Rapport de Projet Java

Etude et développement d'une application de l'Analyse et Visualisation des Données

Filière : Informatique et Ingénierie de données

Projet de Java

Effectué dans l'école des sciences appliquées Khouribga

rendu le : 15 Mai 2025

Réalisé par :

AMINE KOULA

Encadré par :

Pr. Noreddine GHERABI

Remerciements

Nous tenons à exprimer notre gratitude à Monsieur Noreddine GHERABI,

pour son encadrement précieux tout au long de ce projet. Ses conseils avisés et son soutien ont été déterminants dans la résolution des défis techniques, notamment lors de l'intégration de JavaFX et de la gestion des données.

Nos remerciements s'adressent également à l'ENSA Khouribga pour les ressources mises à disposition qui ont facilité la réalisation de ce travail.

Résumé

Ce projet, développé dans le cadre du module de POO, vise à fournir une solution logicielle pour analyser des données commerciales (issues de fichiers CSV ou d'une base de données MySQL) et générer des visualisations interactives. L'application permet le chargement, l'affichage, la manipulation (ajout, modification, suppression) et l'analyse des ventes à travers une interface graphique conviviale développée avec JavaFX. Elle est structurée selon le modèle MVC, favorisant une séparation claire entre la logique métier, l'interface utilisateur et la gestion des données.

Le système intègre également des fonctionnalités de navigation entre scènes, une gestion dynamique des données partagées, et un tableau de bord qui fournit des indicateurs clés comme le total des ventes, la quantité vendue et le nombre de catégories de produits. L'objectif principal est de doter l'utilisateur d'un outil simple mais puissant pour explorer et comprendre ses données commerciales, tout en mettant en pratique les principes de la programmation orientée objet, la modularité et la réutilisabilité du code.

Sommaire

Table des matières

Chapitre1 : Introduction

Introduction

Dans un contexte économique de plus en plus axé sur la donnée, la capacité à collecter, analyser et exploiter efficacement des informations commerciales devient un atout stratégique pour toute entreprise. Le développement de solutions logicielles permettant d'automatiser ces tâches s'impose alors comme une nécessité. C'est dans cette optique que s'inscrit le présent projet, réalisé dans le cadre du module de Programmation Orientée Objet (POO).

L'objectif principal de ce projet est de concevoir et développer une application de bureau en Java, à l'aide de JavaFX, capable de charger, manipuler, visualiser et analyser des données de ventes provenant de différentes sources, telles que des fichiers CSV ou une base de données relationnelle MySQL. Le projet met en œuvre les concepts fondamentaux de la programmation orientée objet — tels que l'encapsulation, l'héritage, le polymorphisme et l'abstraction — tout en appliquant de bonnes pratiques de structuration logicielle comme l'architecture MVC (Modèle-Vue-Contrôleur).

À travers cette application, l'utilisateur peut facilement explorer les données, effectuer des opérations de modification (ajout, suppression, mise à jour) et obtenir des indicateurs de performance commerciale via un tableau de bord interactif. Ce travail constitue ainsi une mise en application concrète des compétences acquises en développement Java, tout en répondant à un besoin réel d'analyse de données commerciales.

I. Objective de projet

L'objectif principal de ce projet est de développer une application de bureau permettant la gestion, l'analyse et la visualisation de données de ventes. L'application offre à l'utilisateur la possibilité de charger des données depuis un fichier CSV ou une base de données MySQL, de les afficher sous forme de tableaux, de les manipuler (ajout, modification, suppression), et de générer des rapports exportables au format PDF. Elle

intègre également un tableau de bord interactif qui présente des indicateurs clés tels que le chiffre d'affaires total, la quantité vendue et le nombre de catégories distinctes.

II. Technologies utilisées :

Pour atteindre ces objectifs, plusieurs technologies ont été utilisées :

- Java (JDK) : langage principal du projet, utilisé pour la programmation orientée objet et la logique métier.
- JavaFX : utilisé pour créer l'interface graphique (GUI) moderne, dynamique et interactive.
- JDBC : permet la connexion et la communication avec la base de données MySQL pour la lecture et la manipulation des données.
- iText : bibliothèque utilisée pour la génération de rapports PDF contenant des informations commerciales.
- Maven : outil de gestion de projet et de dépendances, facilitant la compilation, le packaging et l'intégration des bibliothèques externes.

Chapitre 2 : Analyse des Besoins

I. Fonctionnalités Principales :

1. Importation des données :

- L'application prend en charge le chargement de fichiers CSV (exemple : sample_sales.csv) ou l'extraction directe depuis une base de données MySQL via des requêtes SQL.
- Un système de validation assure l'intégrité des données (ex : vérification des formats de date et des montants).

2. Analyse des données :

- Calcul des ventes totales par catégorie (ex : Informatique vs Électronique).
- Identification des produits phares (plus vendus, plus chers) et des tendances mensuelles.
-

3. Visualisation

- Des graphiques dynamiques (histogrammes, courbes) générés avec JavaFX permettent une analyse intuitive.

4. Sécurité

- Un système d'authentification (login/mot de passe) protège l'accès à l'application.

5. Exportation des données :

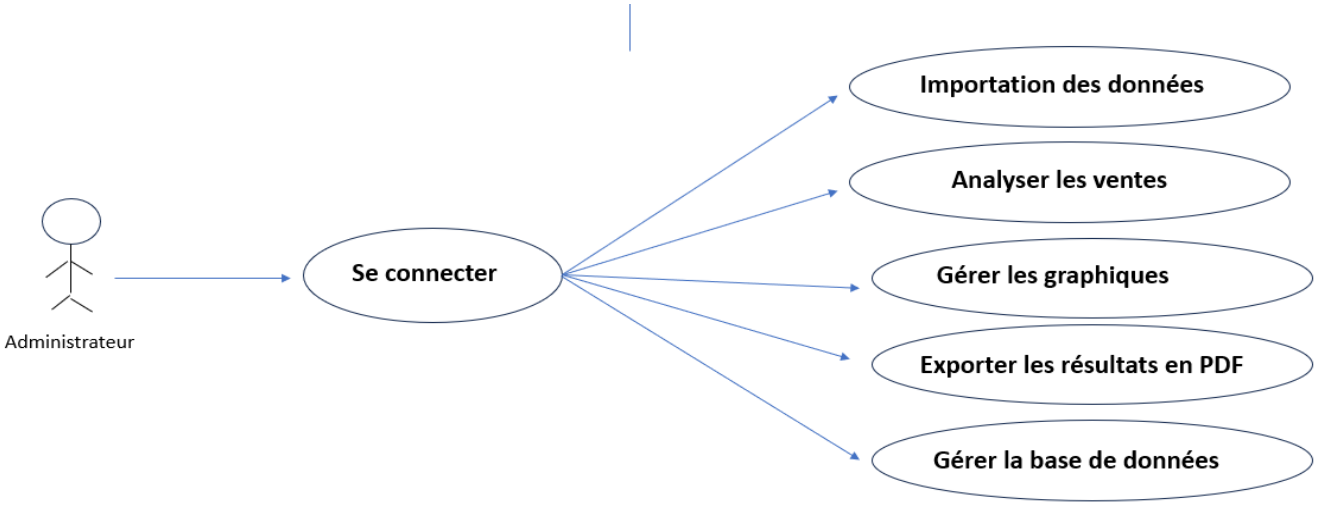
- Exporter les diagrammes et les analyses qui sont faites dans l’application sous forme d’un rapport PDF dans le dossier du projet.

