

# **Gest Stock**

« Création d'une plateforme de gestion de stock avec JavaSwing. »

Par Vasco, Guillaume et Baptiste – Au cours du BTS – Projet de Formation

# Table des matières

1. Introduction .....	4
2. Objectifs du Projet .....	4
2.1 Contexte de Développement .....	4
2.2 Objectifs Principaux .....	4
3. Architecture de l'Application.....	4
3.1. Style Architectural MVC .....	4
3.2. Modules et Composants Clés.....	5
3.3. Fichiers Essentiels et Leurs Rôles .....	5
3.4. Choix Technologiques.....	6
4. Gestion et Structure des Données .....	7
4.1. Mes Contributions aux Entités : AppUser, Role et Product.....	7
4.2. Mécanisme de Stockage et Sécurité .....	9
4.3. Flux de Données Principal .....	9
5. Fonctionnalités Principales et Mon Rôle dans leur Implémentation .....	10
5.1. Mon Travail sur la Gestion des Utilisateurs et des Rôles (AppUser, Role).....	10
5.2. Ma Contribution à la Gestion des Produits (Product) .....	11
5.3. Implémentation des Tests Unitaires.....	13
5.4. Gestion des Catégories et Fournisseurs (Collaboration) .....	14
5.5. Contrôle d'Accès Basé sur les Rôles .....	14
6. Interface Utilisateur (UI) .....	14
6.1. Description des Écrans/Vues Principales .....	14
6.2. Définition de l'UI et Composants Clés.....	14
6.3. Flux d'Interaction Utilisateur Principaux.....	15
7. Points Particuliers et Choix de Conception .....	18
8. Processus de Build et de Déploiement .....	19
8.1. Instructions de Build .....	19
8.2. Instructions d'Exécution/Déploiement .....	19
8.3. Dépendances.....	19
9. Stratégies de Gestion des Erreurs et de Journalisation.....	19
9.1 Gestion des Erreurs .....	19

9.2 Journalisation (Logging) .....	19
10. Mon Rôle Spécifique dans le Projet (Synthèse) .....	20
11. Bilan Personnel et Apprentissages .....	20
12. Perspectives d'Évolution et Améliorations Futures .....	20
13. Conclusion .....	21

## 1. Introduction

J'ai participé activement au développement de GestStock, une application de bureau conçue en Java avec l'interface graphique Swing. Notre objectif était de créer un système complet pour la gestion de stock, incluant la gestion des utilisateurs avec une authentification basée sur les rôles, la gestion des produits, des catégories, des fournisseurs, et le suivi des ventes. La persistance des données est assurée par une base de données SQL Server.

## 2. Objectifs du Projet

### 2.1 Contexte de Développement

Nous avons entrepris ce projet dans un cadre académique, avec l'ambition de développer une application de gestion de stock robuste et complète. L'accent a été mis sur l'application d'une architecture logicielle structurée et des bonnes pratiques de développement en Java.

### 2.2 Objectifs Principaux

Pour ce projet, nos objectifs communs étaient de :

- Maîtriser et appliquer l'architecture **MVC** en Java, en assurant une séparation claire des responsabilités entre les différentes couches ;
- Implémenter un **système d'authentification sécurisé**, incluant le hachage des mots de passe et une gestion fine des rôles utilisateurs. J'ai particulièrement contribué à cet aspect ;
- Développer une **interface graphique intuitive et moderne** en utilisant Java Swing, en y intégrant le thème FlatLaf ;
- Gérer la **persistance des données** de manière efficace avec JDBC et une base de données SQL Server ;
- Mettre en œuvre les opérations **CRUD complètes** pour toutes les entités métier de l'application (utilisateurs, produits, catégories, etc.) ;
- Élaborer des **tests unitaires** pour valider les fonctionnalités critiques, une tâche sur laquelle j'ai également beaucoup travaillé.

## 3. Architecture de l'Application

### 3.1. Style Architectural MVC

Nous avons collectivement décidé d'adopter une architecture **MVC (Modèle-Vue-Contrôleur)** pour GestStock. Ce choix nous a permis de bien structurer notre code et de séparer clairement les préoccupations :

