Documentation Technique

Sommaire

Étudiants
Nolhan Biblocque
Léo Guinvarc'h
Victor Jockin
Mathys Laguilliez
Mucahit Lekesiz

Enseignant
André Péninou

Formation
BUT Informatique
2ème Année
Promotion 2024-2025

Établissement
IUT de Blagnac
Université Toulouse II – Jean Jaurès (31)

1. Présentation de l’application

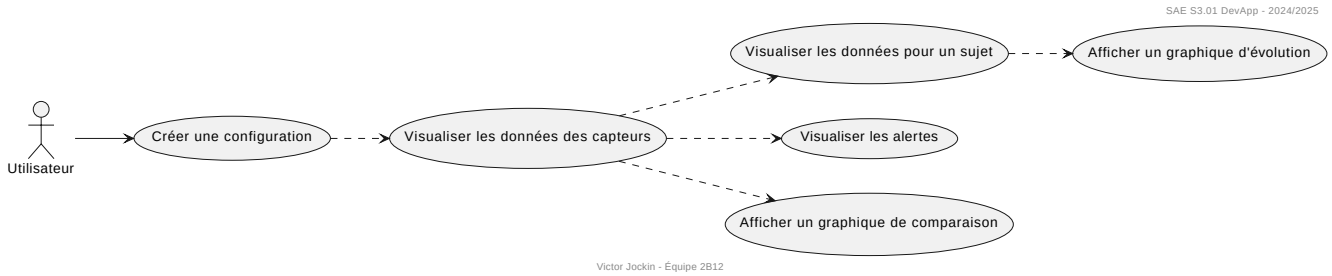
1.1 Objectif et fonctionnement

L’application développée a pour objectif de fournir un outil centralisé de visualisation et d’analyse des données issues de capteurs connectés. Elle repose sur une architecture modulaire combinant Java/JavaFX et Python.

- JavaFX assure l’aspect "interface utilisateur" (IHM).
- Java assure le traitement des actions utilisateurs et plus généralement le fonctionnement interne de l’application.
- Python assure le processus de collecte des données des capteurs.

1.2 Fonctionnalités principales

1.2.1 Diagramme des Cas d’Utilisation



1.2.2 Description des fonctionnalités

- **Création d’une configuration :** Paramétrage des capteurs à consulter, des sujets à observer, des types de données à récupérer et des seuils d’alerte.
- **Visualisation des données des capteurs :** Affichage des relevés en temps réel dans un tableau de bord.
- **Visualisation des alertes :** Signalement dans le tableau de bord des dépassements de seuils d’alerte définis.
- **Analyse des données :** Représentation des données sous forme de graphiques d’évolution ou de comparaison.

2. Architecture de l'application

2.1 Architecture générale

2.1.1 Arborescence

Arborescence du projet

```
application_iot/
├──── resources/
│     ├──── data/
│     │     ├──── data.csv
│     │     ├──── alert.csv
│     │     └──── ...
│     ├──── mqtt.py
│     └──── configuration.ini
├──── src/
├──── target/
│     ├──── site/
│     │     └──── apidocs/
│     │           ├──── index.html
│     │           └──── ...
│     └──── ...
├──── dependency-reduced-pom.xml
└──── pom.xml
```

Description des répertoires et fichiers

`resources` : Contient les fichiers et programmes externes nécessaires au fonctionnement de l'application Java / JavaFX.

- `data` : Contient les fichiers de données générés par le programme Python.
  - `data.csv` : Le fichier de données global.
  - `alert.csv` : Le fichier d'alertes.
  - Les autres fichiers de données chacun associé à un sujet observé.
- `mqtt.py` : Le programme Python chargé de la collecte des données (client MQTT).
- `configuration.ini` : Le fichier de configuration du programme Python.

`src` : Contient le code source de l'application Java / JavaFX.

`target` : Contient les fichiers générés par le processus de compilation de l'application.

- `site/apidocs/index.html` : Page principale de la documentation Javadoc de l'API.

`pom.xml` : Fichier gérant les dépendances et la configuration du projet Maven pour l'application Java / JavaFX.

`dependency-reduced-pom.xml` : Réduction du fichier `pom.xml`.

2.1.2 Sous-systèmes de l'application

Application Java / JavaFX

- **Rôle :** Gestion de l'interface graphique et mise en relation des différents sous-systèmes et fichiers.
- **Tâches réalisées :**
  - Gestion d'une interface de paramétrage d'une configuration.
  - Lancement et interruption du programme Python chargé de la collecte des données.
  - Lecture des fichiers de données écrits par le programme Python.
  - Gestion d'un tableau de bord permettant la visualisation des données des capteurs.

Programme Python

- **Rôle :** Collecte des données envoyées par les capteurs SOLAREDGE et AM107.
- **Tâches réalisées :**
  - Initialisation en fonction des paramètres définis dans le fichier de configuration.
  - Réception des données envoyées par les capteurs.
  - Écriture des données reçues dans des fichiers CSV.

2.1.3 Fichiers utilisés

Fichier de configuration

Le fichier de configuration `configuration.ini` situé sous le répertoire `resources` contient les paramètres de la configuration créée par l'utilisateur au travers de l'interface de l'application Java. Ce fichier est lu par le programme Python à son lancement qui adapte ainsi son comportement en fonction des paramètres spécifiés.

STRUCTURE DU FICHIER

```
[MQTT] ; [1]
broker=mqtt.iut-blagnac.fr
port=1883
topic={{ PRÉFIXE DES TOPIC MQTT }}

[SUBJECTS] ; [2]
subject1={{ SUJET 1 }}
subject2={{ SUJET 2 }}
...

[DATA_TYPE] ; [3]
dataType1={{ TYPE DE DONNÉES 1 }}
dataType2={{ TYPE DE DONNÉES 2 }}
```

```
dataType3={{ TYPE DE DONNÉES 3 }}
...

[THRESHOLD] ; [4]
{{ TYPE DE DONNÉES 1 }}={{ SEUIL }}
{{ TYPE DE DONNÉES 2 }}={{ SEUIL }}
{{ TYPE DE DONNÉES 3 }}={{ SEUIL }}
...

[PARAMS] ; [5]
frequency={{ FRÉQUENCE }}
```

[1] Paramètres de connexion MQTT

- `broker` : Adresse du broker MQTT (valeur fixe).
- `port` : Port utilisé pour la connexion au broker (port standard MQTT, valeur fixe).
- `topic` : Préfixe des topics auxquels le programme Python doit s'abonner.
  - Pour accès aux capteurs AM107, le préfixe correspondant est `AM107/by-room/`.
  - Pour accès aux capteurs SOLAREDGE, le préfixe correspondant est `solaredge/blagnac/`.

[2] Liste des sujets à observer

- `subjectI` : I-ème sujet à observer.
  - Pour les capteurs AM107, le nombre de sujets à observer peut aller jusqu'au nombre total de salles disponibles, soit 53.
  - Pour les capteurs SOLAREDGE, le nombre de sujets à observer se limite à 1 : `overview`.

[3] Liste des types de données à récupérer

- `dataTypeI` : I-ème type de données à récupérer pour le type de capteurs consulté.