Résumé sous frome Tableau :

Fonctionnalités	Description
Importation des données	Charger les données depuis fichier CSV, ou les charger depuis Base de données Mysql
Analyses des ventes	Calcul les totaux par categorie, tendance mensuelles, produits plus vendus..
Visualisation	Générer des graphiques(charts,pie,line..) en utilisant javafx
Export des résultats	Sauvegarder les analyses en PDF (iText).
Authentification	Système de login/mot de passe pour sécuriser l'accès.

II. Diagramme des cas d'utilisation :

Pour passer à utiliser l'application il faut d'abord s'identifie par Username et mot pass . Seuls les administrateurs peuvent s'enregistrer car l'application permet d'importer des données de ventes depuis un CSV ou une base MySQL, d'analyser les tendances (par catégorie, mois, etc.), et d'exporter les résultats. L'accès est sécurisé par un système de login/mot de passe.



"L'application est conçue pour un seul type d'utilisateur : **l'Administrateur***.

- *Aucun autre rôle (comme "utilisateur standard" ou "invité") n'est prévu.*
- *Cette restriction garantit un contrôle total sur les données sensibles et les fonctionnalités critiques (import/export, gestion de la base de données)."*

Chapitre 3 : Conception du Système :

I. Architecture Logicielle (MVC) :

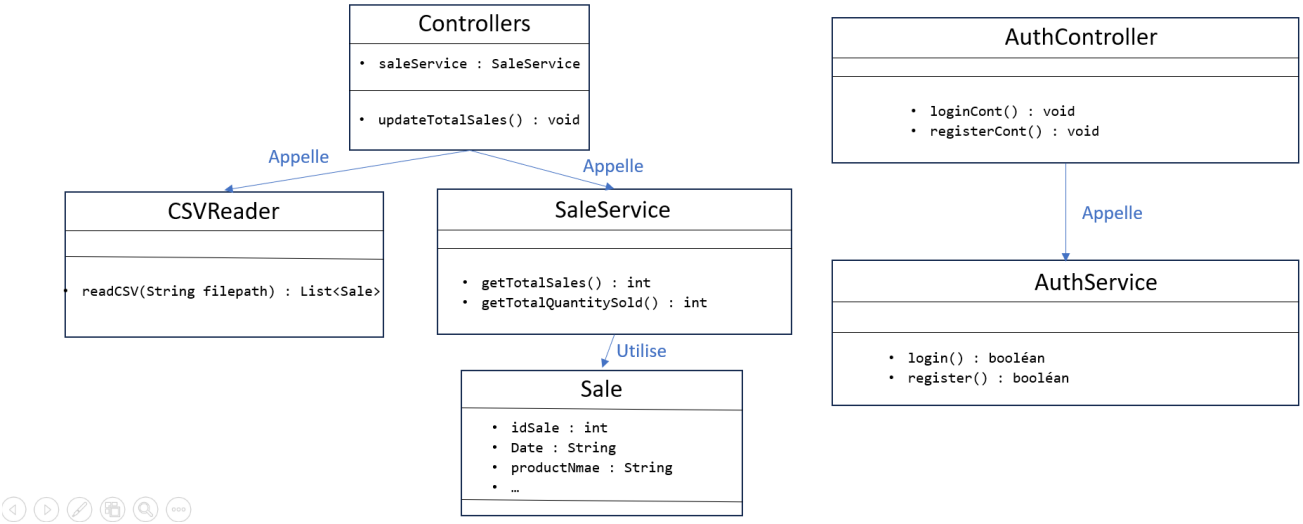
- **Model** : Gere les données et la logique métier.
    - Classes Sale pour représenter les données.
    - Classe SaleService pour les opération sur analyse sur Sale.
- **Vue :** Interfaces graphiques (FXML)
    - Exemples : main.fxml , DataManipulation.fxml
- **Controleur** : lie le modèle et la vue

- Classes : AuthController, MainController

Un Schéma pour bien savoir comment il fonctionne :

II. Diagramme de classes :

La figure suivante représente le diagramme des classes importante de notre application :



III. Gestion des données :

1. Configuration de la Base de données :

L'application repose sur une base de données MySQL configurée (Dans classe DBConfig) pour stocker efficacement les données de ventes et les informations d'authentification. La structure comprend deux tables principales : la table ventes (**Sale**) pour enregistrer toutes les transactions commerciales (incluant les détails des produits, quantités, prix et dates), et la table utilisateurs (**login**) pour gérer les accès sécurisés à l'application. La connexion à la base est établie via JDBC. Les requêtes SQL utilisent des PreparedStatement dans une classe SaleRepository pour prévenir les injections. Cette architecture robuste et sécurisée permet une gestion fiable des données tout au long du processus métier.

Table Sale Contient toutes les informations relatives aux ventes, avec les champs suivants :

- **Id_sale** : (int,clé primaire)
- **date** : (String)
- **product** : (String)
- **category** : (String)
- **quantity** : (String)
- **price** : (float)
- **total_price** : (float)

Table login Gère l'authentification des utilisateurs avec :

- **id :** (auto incrimentation,clé primaire)
- **first_name** : (String)
- **last_name** : (String)
- **username** : (String)
- **email** : (String)
- **password** : (String)
- **email** : (float)

2. Modélisation de données:

La classe Sale constitue la pierre angulaire du modèle de données de l'application, représentant de manière exhaustive le concept métier d'une transaction commerciale. Conçue selon les principes fondamentaux de la programmation orientée objet, cette classe encapsule sept attributs essentiels soigneusement choisis pour couvrir tous les aspects d'une vente : un identifiant unique (idSale) garantissant la traçabilité des transactions, la date de l'opération (stockée sous forme de chaîne de caractères pour une flexibilité dans le formatage), les informations produit (nom et catégorie), ainsi que les données quantitatives (quantité, prix unitaire et montant total).

Cette modélisation présente plusieurs avantages architecturaux notables. D'une part, elle reflète parfaitement le schéma relationnel de la table sale en base de données et au structure de données au fichier CSV, facilitant ainsi les opérations de persistance via JDBC. D'autre part, elle offre une grande flexibilité grâce à ses deux constructeurs (paramétré et par défaut) et à ses méthodes d'accès (getters/setters) qui assurent un contrôle fin de l'encapsulation tout en permettant des validations métier si nécessaire. La méthode toString() surchargée fournit quant à elle un outil précieux pour le débogage et la génération de rapports textuels.

