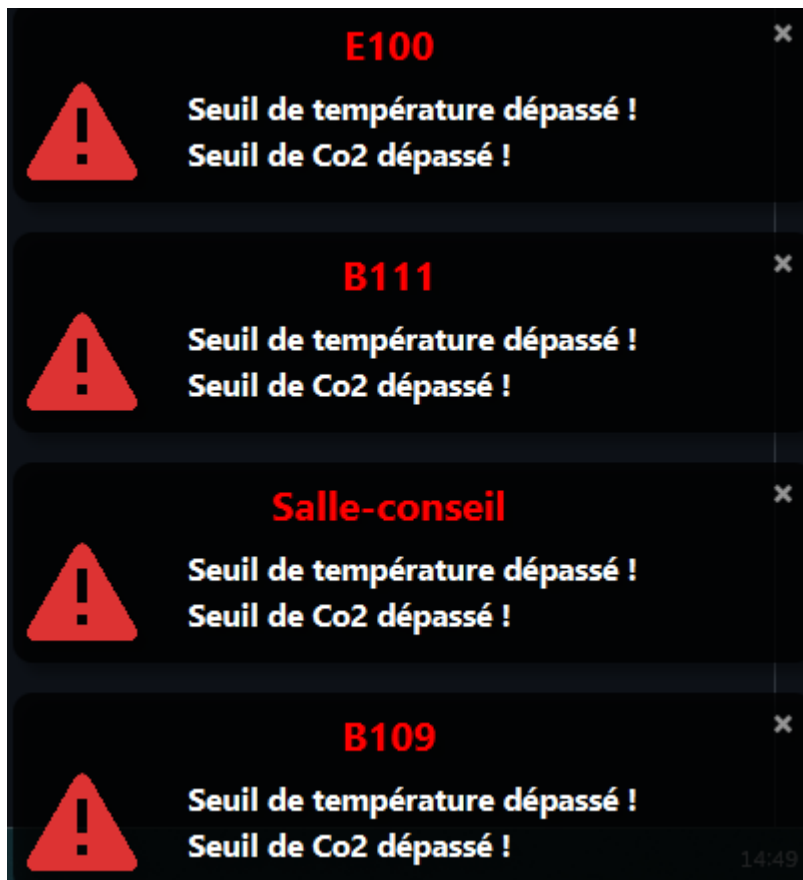
Extrait de code commenté :

```
176     private void initGetNewDatasThread() {
177         updatesDataThread = new Thread(() -> {
178             long lastModified = 0;
179             while (true) {
180                 try {
181                     File jsonFile = new File(pathname:"code\\IOT\\Python\\donnees.json");
182                     long currentLastModified = jsonFile.lastModified();
183                     if (currentLastModified > lastModified) {
184                         Platform.runLater(() -> {
185                             if (jsonFile.exists()) {
186                                 JsonUtilities.updateHistoryFromFile(primaryStage, isCurrentDatas:true, _isAlertFile:false, obsList,
187                                     listAllRoomsDatas, _listAllRoomsAlerts:null, comboBoxRooms);
188                                 updateSceneByView();
189                                 checkAlertForLastData();
190                             }
191                         });
192                         lastModified = currentLastModified;
193                     }
194                 } catch (Exception e) {
195                     e.printStackTrace();
196                 }
197             }
198         });
199         updatesDataThread.start();
200     }
```

La méthode suivante "**initGetNewDatasThread**" initialise et lance un thread permettant de récupérer les nouvelles données écrites dans le fichier de données JSON. Le thread vérifie en permanence la date de dernière modification du fichier, si une nouvelle modification a lieu, un appel à "**updateHistoryFromFile**" aura lieu pour mettre à jour l'ArrayList de données en parcourant de nouveau le fichier Json.

Recevoir des alertes

Vérifie les données captées par les capteurs pour chaque salle surveillée. Si une donnée dépasse le seuil prédéfini, l'application affiche instantanément une alerte correspondante. Elles s'affichent en bas à droite que l'application soit au premier plan ou non.