[4] Liste des seuils d'alerte par type de données (capteurs AM107 uniquement)

- Cette section indique, pour chaque type de données listé dans la section `DATA_TYPE`, le seuil dont le dépassement déclenchera une alerte.

[5] Paramètres avancés

- `frequency` : Fréquence de lecture des données.
  - **À noter :** La valeur pour ce paramètre n'a actuellement aucun impact sur le comportement du programme Python car non traitée. La fréquence définie lors du paramétrage de la configuration est cependant prise en compte par le processus de lecture des données de l'application Java.

Fichiers de données

Les fichiers de données situés sous le répertoire `resources/data` sont des fichiers CSV permettant de stocker les données des capteurs. Ces fichiers sont créés et remplis par le programme Python et lus par l'application Java.

La première ligne de chaque fichier CSV contient les en-têtes décrivant la nature des données des lignes suivantes (lignes de données).

À noter : Dans les fichiers CSV manipulés, le séparateur de données utilisé est le point-virgule (`;`).

Fichier de données global

Le fichier `data.csv` correspond au fichier de données global. Il contient les dernières données reçues pour chaque sujet.

- Dans le cas des capteurs AM107, une ligne de données du fichier correspond aux dernières données reçues pour une salle.
- Dans le cas des capteurs SOLAREDGE, la seule ligne de données présente dans le fichier correspond aux dernières données reçues pour le panneau solaire.

Ce fichier est utilisé par l'application Java afin d'afficher dans le tableau de bord les données en temps réel pour chaque sujet observé ainsi que pour générer des diagrammes de comparaison des sujets sur un type de données.

STRUCTURE DU FICHIER

```
{{ TYPE DE SUJET }};{{ TYPE DE DONNEE 1 }};{{ TYPE_DE DONNEE 2 }}
{{ SUJET 1 }};{{ DERNIÈRE VALEUR MESURÉE }};{{ DERNIÈRE VALEUR MESURÉE }}
{{ SUJET 2 }};{{ DERNIÈRE VALEUR MESURÉE }};{{ DERNIÈRE VALEUR MESURÉE }}
{{ SUJET 3 }};{{ DERNIÈRE VALEUR MESURÉE }};{{ DERNIÈRE VALEUR MESURÉE }}
...
```

Fichier d'alertes (capteurs AM107 uniquement)

Le fichier `alert.csv` correspond au fichier d'alertes. Il contient l'ensemble des alertes déclenchées par des dépassements de seuils. Une ligne de données du fichier correspond donc à une alerte pour un type de données et pour une salle.

Ce fichier est utilisé par l'application Java afin d'afficher les alertes en temps réel dans le tableau de bord.

STRUCTURE DU FICHIER

```
room;dataType;threshold;measuredValue
{{ SALLE 1 }};{{ TYPE DE DONNÉES }};{{ SEUIL }};{{ VALEUR MESURÉE }}
{{ SALLE 2 }};{{ TYPE DE DONNÉES }};{{ SEUIL }};{{ VALEUR MESURÉE }}
...
```

Fichiers de données par sujet

Les fichiers dont le nom est de la forme `SUJET.csv` correspondent chacun à un fichier de données pour un sujet en particulier. Un fichier de ce type contient l'historique des données reçues pour un sujet.

- Dans le cas des capteurs AM107, autant de fichiers sont créés que de sujets sont observés. Les noms de ces fichiers correspondent aux noms des salles observées (exemple : `B101.csv`).
- Dans le cas des capteurs SOLAREDGE, un seul fichier nommé `overview` est créé.

Ces fichiers sont exploités par l'application Java afin de construire des graphiques décrivant l'évolution des valeurs pour un type de données.

STRUCTURE DU FICHIER

```
{{ TYPE DE SUJET }};{{ TYPE DE DONNEE 1 }};{{ TYPE_DE DONNEE 2 }}
{{ SUJET }};{{ VALEUR MESURÉE À L'INSTANT T0 }};{{ VALEUR MESURÉE À L'INSTANT T0 }}
{{ SUJET }};{{ VALEUR MESURÉE À L'INSTANT T1 }};{{ VALEUR MESURÉE À L'INSTANT T1 }}
{{ SUJET }};{{ VALEUR MESURÉE À L'INSTANT T2 }};{{ VALEUR MESURÉE À L'INSTANT T2 }}
...
```

2.1.4 Interactions entre sous-systèmes et fichiers

1. **Écriture du fichier de configuration par l'application Java**
   - Après le paramétrage d'une configuration par l'utilisateur dans l'interface graphique, l'application Java crée un fichier `configuration.ini` sour le répertoire `resources` décrivant la configuration créée.
   - **À noter :** À cette étape, si un fichier de configuration existe déjà, celui-ci est remplacé par le fichier de configuration nouvellement créé. Aucun mécanisme d'historisation ou de sauvegarde des fichiers de configurations n'a été mis en place.
2. **Lancement du programme Python par l'application Java**
   - Une fois le fichier de configuration créé, l'application Java démarre le processus de collecte des données en lançant en exécution le programme Python.
3. **Collecte des données par le programme Python**
   - Au lancement, le programme Python lis le fichier de configuration définissant son comportement.
   - Une fois lancé, il attend jusqu'à interruption les données envoyées par les sujets (capteurs).
   - À chaque réception de données, celles-ci sont enregistrées dans les fichiers de données correspondants.
4. **Lecture des fichiers de données par l'application Java**
   - En parallèle de l'exécution du programme Python, l'application Java lis à intervalle régulier (fréquence définie dans le fichier de configuration) les fichiers de données.
   - Les données lues sont ensuite stockées dans des structures de données puis transmises au tableau de bord de l'application pour affichage.

5. **Interruption du programme Python par l'application Java**
    - Lorsque le tableau de bord de l'application est fermé par l'utilisateur, le programme Python est automatiquement arrêté.
    - **À noter :** Après arrêt du processus de collecte des données, le fichier de configuration ainsi que les fichiers de données écrits sont conservés. Ils seront écrasés lors de la prochaine exécution de l'application.

2.2 Ressources externes

2.2.1 API utilisées

JavaFX

- **Rôles :**
    - Conception de l'IHM avec le module `javafx-fxml` (création d'interfaces utilisateur via des fichiers FXML).
    - Prise en charge et gestion de l'interface graphique dans l'application.
- **Version utilisée :** 17
- **Site officiel de JavaFX :** JavaFX - Home
- **Documentation officielle :** Oracle - JavaFX Documentation

OpenCSV

- **Rôle :** Lecture des fichiers de données au format `CSV` générés par le programme python collecteur de données.
- **Version utilisée :** 5.5.2
- **Site officiel de JavaFX :** OpenCSV - About / Opencsv Users Guide
- **Documentation officielle :** OpenCSV - About / Developer Documentation

2.2.2 Plugins utilisés

JavaFX Maven Plugin

- **Rôle :** Packaging et exécution de l'application JavaFX.
- **Version utilisée :** 0.0.8
- **Site officiel de Maven Repository :** Maven Repository - JavaFX Maven Plugin Maven Mojo
- **Lien vers le dépôt GitHub du plugin :** GitHub - Maven plugin for JavaFX

Apache Maven Shade Plugin

- **Rôle :** Création d'un exécutable au format `JAR` contenant toutes les dépendances nécessaires au fonctionnement de l'application.
- **Version utilisée :** 3.4.1

- **Site officiel d'Apache Maven :** Apache Maven Project - Apache Maven Shade Plugin

Apache Maven Javadoc Plugin

- **Rôle :** Génération de la documentation du projet Java avec `Javadoc`.
- **Version utilisée :** 3.4.1
- **Site officiel d'Apache Maven :** Apache Maven Project - Apache Maven Javadoc Plugin

2.3 Structuration de l'application Java

2.3.1 Patterns mis en oeuvre

Architecture MVC

L'application Java repose sur une architecture MVC (Modèle-Vue-Contrôleur / Model-View-Controller) permettant la séparation des couches de **présentation**, de **logique métier** et de **traitement des actions utilisateur**.