D'un point de vue métier, cette implémentation permet de répondre à tous les cas d'usage identifiés lors de l'analyse des besoins : calcul du chiffre d'affaires par catégorie, identification des produits les plus vendus, ou encore analyse des tendances temporelles. La structure des données a été volontairement conçue pour faciliter les extensions futures, comme l'ajout de remises commerciales ou de taxes, sans nécessiter de refonte majeure de l'architecture. Cette robustesse, combinée à une parfaite adéquation avec les standards Java, fait de la classe Sale un composant à la fois performant, maintenable et parfaitement intégré dans l'écosystème technique de l'application (JavaFX pour l'affichage, JDBC pour la persistance).

3. Gestion des Erreurs et Robustesse de l'Application:

Le système intègre une stratégie complète de gestion des erreurs couvrant à la fois le module d'authentification et la gestion des ventes, garantissant ainsi la fiabilité et la sécurité de l'application. Pour l'authentification (AuthService), plusieurs mécanismes ont été mis en œuvre : une validation rigoureuse des champs obligatoires (login/mot de passe) avant toute requête SQL, des messages d'erreur clairs et localisés en français pour guider l'utilisateur ("Connexion échouée, veuillez vérifier vos informations"), et un logging technique détaillé (via e.printStackTrace()) permettant un débogage efficace. Les requêtes utilisent systématiquement des PreparedStatement pour éliminer tout risque d'injection SQL, tandis que les messages d'erreur génériques en cas de panne serveur évitent de divulguer des informations sensibles.

Côté gestion des ventes (SaleRepository), la logique CRUD est renforcée par des contrôles d'intégrité des données : vérification de l'unicité des IDs à l'insertion, gestion implicite des transactions pour assurer la cohérence des données, et feedback utilisateur précis via des

alertes JavaFX contextualisées ("Vente #123 ajoutée avec succès" ou "Erreur : ID déjà existant"). Chaque exception SQL est systématiquement capturée et loguée avec son contexte (ID concerné, type d'opération), facilitant la maintenance.

Cette approche unifiée repose sur un pattern commun :

- Tentative d'exécution des opérations dans un bloc try
- Capture explicite des SQLException
- Double canal de notification :
- Aux utilisateurs : messages simples et actionnables via Alert
- Aux développeurs : logs détaillés avec stacktrace et contexte métier

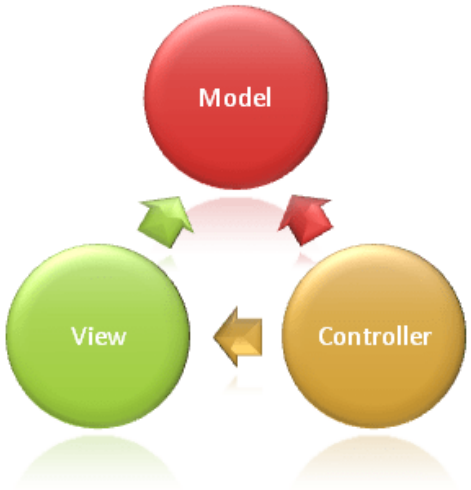
Les résultats montrent une application résiliente aux pannes (tentatives de connexion invalides, conflits de données) tout en offrant une expérience utilisateur fluide. Cette implémentation suit les bonnes pratiques de sécurité (OWASP) et de robustesse (design for failure), comme en témoignent les choix techniques : isolation des erreurs métier/techniques, journalisation structurée, et rollback automatique des transactions. Des pistes d'amélioration pourraient inclure l'ajout de codes d'erreur standardisés et la traçabilité complète des opérations via un système de logging centralisé.

Chapitre 4 : Réalisation :

I. Présentation de l'application Interface:

Cette application utilise un outil de gestion et d'automatisation de production des projets logiciels java en général (**Maven**) . Il simplifie la gestion des dépendances en automatisant le téléchargement des bibliothèques externes, comme javaFX, Mysql connector , via le fichier pom.xml. Ensuite, Maven standardise la structure du projet avec des répertoires dédiés (src/main/java), ce qui facilite la collaboration et la maintenance.

L'architecture **Modèle-Vue-Contrôleur (MVC)** a été adoptée pour organiser le code de manière claire et modulaire en séparant (models,controllers,views). Cette séparation des responsabilités améliore la lisibilité du code, simplifie les tests unitaires et permet des mises à jour futures sans tout réécrire. Par exemple, si je veux changer l'interface graphique, je n'ai pas à modifier la logique métier.



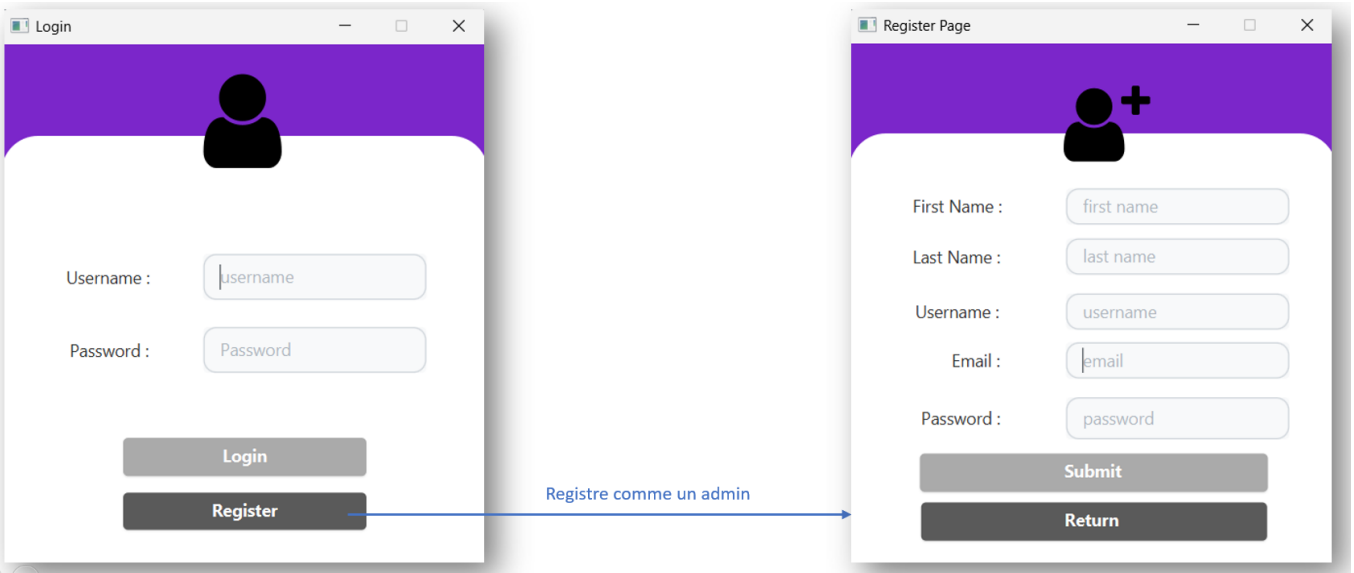
J'ai développé cette application avec **Visual Studio Code (VS Code)** pour sa légèreté et ses extensions puissantes.