Partie du UseCase :

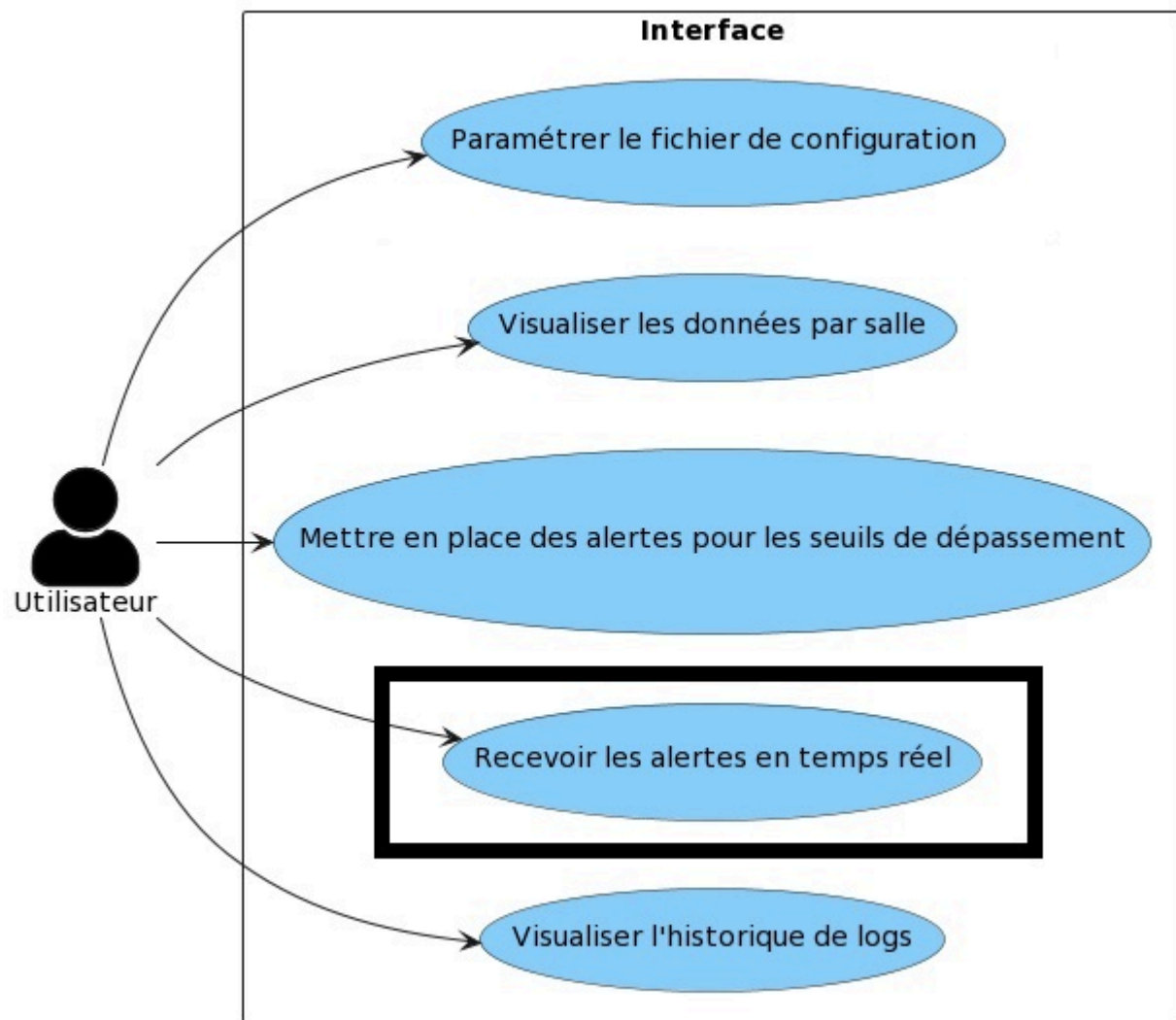
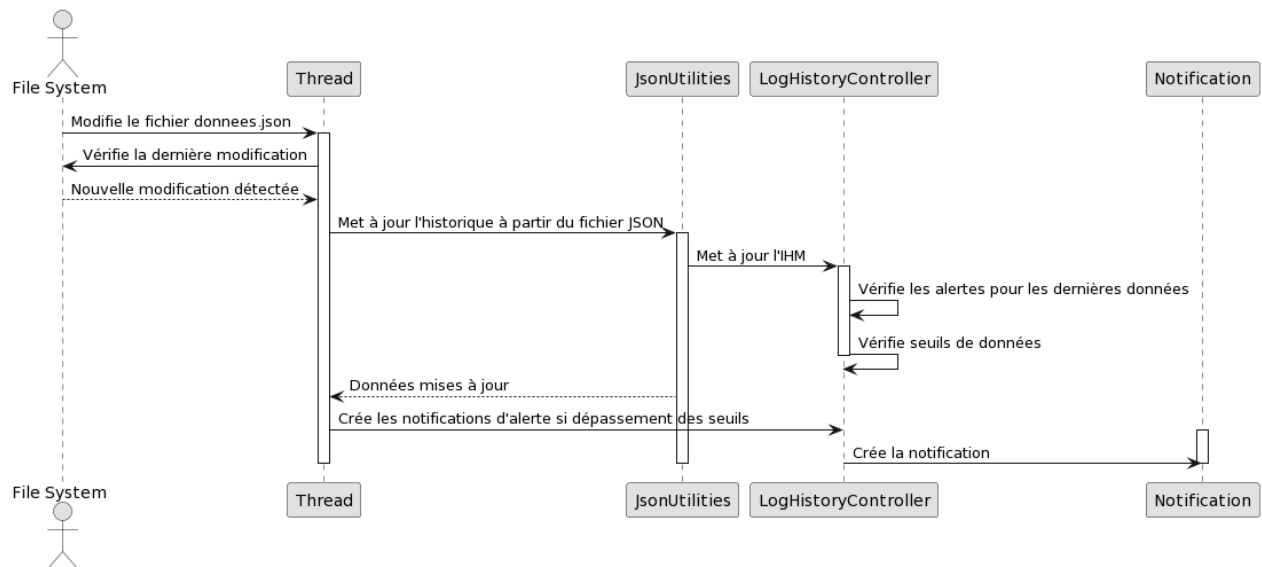