Présentation

- **Composante MVC associée :** Vue (*View*).
- **Rôle :**
  - Afficher les données envoyées par le Contrôleur.
  - Permettre à l'utilisateur d'intéragir avec l'interface graphique.

Logique métier

- **Composante MVC associée :** Modèle (*Model*).
- **Rôle :**
  - Représenter les données manipulées par l'application.
  - Appliquer des règles de gestion sur les données.
  - Fournir une interface permettant l'accès aux données et leur mise à jour.
  - Notifier le Contrôleur après une mise à jour des données.

Traitement des actions utilisateur

- **Composante MVC associée :** Contrôleur (*Controller*).
- **Rôle :**
  - Effectuer des opérations sur le Modèle en fonction des actions utilisateur.
  - Mettre à jour la Vue afin de refléter les changements dans le Modèle.

Composants en Singleton

Configuration

La classe modèle représentant la configuration paramétrée par l'utilisateur (`Configuration.java`) est implémentée en *Singleton* en ce que l'application permet actuellement de définir une seule configuration à la fois. En d'autres termes, lorsqu'une nouvelle configuration est définie, celle-ci écrase automatiquement la configuration précédente.

Une implémentation selon le patron *Singleton* permet ainsi à cette classe de fournir une méthode donnant accès à l'unique instance de la configuration.

Cette implémentation serait susceptible d'évoluer si un mécanisme d'historisation ou de sauvegarde des différentes configurations déifnies par l'utilisateur était mis en place.

2.3.2 Packages

Arborescence des packages

Les packages de l'application Java sont situés sous le répertoire `src/main/java`.

```
application
 ├──── config
 ├──── control
 ├──── data
 ├──── enums
 ├──── model
 ├──── styles
 ├──── thread
 ├──── tools
 └──── view
```

Description des packages et de leur contenu

`application` : Package "racine" de l'application.

- `ApplicationMainFrame` : Contrôleur de dialogue du menu principal (fenêtre principale de l'application).
- `Main` : Classe principale de l'application.

`application.config` : Package des classes manipulant le fichier de configuration.

- `ConfigurationFileWriter` : Classe permettant d'écrire un fichier de configuration.

`application.control` : Package des contrôleurs de dialogue (Cf. Architecture MVC).

- `ConfirmationFileForm` : Contrôleur de dialogue du formulaire de paramétrage de la configuration.

- `DataVisualisationPane` : Contrôleur de dialogue du tableau de bord (fenêtre de visualisation des données).
- **À noter :** La classe `ApplicationMainFrame` située dans le package `application` pourrait être déplacée dans ce package en ce qu'il s'agit d'un contrôleur de dialogue.

`application.data` : Package des classes relatives aux données.

- `DataCollector` : Classe de gestion du processus de collecte des données.
- `DataLoader` : Classe d'accès aux fichiers de données.
- `DataTypeUtilities` : Classe utilitaire fournissant des méthodes relatives aux types données (ex : formatage de noms).

`application.enums` : Package des énumérations.

- `Room` : Classe d'énumération des salles existantes.
- `RoomDataType` : Classe d'énumération des types de données des salles.
- `Sensor` : Classe d'énumération des types de capteurs (`AM107` / `SOLAREDGE`).
- `SolarPanelDataType` : Classe d'énumération des types de données des panneaux solaires.

`application.model` : Package des classes modèles (Cf. Architecture MVC).

- `Configuration` : Classe modèle représentant une configuration.
- `DataRow` : Classe modèle représentant une ligne de données (Cf. Fichiers de données).

`application.styles` : Package des classes de stylisation de l'interface graphique.

- `FontLoader` : Classe d'accès aux typographiques (fonts) utilisées dans l'interface graphique.

`application.thread` : Package des threads.

- `CsvReaderTask` : Thread chargé de lire le fichier de données global (`data.csv`) et le fichier d'alertes (`alert.csv`).
- `UpdateAlertDisplayTask` : Thread chargé de la mise à jour de l'affichage des alertes dans le tableau de bord.
- `UpdateDataDisplayTask` : Thread chargé de la mise à jour de l'affichage des données dans le tableau de bord.

`application.tools` : Package des classes utilitaires.

- `DataFileReading` : Classe utilitaire fournissant des méthodes de lecture de fichiers de données.
- `GraphGenerator` : Classe utilitaire fournissant des méthodes de génération de graphiques.
- `TextUtilities` : Classe utilitaire fournissant des méthodes relatives à des éléments textuels (NON UTILISÉE).

`application.view` : Package des contrôleurs de vue (Cf. Architecture MVC).

- `ApplicationMainFrameViewController` : Contrôleur de vue du menu principal.

- `ConfigurationFileFormViewController` : Contrôleur de vue du formulaire de paramétrage de la configuration.
- `DataVisualisationPaneViewController` : Contrôleur de vue du tableau de bord.

2.4 Spécifications techniques

2.4.1 Conventions de nommage

L'anglais est utilisé pour tous les noms de classes, de variables, de packages et de vues.

Nommage des classes

Les noms de classes Java sont formatés en Upper Camel Case.

Contrôleurs de dialogue

- **Règle :** Nom de la vue FXML en Upper Camel Case.
- **Exemple :** Contrôleur de dialogue associé à la vue `configurationFileForm.fxml` → `ConfigurationFileForm`.

Contrôleurs de vue

- **Règle :** Nom de la vue FXML en Upper Camel Case + `ViewController`.
- **Exemple :** Contrôleur de la vue `configurationFileForm.fxml` → `ConfigurationFileFormViewController`.

Classes utilitaires

- **Règle :** Objet manipulé + `Utilities`.
- **Exemple :** Classe utilitaire fournissant des méthodes relatives aux types données → `DataTypeUtilities`.

Classes modèle

- **Règle :** Nom de l'objet représenté.
- **Exemple :** Classe modèle représentant une configuration → `Configuration`.

Classes d'énumération

- **Règle :** Nom du type d'objet énuméré.
- **Exemple :** d'énumération des salles → `Room`.

Les classes d'énumération, pouvant avoir des noms identiques à ceux de classes modèles, ont été placées dans un package `enums` dédié afin d'éviter toute confusion.