Représentation de Architecture de notre application :

```
src/
├── main/
│   ├── java/
│   │   └── com/
│   │       └── example/
│   │           ├── App.java                  # Point d'entrée principal
│   │           ├── config/                   # Configuration
│   │           │   ├── DBConfig.java   # Paramètres de connexion DB
│   │           ├── controllers/
│   │           │   ├── MainController.java   # Contrôleur principal
│   │           │   ├── AuthController.java   # Gestion authentification
│   │           │   ├── ChartController.java   # affichage charts
│   │           │   ├── DataManController.java # Gestion données
│   │           ├── models/
│   │           │   ├── Sale.java             # Modèle de données Vente
│   │           │   ├── services/
│   │           │   │   ├── AuthService.java # Service d'authentification
│   │           │   │   ├── SaleService.java # Gestion des ventes
│   │           │   ├── repositories/
│   │           │   │   ├── SaleRepository.java # Accès DB pour les ventes et user
│   │           │   └── utils/
│   │           │       ├── CSVReader.java    # Lecture fichiers CSV
│   │           │       ├── PDFExporter.java  # Export PDF
│   │           │       ├── ChartGenerator.java # Génération graphiques
│   │           │       ├── SceneManager.java # changer les scenes
│   │           │       └── ChatImg.java # pour capturer les charts
│   │           └── views/
│   │           ├── MainApp.java # ici ou on lance l'app
│   ├── resources/
│   │   ├── com/example/views/
│   │   │   ├── main.fxml                   # Interface principale
│   │   │   ├── login.fxml                 # Ecran de connexion
│   │   │   ├── register.fxml              # Ecran register
│   │   │   ├── DataManipulation.fxml       # manipulation de données
│   │   │   ├── ChartAnalysis.fxml          # Ecran affiche les charts
```

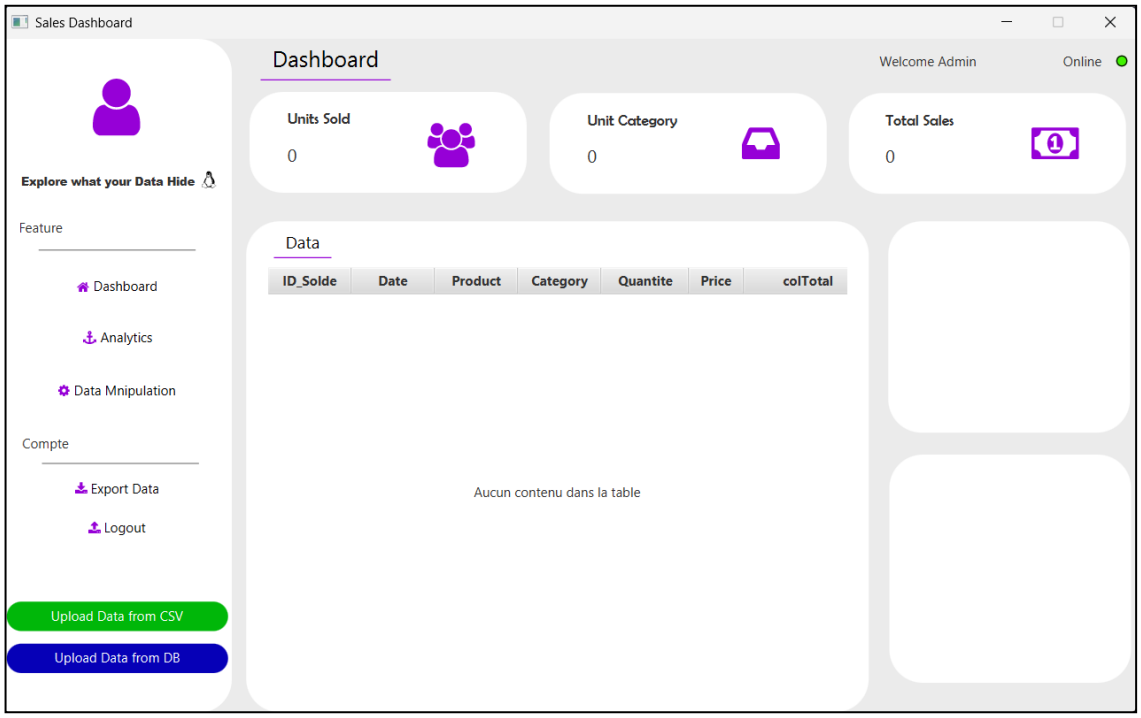
i. Connexion au application :

C’est la première page qui s’affiche, elle permet aux utilisateurs de se connecter :