Diagramme de séquence :



Ce schéma représente un processus automatisé où le système de fichiers modifie le fichier "donnees.json". Lorsqu'une modification est détectée, un thread est activé pour mettre à jour l'historique via JsonUtilities. En parallèle, le contrôleur de l'historique (LogHistoryController) vérifie les alertes pour les dernières données et les seuils. Une fois les données mises à jour, le contrôleur crée des notifications d'alerte, si les seuils sont dépassés, grâce à la classe de notification. Ce processus garantit une surveillance continue des données, avec une réactivité en temps réel pour informer les utilisateurs en cas de dépassement des seuils.

Classes utilisées :

WhareHouseMonitorController.java : Contrôleur pour la fenêtre de l'entrepôt, gère les interactions et la logique.

Extrait de code commenté :

```

209     private void checkAlertForLastData() {
210         if (listAllRoomsDatas.size() > 0) {
211             Data data = listAllRoomsDatas.get(listAllRoomsDatas.size() - 1);
212             String alerts = "";
213             if (data.getTemperature() > maxTemperature) {
214                 alerts += "    Seuil de température dépassé !\n";
215             }
216             if (data.getHumidity() > maxHumidity) {
217                 alerts += "    Seuil d'humidité dépassé !\n";
218             }
219             if (data.getActivity() > maxActivity) {
220                 alerts += "    Seuil d'activité dépassé !\n";
221             }
222             if (data.getCo2() > maxCo2) {
223                 alerts += "    Seuil de Co2 dépassé !\n";
224             }
225             if (!alerts.equals(anObject:"")) {
226                 createAlertNotification(data.getId(), alerts);
227             }
228         }
229     }
230
231     private void createAlertNotification(String _roomId, String _alert) {
232         Notifications.create()
233             .title(_roomId)
234             .text(_alert)
235             .hideAfter(Duration.seconds(s:5))
236             .position(Pos.BOTTOM_RIGHT)
237             // .owner(primaryStage)
238             .graphic(new ImageView(new Image(url:"/application/images/alert_data_icon.png",
239                 requestedWidth:45, requestedHeight:45, preserveRatio:false, smooth:false)))
240             .show();
241     }

```

La méthode "**checkAlertForLastData**" est appelée à chaque nouvel entrée dans l'Arraylist de données. Cette méthode va vérifier si les seuils d'alerte ont été atteints pour la nouvelle donnée. Si c'est le cas, un appel à "**createAlertNotification**" va avoir lieu pour créer une notification et alerter l'utilisateur des seuils dépassés.