Threads

- **Règle :** Nom de la tâche réaliée par le thread + `Task`.
- **Exemple :** Thread chargé de la mise à jours de l'affichage des données dans le tableau de bord → `UpdateDataDisplayTask`.

Classes avec méthodes statiques

- **Règle :** Objet concerné + Verbe d'action.
- **Exemple :** Classe d'accès aux données → `DataLoader`.

Nommage des variables

Les noms de variables Java sont formatés en Lower Camel Case.

Variables éphémères

S'applique aux variables de type indice ou aux compteurs de boucles.

- **Règle :** Nom "court".
- **Exemples :**
  - Compteur de boucle `for` → `i`.
  - Entrée couremment traité dans une boucle de type `for-each` parcourant le contenu d'un dictionnaire → `m`.

Remarque : Les noms de ces variables peuvent être plus explicites si besoin.

Variables récurrentes

S'applique aux variables et aux collections utilisées à plusieurs endroits dans une classe.

- **Règle :** Nom explicite.
- **Exemples :**
  - Chaîne de caractères décrivant le préfixe d'un topic MQTT → `topicPrefix`.
  - Liste de types de données → `dataTypeList`.

Variables de composants graphiques

S'applique uniquement à une variable d'un contrôleur de vue correspondant à un élément graphique de la vue associée.

- **Règle :** Rôle du composant graphique + Éventuellement type du composant.
- **Exemples :**
  - Composant graphique de type Bouton (`Button`) → `button`.
  - Conteneur de graphiques de type `VBox` → `graphContainerVBox` ou `graphContainer`.

Paramètres

S'applique uniquement aux paramètres de fonctions et de méthodes.

- **Règle :** `p` + Nom explicite en Upper Camel Case.
- **Exemples :**
  - Liste de types de données passée en paramètre d'une fonction / méthode → `pDataTypeList` (Cf. Variables récurrentes).
  - Composant graphique de type Bouton (`Button`) passé en paramètre d'une fonction / méthode → `pButton` (Cf. Variables de composants graphiques).

Nommage des packages

- **Règle :** Nom court décrivant le type des classes contenues par la package en Lower Camel Case.
- **Exemple :** Package des classes utilitaires → `tools`.

Nommage des vues FXML

- **Règle :** Nom de la vue en Lower Camel Case.
- **Exemple :** Vue du formulaire de paramétrage du fichier de configuration → `configurationFileForm.fxml`.

2.4.2 Conventions de commentaire

Les commentaires des classes Java sont entièrement rédigés en français.

Commentaire d'en-tête de classe

Modèle

```
/**
 * [ TYPE DE CLASSE + RÔLE ]
 *
 * Date de dernière modification :
 * - [ DATE ] -
 *
 * @author [ DÉVELOPPEUR ]
 * ...
 * - [ NOM DE L'ÉQUIPE DE DÉVELOPPEMENT ] -
 */
```

Exemple

Classe : `ConfigurationFileForm`

```
/**
 * Contrôleur de dialogue du formulaire de paramétrage
 * d'un fichier de configuration.
 *
 * Date de dernière modification :
 * - Mardi 10 décembre 2024 -
 *
 * @author Victor Jockin
 * - Équipe G2B12 -
 */
```

Commentaire d'en-tête de méthode

Modèle

```
/**
 * [ RÔLE DE LA MÉTHODE ]
 * @param pParam1    [ DESCRIPTION DU PARAMÈTRE ]
 * @param pParam2    [ DESCRIPTION DU PARAMÈTRE ]
 * ...
 * @return  [ DESCRIPTION DE LA VALEUR RETOURNÉE ]
 * @throws EXCEPTION [ CONDITION DE LEVÉE DE L'EXCEPTION ]
 */
```

Exemple

Méthode : `ConfigurationFileForm.isThresholdValid(RoomDataType, double)`

```
/**
 * Indique si un seuil d'alerte pour un type de données de salle est valide.
 * @param pRoomDataType un type de données de salle
 * @param pThreshold    un seuil d'alerte
 * @return  true si le seuil d'alerte est valide, false sinon
 */
```

2.4.3 Structures récurrentes de classes

Dans cette section, on suppose avoir l'arborescence suivante :

```
src/
 ├──── main/
 │     ├──── java/
 │     │     └──── application/
 │     │          ├──── control/
 │     │          │     ├──── Example.java  [1]
 │     │          │     └──── ...
 │     │          ├──── view/
 │     │          │     ├──── ExampleViewController.java  [2]
 │     │          │     └──── ...
 │     │          └──── ...
 │     └──── resources/
 │          └──── application/
 │               ├──── view/
 │               │     ├──── example.fxml  [3]
 │               │     └──── ...
 │               └──── ...
 └──── ...
```

1. Contrôleur de dialogue d'exemple.
2. Contrôleur de vue d'exemple.
3. Vue FXML d'exemple.

Contrôleurs de dialogue

```
package application.control ;

import application.view.ExampleViewController ;

...

public class Example
{
    // déclaration des constantes
    // -------------------------

    private static final double MIN_WINDOW_WIDTH    = ... ;     // largeur minimale de
la fenêtre
    private static final double MIN_WINDOW_HEIGHT   = ... ;     // hauteur minimale de
la fenêtre
    ...


    // déclaration des attributs
```

```
    // -------------------------

    // attributs relatifs au contrôleur de dialogue
    private Stage eStage ;
    private ExampleViewController eViewController ;

    // attributs relatifs au Modèle
    ...


    /**
     * Constructeur : charge la fenêtre d'exemple.
     */
    public Example(Stage _parentStage)
    {
        try
        {
            // initialisation des attributs relatifs au Modèle
            ...

            // initialisation d'un nouveau stage pour la fenêtre d'exemple
            this.eStage = new Stage() ;
            this.eStage.initOwner(_parentStage) ;
            this.eStage.initModality(Modality.WINDOW_MODAL) ;

            // chargement de la vue FXML de la fenêtre d'exemple
            FXMLLoader fxmlLoader = new
FXMLLoader(ExampleViewController.class.getResource("example.fxml")) ;

            // initialisation de la scène
            Scene scene = new Scene(fxmlLoader.load(), MIN_WINDOW_WIDTH,
MIN_WINDOW_HEIGHT) ;
            this.eStage.setScene(scene) ;
            this.eStage.setTitle("Exemple") ;

            // initialisation du contrôleur de vue
            this.eViewController = fxmlLoader.getController() ;
            this.eViewController.setStage(this.eStage) ;
            this.eViewController.setCffDialogController(this) ;
            this.eViewController.initializeView() ;

            // application des styles à la scène

this.eStage.getScene().getStylesheets().add(getClass().getResource("/application/style
/e.css").toExternalForm()) ;
        }
        catch (Exception e)
        {
            e.printStackTrace() ;
        }
    }
```

```
    // accesseurs
    // ----------

    ...


    // méthodes publiques de gestion du dialogue
    // -----------------------------------------

    /**
     * Effectue le dialogue d'exemple.
     */
    public void doExampleDialog() { this.eViewController.displayDialog() ; }

    ...


    // méthodes privées
    // ----------------

    ...
}
```