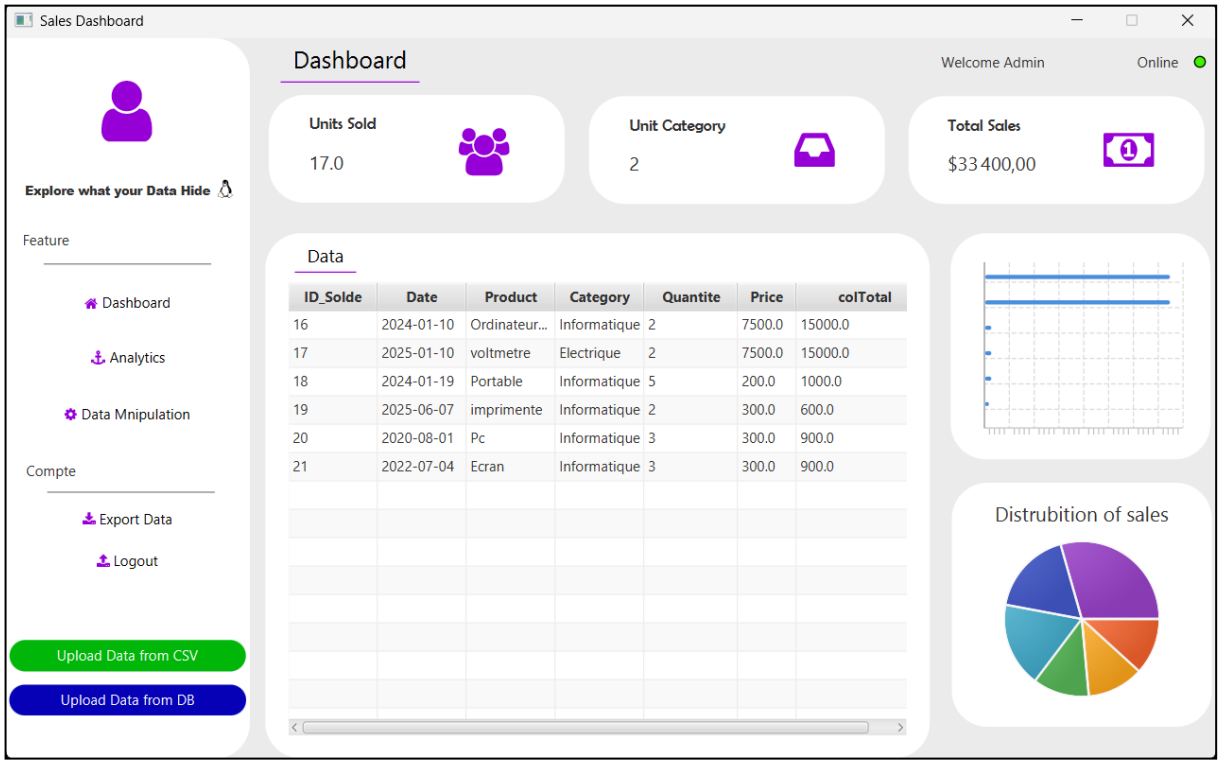


ii. Tableau de bord principal :

Apres la connexion il s'affiche un tableau de bord vide, il faut importer les donner soit depuis

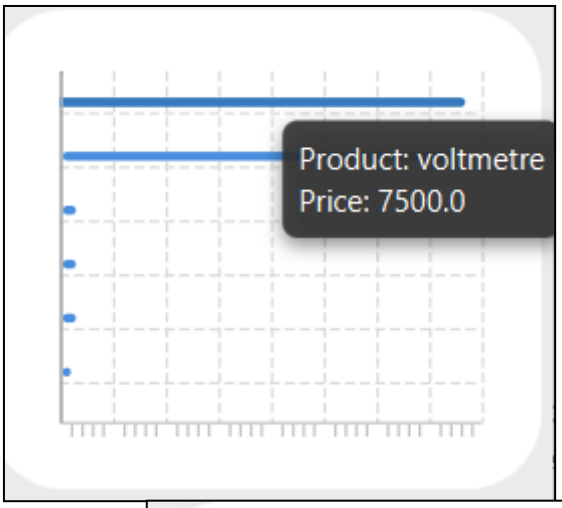


Apres avoir importer notre données les calcul sont faites (units sold,unit category, total sales) et les graphs s'affiche.

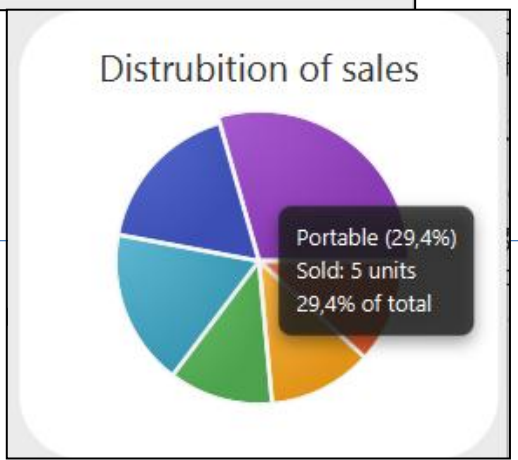


En peut aller au page manipulation des données **(Data Manipulation)** ou la page des analyses **(Analytics)**

- **Units Sold** : Représente Combien de produits vendus au total.
- **Unit Category** : Représente combien on a de catégorie.
- **Total Sales** : Représente la somme de prix total.



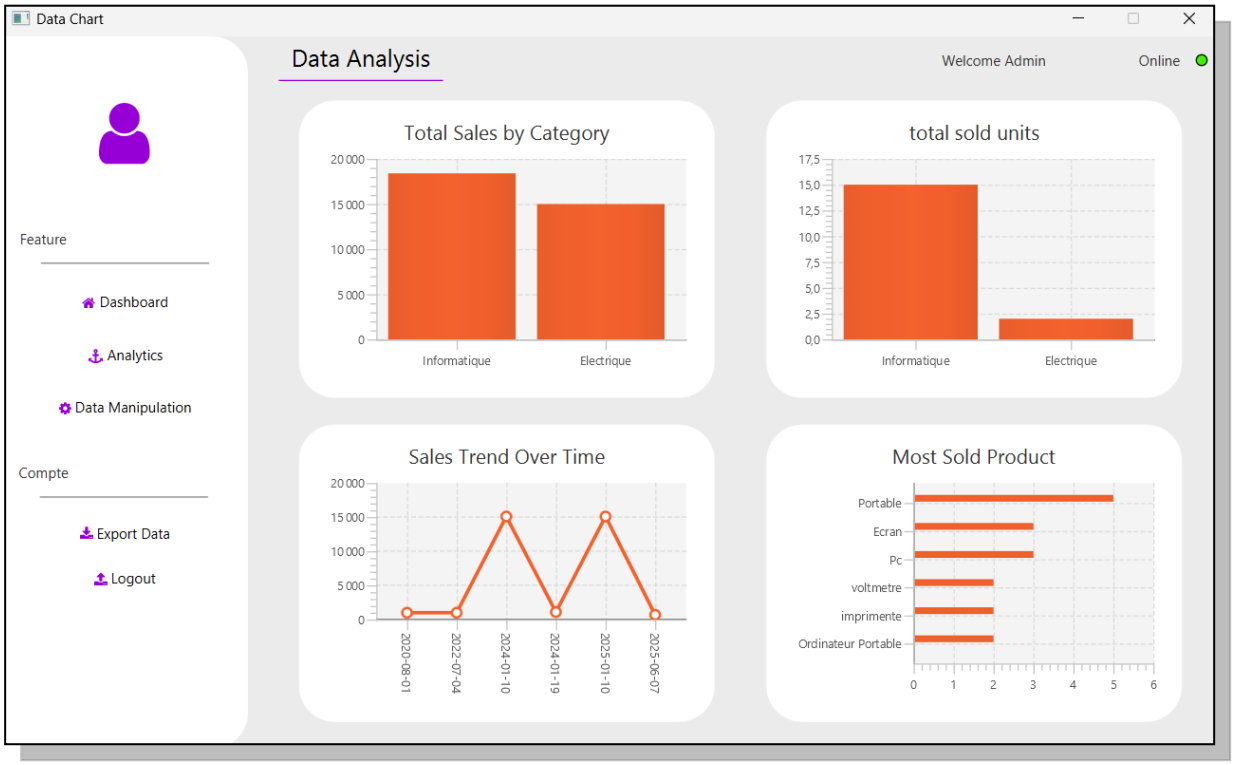
- Ce graph représente les produits ordonnés par ses prix unitaires en peut avoir le plus et mois cher.
- Pour afficher les informations de barre il suffit de pointer sur la barre



- Ce diagramme circulaire représente les pourcentages des produits et combien d'unités vendus
- Pour afficher les informations de barre il suffit de pointer sur la barre

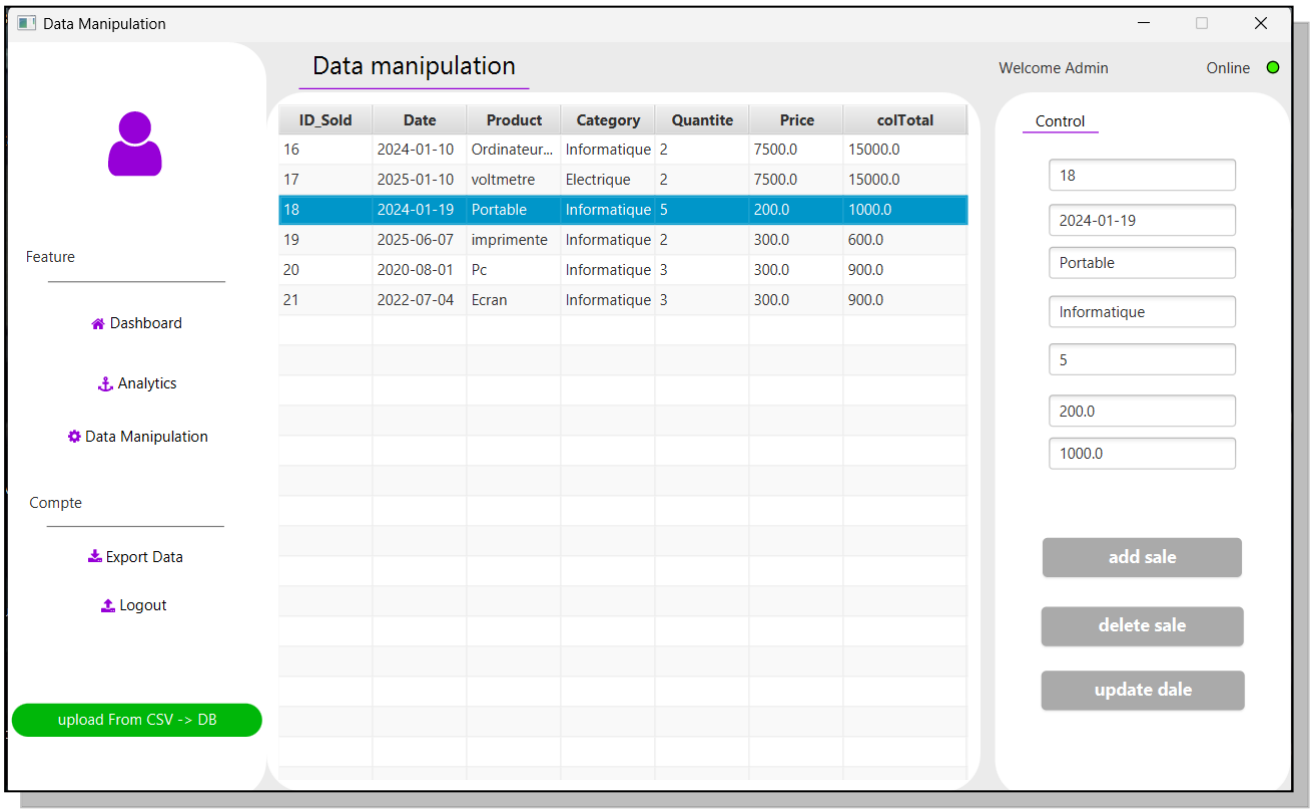
iii. Page des Analyses :

Il faut que les données soient déjà importées depuis Tableau de bord pour afficher ces graphiques dans la page (Data Analysis**).



- Le Premier Graph (**Total Sales By Category**) représente ventes totales par catégorie de produit.
- Le Deuxième graph (**Total Sold units**) représente combien d'unités a été vendus par chaque catégorie.
- Le Troisième graph (Line chart **Sales Trend Over Time**) représente les ventes par mois.
- Le Dernière graph (**Most Sold Product**) représente combien d'unité a été vendus pour chaque produit par ordre décroissant.

iv. Page de Manipulation des données :



- **Ajouter une vente** : pour ajouter une vente il faut remplir les champs de la vente et appuyer sur **add sale.**

- **Supprimer une vente :** il faut choisir quel produit tu veut supprimer puis appuyer sur **delete sale.**

- **Modifie :** il faut choisir puis changer le champ que tu veux le modifie ensuit appuyer sur **update sale**

- **Upload From CSV ->DB :** ce bouton c'est pour transférer les données d'un fichier csv dans la base de données.

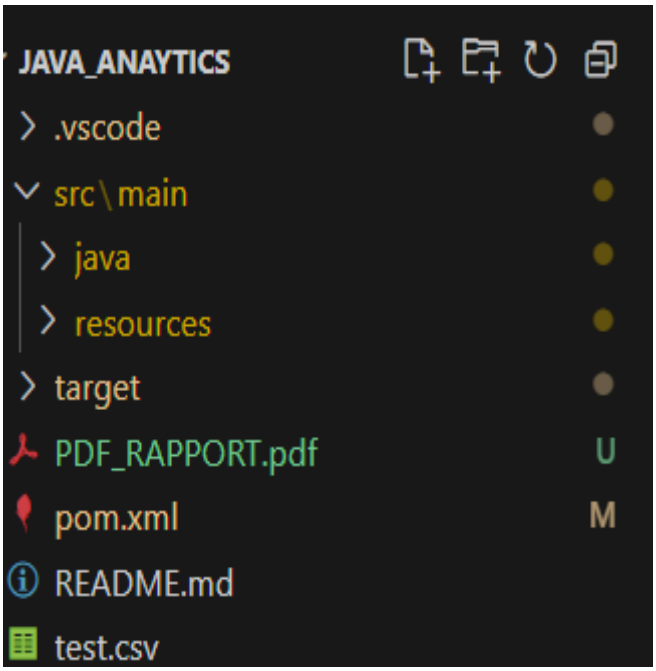
v. Export les données sous forme PDF :

Dans n'importe qu'elle page en peut appuyer sur bouton **Export Data** Enregistrer les Diagrammes de l'interface **Data Analysis** et **Dashboard** et le tableau des données dans un fichier PDF.

Le PDF s'enregistre dans le dossier de projet (java_analytics) sous le nom **PDF_RAPPORT.pdf**

**** Il faut importer les données dans Tableau de bord avant d'appuyer sur le bouton d'export**

**** Il ne faut pas appuyer sur export si le fichier est déjà ouvert**



La Premier partie fu fichier PDF_RAPPORT.pdf contient des informations comme la date de la génération du rapport. Aussi un tableau des données.