Contrôleurs de vue

```
package application.view ;

import application.control.Example ;

...

public class ExampleViewController
{
    // déclaration des constantes
    // --------------------------

    ...


    // déclaration des attributs
    // -------------------------

    // attributs relatifs au contrôleur de vue
    private Stage stage ;
    private Example eDialogController ;

    // attributs relatifs au Modèle
    ...
```

```
// éléments graphiques de la vue FXML (ordonnés par ordre d'apparition)
// ------------------------------------------------------------------

@FXML private ... ;
...


// méthodes d'initialisation du contrôleur de vue
// ----------------------------------------------

/**
 * Définit le stage de la vue.
 * @param _stage    un stage
 */
public void setStage(Stage _stage)
{
    this.stage = _stage ;
}

/**
 * Définit le contrôleur de dialogue de la vue.
 * @param _eDialogController  un contrôleur de dialogue
 */
public void setEDialogController(Example _eDialogController)
{
    this.eDialogController = _eDialogController ;
}

/**
 * Initialise la vue.
 */
public void initializeView()
{
    // initialisation des éléments graphiques de la vue
    ...
}

/**
 * Affiche la fenêtre.
 */
public void displayDialog()
{
    this.stage.showAndWait() ;
}

/**
 * Gère la fermeture de la fenêtre.
 * @param e un évènement de fenêtre
 */
```

```
    private void closeWindow(WindowEvent e)
    {
        this.doClose() ;
        e.consume() ;
    }


    // méthodes de traitement des actions utilisateur
    // ----------------------------------------------

    /**
     * Ferme la fenêtre.
     */
    @FXML
    private void doClose()
    {
        this.stage.close() ;
    }


    // méthodes privées
    // ----------------

    ...
}
```

2.4.4 Threads utilisés

Thread de récupération des données

Le thread `CsvReaderTask` lis, à intervalle de temps régulier (cf. fréquence définie dans la configuration), le fichier de données global (`data.csv`) et le fichier d'alertes (`alert.csv`). Il notifie ensuite le contrôleur de dialogue du tableau de bord `DataVisualisationPane` que les données ont été actualisées.

Méthode de lecture du fichier de données global

```
/**
 * Lis le fichier de données.
 */
private void readDataFile()
{
    Map<String, Map<String, String>> dataMap = new HashMap<>() ;
    try (CSVReader csvReader = new CSVReaderBuilder(new
FileReader(DataLoader.getAllRoomDataFile()))
            .withCSVParser(new CSVParserBuilder().withSeparator(delimiter).build())
            .build()
    ) {
        String[] header = csvReader.readNext() ;
```

```
        String[] values = null ;
        while ((values = csvReader.readNext()) != null && !values[0].equals(""))
        {
            dataMap.put(values[0], new HashMap<String, String>()) ;
            for (int i = 1; i < values.length; i++)
            {
                dataMap.get(values[0]).put(header[i], values[i]) ;
            }
        }
        this.dvpDialogController.setDataMap(dataMap) ; // LE CONTRÔLEUR DE DIALOGUE
EST NOTIFIÉ DE LA MISE À JOUR
    }
    catch (FileNotFoundException e) { throw new RuntimeException(e) ; }
    catch (IOException e) { throw new RuntimeException(e) ; }
    catch (CsvValidationException e) { throw new RuntimeException(e) ; }
}
```

Méthode de lecture du fichier d'alertes

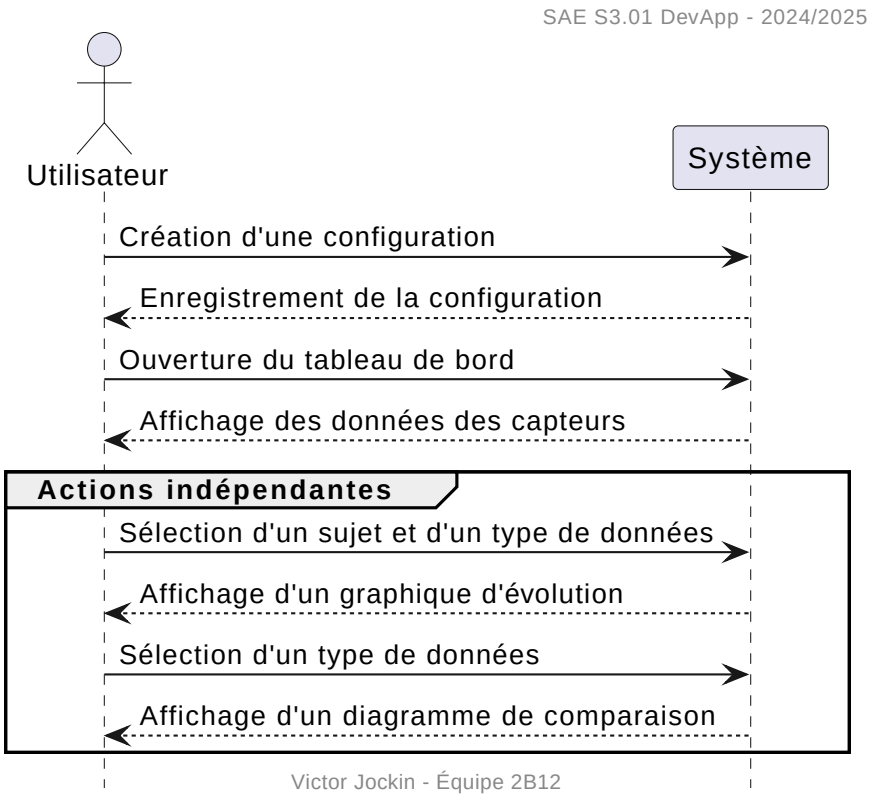
```
/**
 * Lis le fichier d'alertes.
 */
private void readAlertFile()
{
    Map<String, Map<String, String>> alertMap = new HashMap<>() ;
    try (CSVReader csvReader = new CSVReaderBuilder(new
FileReader(DataLoader.getAlertFile()))
            .withCSVParser(new CSVParserBuilder().withSeparator(delimiter).build())
            .build()
    ) {
        String[] header = csvReader.readNext() ;
        String[] values = null ;
        while ((values = csvReader.readNext()) != null && !values[0].equals(""))
        {
            alertMap.put(values[0], new HashMap<String, String>()) ;
            for (int i = 1; i < values.length; i++)
            {
                alertMap.get(values[0]).put(header[i], values[i]) ;
            }
        }
        this.dvpDialogController.setAlertMap(alertMap) ; // LE CONTRÔLEUR DE DIALOGUE
EST NOTIFIÉ DE LA MISE À JOUR
    }
    catch (FileNotFoundException e) { throw new RuntimeException(e) ; }
    catch (IOException e) { throw new RuntimeException(e) ; }
    catch (CsvValidationException e) { throw new RuntimeException(e) ; }
}
```

Threads de mise à jour de l'affichage

Les threads `UpdateDataDisplayTask` et `UpdateAlertDisplayTask` sont appelés à chaque notification d'actualisation des données de la part du thread de récupération des données (`CsvReaderTask`). Ils sont respectivement chargés de la mise à jour de l'affichage des données et des alertes dans le tableau de bord.