Rapport des ventes

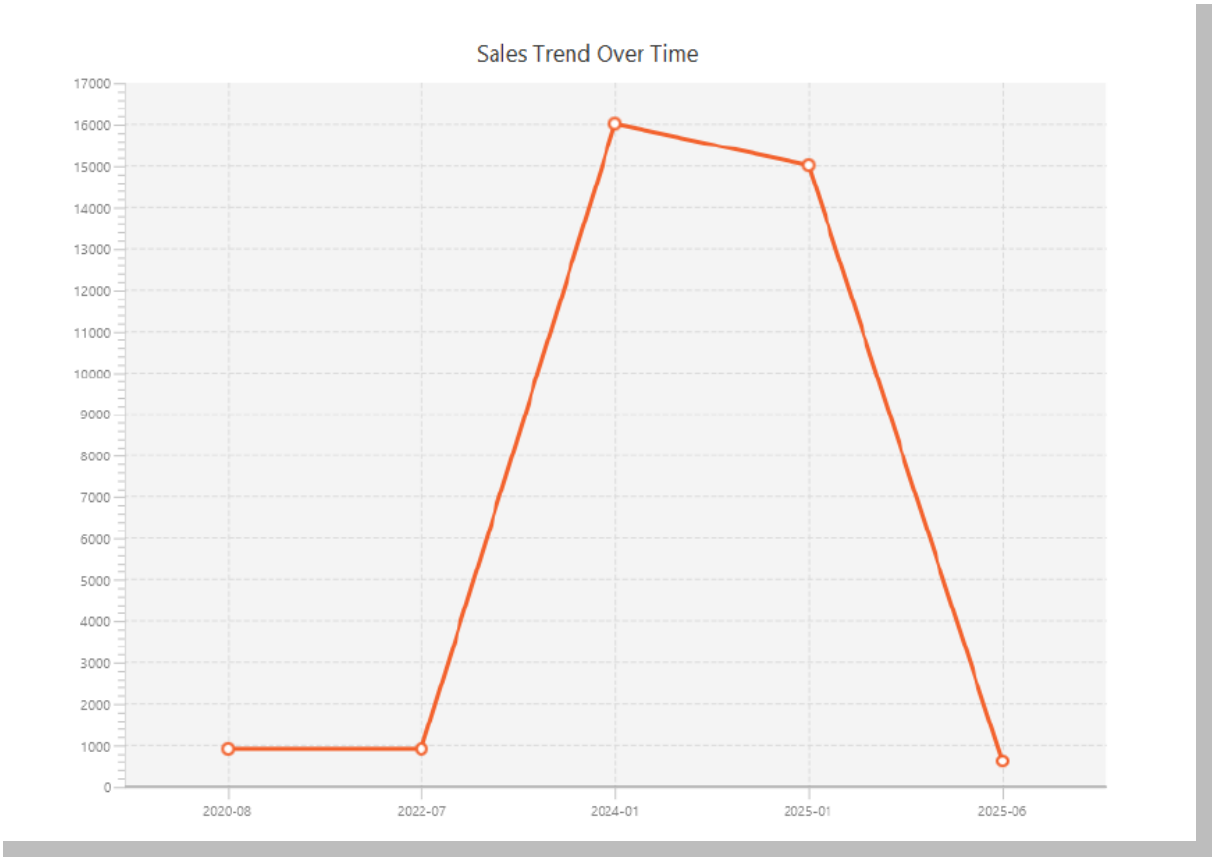
Date du rapport : 2025-05-14
Généré par : Analyse des ventes - Amine Koula
Résumé : Rapport graphique sur les ventes

Tableau des vesntes :

ID	Date	Produit	Categorie	Quantité	Prix unitaire	Prix total
16	2024-01-10	Ordinateur Portable	Informatique	2	7500.0	15000.0
17	2025-01-10	voltmetre	Electrique	2	7500.0	15000.0
18	2024-01-19	Portable	Informatique	5	200.0	1000.0
19	2025-06-07	imprimente	Informatique	2	300.0	600.0
20	2020-08-01	Pc	Informatique	3	300.0	900.0
21	2022-07-04	Ecran	Informatique	3	300.0	900.0

Figure : Depuis fichier PDF_RAPPORT.pdf

Exemple d'un diagramme depuis l'application dans le fichier



II. Implémentation Technique:

i. Importation des données CSV :

L'application doit lire un fichier CSV et convertir chaque ligne en objet Sale.

La classe CSVReader.java utilise la BufferedReader pour parser le fichier :

```java
public static List<Sale> readCSV(String filePath){
    List<Sale> sales = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
        String line;
        boolean isFirstLine = true;

        while ((line = br.readLine()) != null) {
            if (isFirstLine) {
                isFirstLine = false;
                continue;
            }

            String[] values = line.split(regex:"[,;]");

            if (values.length < 7) {
                System.err.println("Skipping invalid line: " + line);
                continue;
            }

            try {
                Sale sale = new Sale(
                    Integer.parseInt(values[0]),
                    values[1],
                    values[2],
                    values[3],
                    Integer.parseInt(values[4]),
                    Double.parseDouble(values[5]),
                    Double.parseDouble(values[6])
                );
                sales.add(sale);
            } catch (Exception e) {
                System.err.println("Error parsing row: " + line);
                e.printStackTrace();
            }
        }
    } catch (Exception e) {
        System.err.println("Error reading file: " + filePath);
        e.printStackTrace();
    }

    return sales;
}
```

Figure : depuis class CSVReader dans utils

Par split en va différencier les champs et faire une vérification que les champs ne sont pas plus que 7.

Puis on ajoute tous les données en ArrayList : Sale pour les traiter après par une boucle while.

ii. Méthodes clés pour l'analyse :

La classe SaleService.java centralise la logique métière pour l'analyse des ventes. Voici les algorithmes implémentés :

```java
public static Map<String,Double> getTotalPriceByCategory(){
    Map<String,Double> totalPriceByCategory = new HashMap<>();
    for(Sale sale : Sales){
        String category = sale.getProductCategory();
        double totalPrice = sale.getTotalPrice();
        if(totalPriceByCategory.containsKey(category)){
            totalPriceByCategory.put(category, totalPriceByCategory.get(category) + totalPrice);
        }else{
            totalPriceByCategory.put(category, totalPrice);
        }
    }
    return totalPriceByCategory;
}

// number of sold products by category
public static Map<String,Integer> getUnitsSoldByCategory(){
    Map<String,Integer> unitSold = new HashMap<>();
    for(Sale sale : Sales){
        String category = sale.getProductCategory();
        int quantity = sale.getQuantity();
        if(unitSold.containsKey(category)){
            unitSold.put(category, unitSold.get(category)+quantity);
        }else{
            unitSold.put(category,quantity);
        }
    }
    return unitSold;
}
```

Figure : depuis class SalService dans services

- la première méthode donne une Map de clé et valeur, clé associé a la catégorie du produit et la valeur c'est la prix total vendus de cette catégorie, calculé par la somme de prix total pour chaque vente.