3. Conception et mise en oeuvre des fonctionnalités

3.1 Diagramme de Séquence Système

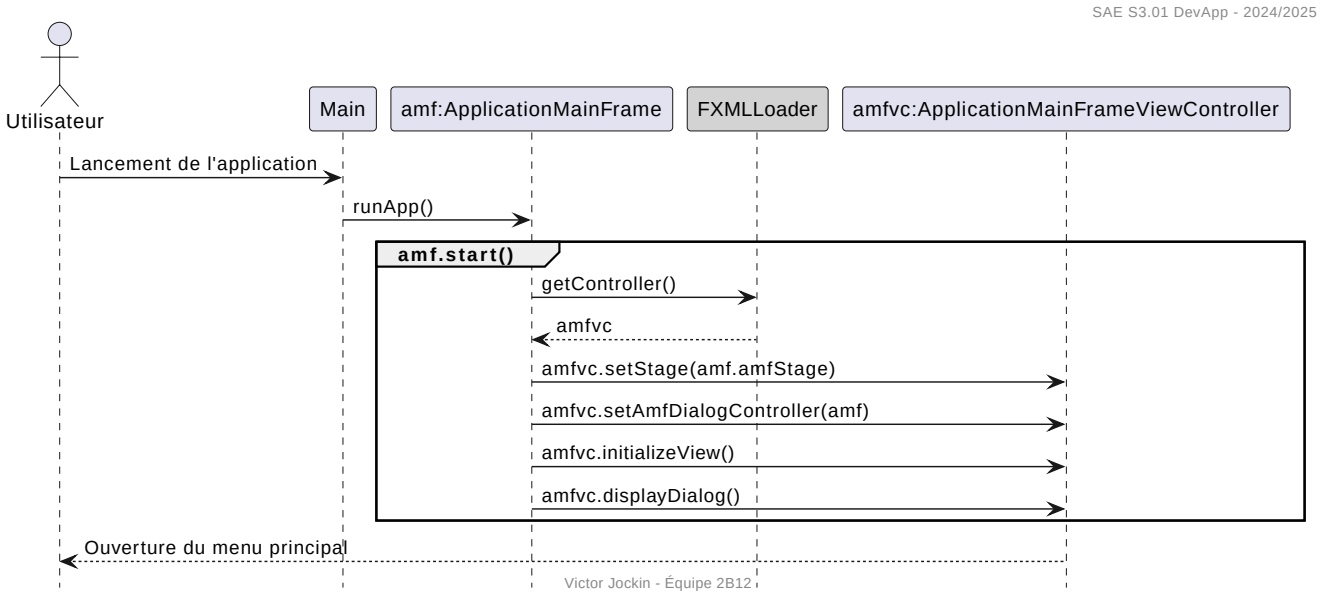


3.2 Implémentation des fonctionnalités

3.2.1 Menu principal

Ouverture du menu principal

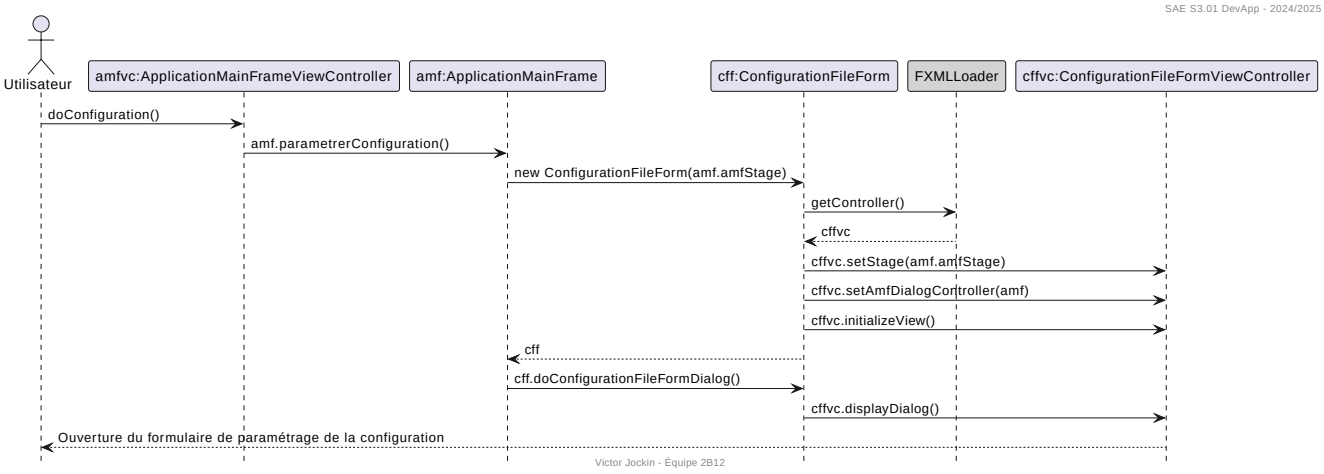
Diagramme de Séquence



3.2.2 Formulaire de paramétrage de la configuration

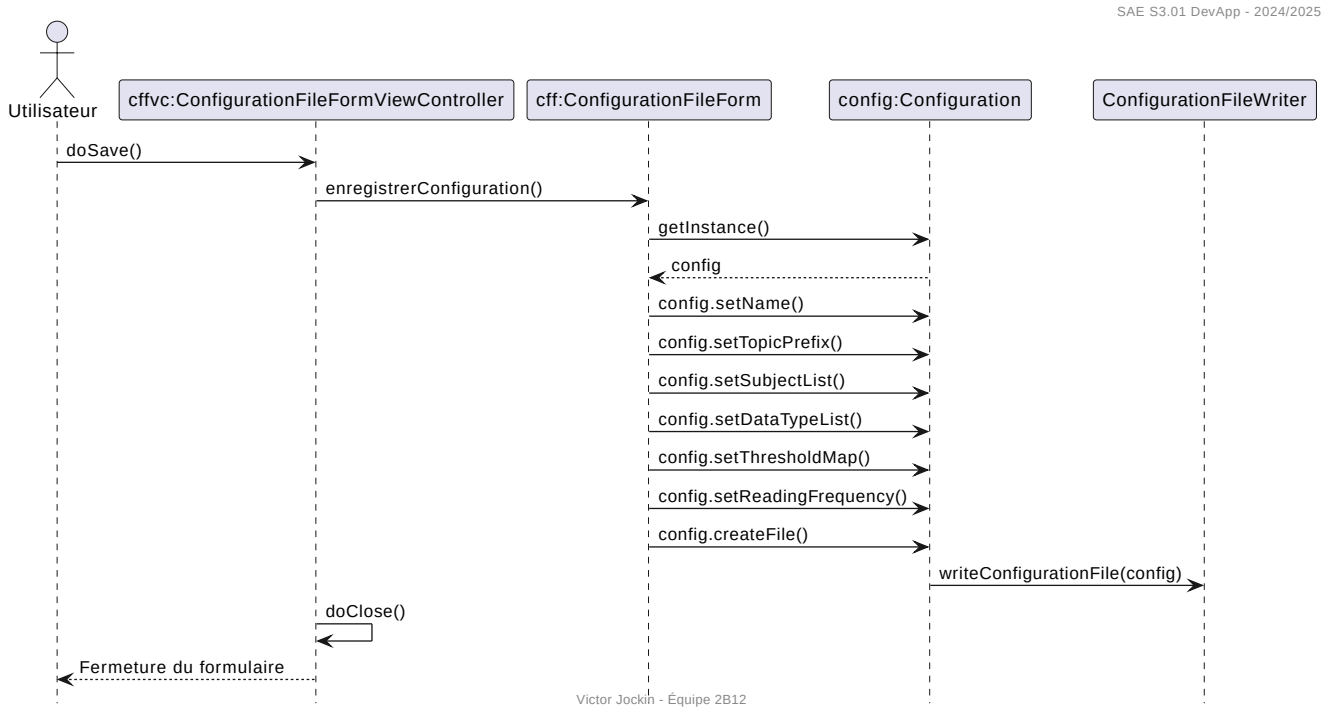
Ouverture du formulaire

Diagramme de Séquence



Enregistrement d'une configuration

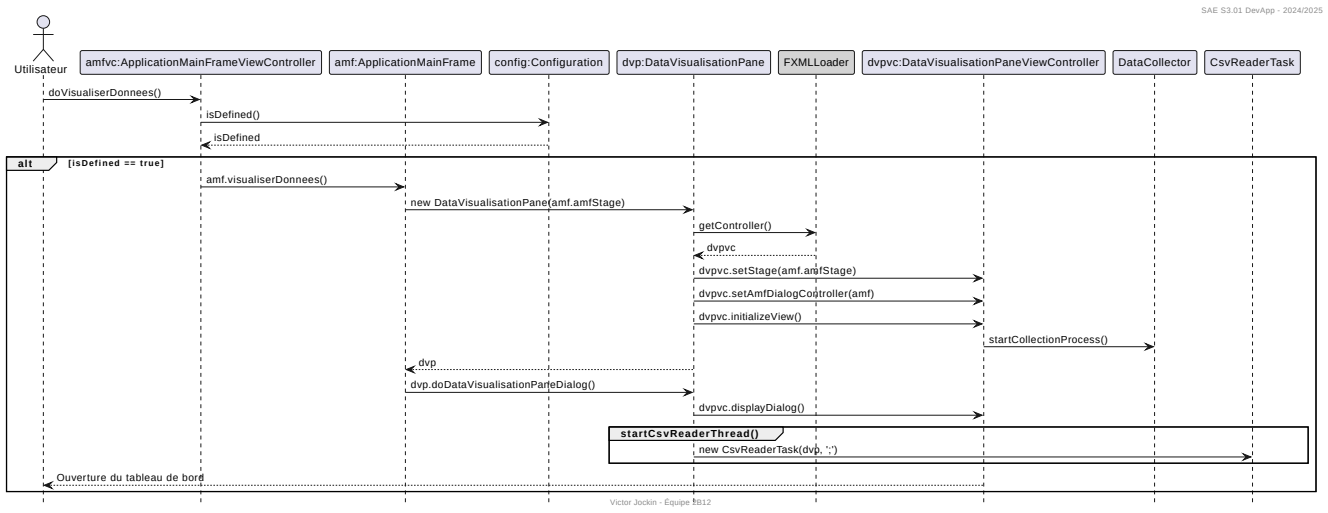
Diagramme de Séquence



3.2.3 Tableau de bord

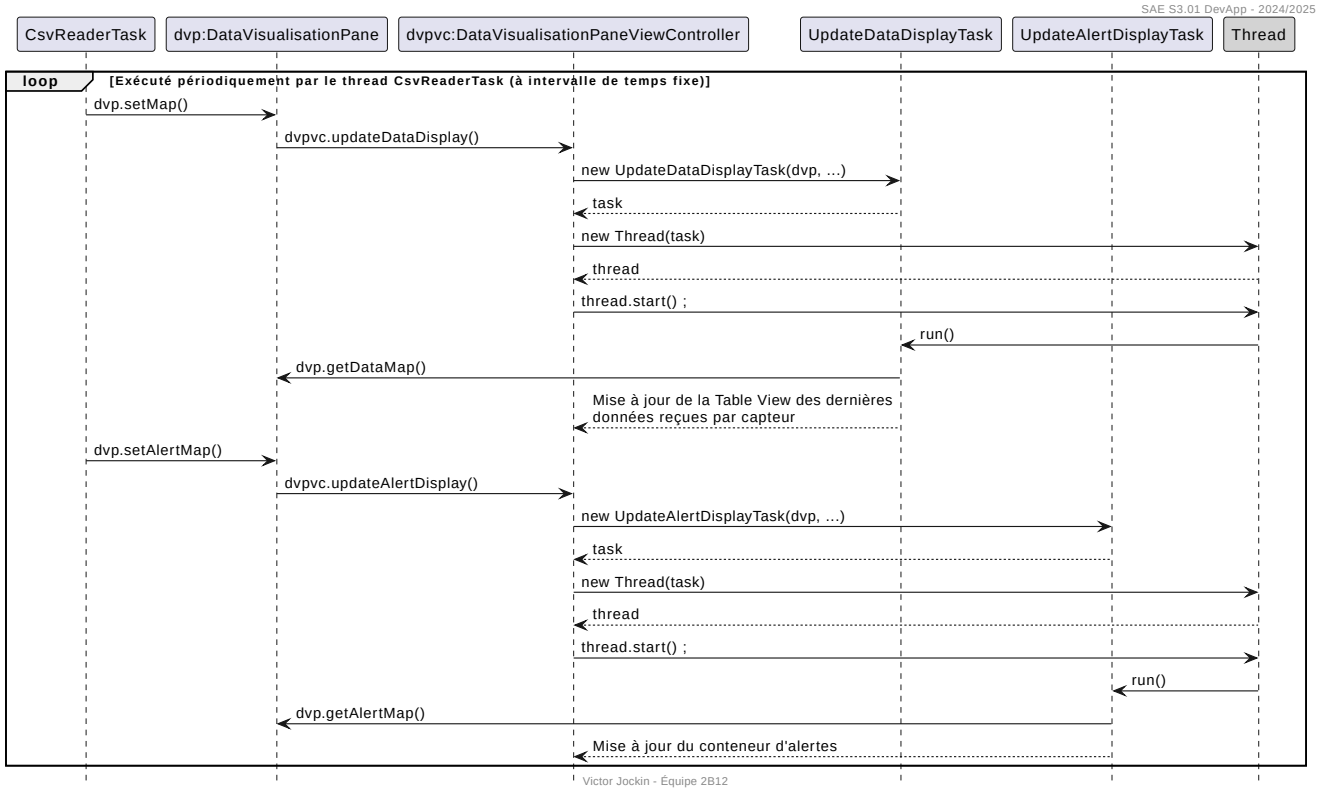
Ouverture du tableau de bord

Diagramme de Séquence



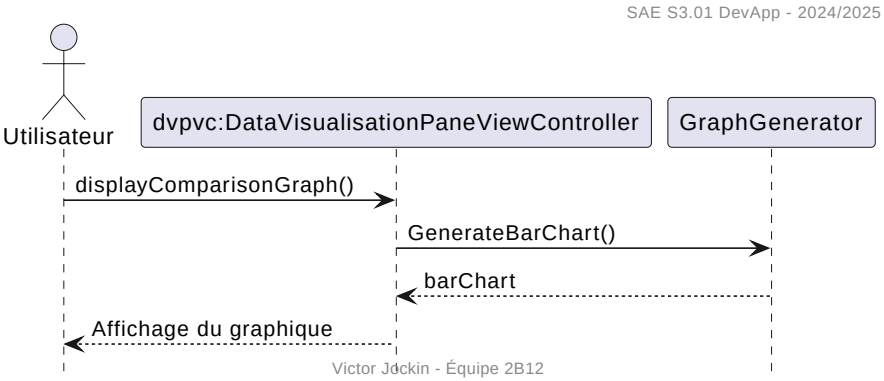
Mise à jour automatique des données affichées

Diagramme de Séquence



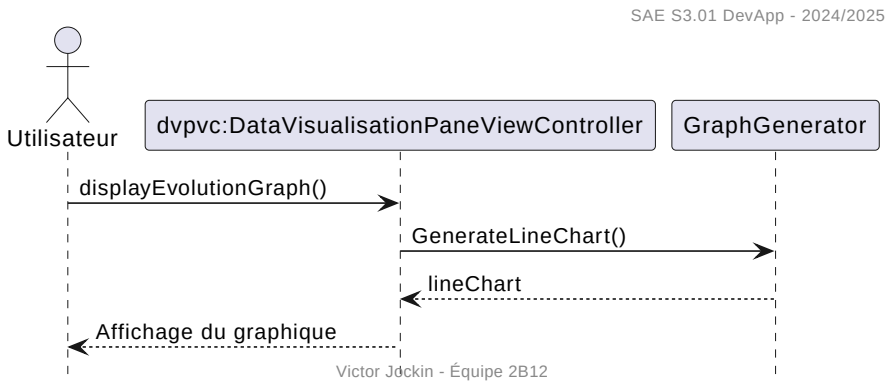
Affichage d'un graphique de comparaison

Diagramme de Séquence



Affichage d'un graphique d'évolution

Diagramme de Séquence



4. Procédures d'installation

4.1 Installation pour développement

4.1.1 Prérequis

1. **Installer l'environnement de développement Java**
   - Télécharger le **JDK 17** (ou version compatible) depuis le site officiel d'Oracle : Oracle - Java Downloads.
   - Installer le JDK en suivant les instructions indiquées par l'installateur.
   - Si nécessaire, ajouter le chemin vers le JDK à la variable d'environnement `PATH`.
   - Dans un terminal, vérifier l'installation avec la commande `java -version` ou `java --version`.
2. **Installer Apache Maven**
   - Télécharger **Maven** (archive ZIP) depuis le site officiel d'Apache Maven : Apache Maven Project - Downloading Apache Maven.
     - Pour une installation sur Linux ou Mac OS, télécharger la ***Binary tar.gz archive***.
     - Pour une installation sur Windows, télécharger la ***Binary zip archive***.
   - Ajouter le chemin vers Maven à la variable d'environnement `PATH`.
   - Dans un terminal, vérifier l'installation avec la commande `mvn -version`, `mvn --version` ou `mvn -v`.
3. **Configurer un IDE**
   - Si nécessaire, installer des plugins de prise en charge de **Maven** et **JavaFX** dans l'IDE utilisé pour le développement.

4.1.2 Étapes d'installation

1. **Cloner le dépôt du projet**
   - Accéder au dépôt GitHub du projet : GitHub - SAE S3.01 DevApp
   - Cloner le dépôt du projet via la commande :

     ```
     git clone https://github.com/IUT-Blagnac/sae-3-01-devapp-2024-2025-g2b12.git
     ```

   - Accéder au répertoire du projet Java situé sous `solution iot/application_iot` via la commande :

     ```
     cd solution\ iot/application_iot
     ```

2. **Construire le projet avec Maven**
   - Supprimer les fichiers et ressources précédemment compilés avec la commande `mvn clean`

puis compiler le projet Java via la commande `mvn install`. Il est également possible d'utiliser directement la commande `mvn clean install`.

3. **Exécuter l'application depuis Maven**
   - Exécuter le projet JavaFX via la commande `mvn javafx:run`.

4.2 Installation pour utilisation

4.2.1 Prérequis

1. **Installer le Java Runtime Environement (JRE)**
   - Vérifier que Java est installé sur la machine en exécutant la commande `java -version` dans un terminal.
   - Si Java n'est pas installé, télécharger et installer le **JRE 8** ou version ultérieure depuis le site officiel de Java : Java - Télécharger Java.
2. **Installer Python 3**
   - Vérifier que Python en version 3 est installé sur la machine en exécutant la commande `python -version` ou `python3 -version` dans un terminal.