- La deuxième méthode correspond a le nombre des produits vendus dans chaque catégorie.

iii. Génération des graphiques :

En utilisant la classe SaleService pour le calcul en peut générer des diagrammes par JavaFX dans la classe ChartGenerator et les utiliser dans les contrôleurs pour les afficher.

```java
// create barchart for sold units by category
public static BarChart<String,Number> createUnitSoldByCategoryChart(ObservableList<Sale> sales){
    final CategoryAxis xAxis = new CategoryAxis();
    final NumberAxis yAxis = new NumberAxis();

    Map<String,Integer> unitSold = new HashMap<>();
    unitSold = SaleService.getUnitsSoldByCategory();

    BarChart<String, Number> barChart = new BarChart<>(xAxis, yAxis);
    barChart.setTitle(value:"total sold units");
    barChart.setLegendVisible(value:false);

    XYChart.Series<String, Number> series = new XYChart.Series<>();
    series.setName(value:"Units Sold");

    for (Map.Entry<String, Integer> entry : unitSold.entrySet()) {
        series.getData().add(new XYChart.Data<>(entry.getKey(), entry.getValue()));
    }
    barChart.getData().add(series);
    return barChart;
}
```

Figure : depuis class ChartGenerator dans utils

- La méthode affichée dans le code prend comme argument une liste de Sale puis utilise une méthode de class SaleService (**getUnitSoldByCategory()**) qui retourner une Map clé valeur et créé un Diagramme de barres qui implémente les résultats de la fonction importé.

iv. Export d'un rapport pdf :

L'exportation du rapport PDF a été réalisée en combinant les données tabulaires et les graphiques générés dynamiquement à partir des ventes. Une classe utilitaire PDFExporter a été développée en utilisant la bibliothèque iText (Lowagie) pour créer un document PDF au format paysage. Le rapport commence par un en-tête contenant un titre, des informations générales (date, auteur, résumé), suivi d'un tableau récapitulatif des ventes. Pour intégrer les graphiques JavaFX dans le PDF, chaque graphique (de type Node) est converti en image à l'aide d'un snapshot, en veillant à ce qu'il soit attaché à une scène JavaFX pour garantir le rendu visuel. Les images sont ensuite insérées dans le document PDF. Ce processus est encapsulé dans un thread secondaire, tandis que la capture d'image est effectuée sur le thread JavaFX via Platform.runLater, assurant ainsi la stabilité et la cohérence du rendu. Cette méthode permet de générer automatiquement un rapport PDF visuel et complet à partir des données chargées dans l'interface.

```
        // Ajouter chaque graphique
        for (Map.Entry<String, Node> entry : chartMap.entrySet()) {
            Node chartNode = entry.getValue();

            // Snapshot du graphique
            BufferedImage chartImage = ChartImg.captureNodeAsImage(chartNode);
            if (chartImage != null) {
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                ImageIO.write(chartImage, formatName:"png", baos);
                Image image = Image.getInstance(baos.toByteArray());
                image.scaleToFit(fitWidth:700, fitHeight:400);
                image.setAlignment(Image.ALIGN_CENTER);
                document.add(image);
            } else {
               System.out.println(x:"Chart img is null !!!!");
            }

        }

        document.close();
        System.out.println("PDF généré avec succès : " + outputPath);

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Figure : depuis class PDFExpoter dans utils

- L'ajoute des graphiques qui sont capturé a l'aide d'un snapshot dans une autre classe (**ChartImg**) qui est été importée

```
@FXML
private void exportPDF(){
    new Thread(()->{
        PDFExporter.exportChartsToPDF(outputPath:"PDF_RAPPORT.pdf",charts,salestable.getItems());
    }).start();
}
```

Figure : depuis class Contrôleur

- L'utilisation du **exportChartsToPDF()** en donnant aux argumments le nom du pdf et une collection des diagrammes avec leurs titres et les données pour créé un tableau dans le pdf.

Chapitre 5 : Difficultés et Solutions :

I. Problèmes Rencontrés

i. Problème 1 : Incompatibilité des versions Java et Maven :

L'utilisation initiale de Java 23 a causé des erreurs de compilation imprévues et des incompatibilités avec certaines dépendances Maven. Le projet refusait de se builder correctement, bloquant tout développement ultérieur.

ii. Problème 2 : Échec d'importation de JFreeChart:

La bibliothèque JFreeChart, bien que puissante, a posé des problèmes persistants d'importation en raison de conflits avec d'autres dépendances et de problèmes de résolution dans le fichier POM. Les erreurs de classe introuvable rendaient son utilisation impossible.

iii. Problème 3 : Gestion des changements de scènes avec données

Le passage entre les différentes scènes JavaFX (ex: écran de login → tableau de bord) tout en conservant les données utilisateur ou les résultats d'analyse posait problème. Les données se perdaient lors des transitions, obligeant à des rechargements coûteux.

II. Solutions Apportés :

i. Solution de Problème 1 :

Après analyse, j'ai complètement désinstallé Java 23 et suis revenu à Java 17 (LTS), une version plus stable et mieux supportée par l'écosystème Maven. J'ai recréé le projet depuis zéro avec cette configuration, ce qui a résolu les problèmes de build et permis une gestion fluide des dépendances.

ii. Solution de Problème 2 :

Plutôt que de perdre du temps à résoudre ces conflits, j'ai opté pour une solution alternative en utilisant les composants graphiques natifs de JavaFX (comme LineChart, BarChart). Bien que moins riche en fonctionnalités que JFreeChart, cette approche a offert une intégration transparente et des performances satisfaisantes pour nos besoins de visualisation.

iii. Solution de Problème 3 :

Afin de faciliter le passage de données entre différentes scènes dans l'application JavaFX, nous avons mis en place une approche générique et réutilisable en utilisant une interface nommée DataReceiver. Lors du chargement d'une nouvelle scène via FXMLLoader, le contrôleur associé est récupéré dynamiquement à l'aide de loader.getController(). Ensuite, grâce au mot-clé instanceof, on vérifie si ce contrôleur implémente l'interface DataReceiver. Si c'est le cas, la méthode setData() est appelée pour transmettre les données souhaitées. Cette solution permet

une meilleure flexibilité, car elle évite de dépendre d'un type de contrôleur spécifique, et garantit que seuls les contrôleurs conçus pour recevoir des données les reçoivent réellement. Cela respecte les principes SOLID, notamment le principe de l'inversion de dépendance et la séparation des responsabilités, tout en assurant une bonne maintenabilité du code.

```
// Pass data to the controller
Object controller = loader.getController();
if (controller instanceof DataReceiver) {
    ((DataReceiver) controller).setData(data);
```

```
public interface DataReceiver {
    void setData(ObservableList<Sale> data);
}
```

Depuis class SceneManager dans utils