   - Si Python n'est pas installé, télécharger et installer la dernière version disponible sur le site officiel de Python : Python - Downloads.

4.2.2 Étapes d'installation

1. **Télécharger l'application**
   - Télécharger l'archive de l'application (fichier ZIP) située sous le répertoire `livrables/IoT` du dépôt GitHub du projet : GitHub - Livrables IoT
     - Pour une installation sur Mac OS, préférer l'archive `application_jar_mac_os.zip`.
     - Pour une installation sur Windows ou Linux, préférer l'archive `application_jar_windows.zip`.
2. **Décompresser l'archive de l'application**
   - Décompresser l'archive téléchargée dans un répertoire à l'aide d'un outil de décompression tel que **WinRAR** ou **7-Zip**.
   - L'arborescence de l'application après décompression doit ressembler à ceci :

```
application/
|-- ressources/
|   |-- data/
|   |-- configuration.ini
|   |-- mqtt.py
|-- application_iot-1.0-SNAPSHOT-shaded.jar
```

4.2.3 Étapes de lancement

1. **Lancer l'application dans le gestionnaire de fichiers**
   - Lancer l'exécutable `application_iot-1.0-SNAPSHOT-shaded.jar` en double-cliquant sur celui-ci.
   - *Le menu principal de l'application devrait alors apparaître à l'écran.*
2. **Lancer l'application en ligne de commande**
   - Ouvrir un terminal et se placer dans le répertoire `application` à l'aide de la commande `cd`.
   - Lancer ensuite l'exécutable de l'application via la commande :

     ```
     java -jar application_iot-1.0-SNAPSHOT-shaded.jar
     ```

   - *Le menu principal de l'application devrait alors apparaître à l'écran.*

Étudiants
Nolhan Biblocque
Léo Guinvarc’h
Victor Jockin
Mathys Laguilliez
Mucahit Lekesiz

Enseignant
André Péninou

Formation
BUT Informatique
2ème Année
Promotion 2024-2025

Établissement
IUT de Blagnac
Université Toulouse II – Jean Jaurès (31)