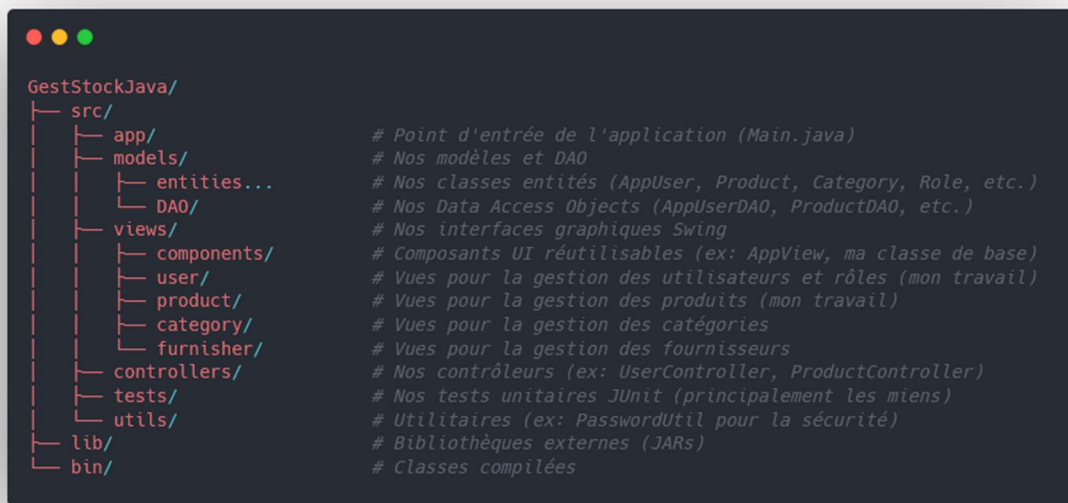
- **Modèle (Model)** : Cette couche, à laquelle j'ai beaucoup contribué pour les entités AppUser, Role et Product, comprend nos classes métier (entités) et les Data Access Objects (DAO) qui gèrent la communication avec la base de données SQL Server ;
- **Vue (View)** : L'ensemble de nos interfaces utilisateur, que nous avons développées en Java Swing. J'ai personnellement conçu et implémenté plusieurs vues, notamment celles liées à la gestion des utilisateurs, des rôles et des produits ;

- **Contrôleur (Controller)** : Cette couche assure la liaison entre le modèle et la vue, contenant la logique métier et coordonnant les actions. J'ai développé les contrôleurs pour les fonctionnalités dont j'avais la charge.

Pour l'accès aux données, nous avons utilisé le pattern **DAO** afin d'abstraire la logique de persistance.

### 3.2. Modules et Composants Clés

Voici comment nous avons structuré notre projet :



(Titre indicatif seulement)

### 3.3. Fichiers Essentiels et Leurs Rôles

- **Point d'entrée principal :**
  - src/app/Main.java : Configure l'application (notamment le Look & Feel FlatLaf) et lance l'interface de connexion ;
- **Configuration et connexion DB :**
  - src/models/DAO/DBConnection.java : Notre gestionnaire de connexion à la base de données SQL Server, utilisant les informations d'un fichier .env ;
  - .env (à la racine du projet) : Stocke les variables d'environnement sensibles pour la connexion à la base de données.
- **Mes contributions aux entités métier principales :**
  - src/models/entities/AppUser.java : L'entité que j'ai définie pour les utilisateurs, intégrant la notion de rôle ;
  - src/models/entities/Role.java : L'entité pour les rôles, permettant un contrôle d'accès précis ;

- `src/models/entities/Product.java` : L'entité pour les produits, avec ses relations vers les catégories et fournisseurs.
- **Interfaces utilisateur clés (dont celles que j'ai développées) :**
  - `src/views/LoginView.java` : L'interface d'authentification ;
  - `src/views/MainMenuView.java` : Le menu principal, dont l'affichage des options est conditionné par le rôle de l'utilisateur connecté ;
  - `src/views/user/CreateUserView.java` (et des vues similaires pour la gestion des utilisateurs que j'ai imaginées comme `UserListView.java`, `ManageUserView.java`) : Mes interfaces pour la création et la gestion des utilisateurs ;
  - `src/views/product/ProductView.java`, `ProductListView.java`, `ManageProductView.java` : Les interfaces que j'ai développées pour le CRUD des produits ;
  - `src/views/components/AppView.java` : Une classe de base que j'ai créée pour standardiser l'apparence et le comportement de toutes nos vues Swing.
- **Gestion des données (mes contributions principales aux DAO) :**
  - `src/models/DAO/AppUserDAO.java` : Le DAO que j'ai écrit pour toutes les opérations sur les utilisateurs et leurs rôles ;
  - `src/models/DAO/ProductDAO.java` : Mon DAO pour la persistance des produits ;
  - `src/models/DAO/RoleDAO.java` : Mon DAO pour la gestion des rôles.

### 3.4. Choix Technologiques

Nous avons utilisé les technologies suivantes :

- **Langage de Programmation Principal** : Java ;
- **Frameworks et Bibliothèques Majeures** :
  - **Java Swing** : Pour l'ensemble de notre interface graphique ;
  - **JDBC** : Pour la connectivité à notre base de données SQL Server ;
  - **FlatLaf** : Une bibliothèque que nous avons intégrée pour donner un thème moderne et sombre à notre application Swing ;
  - **dotenv-java** : Pour gérer nos variables d'environnement de manière sécurisée ;
  - **Microsoft SQL Server JDBC Driver** : Le connecteur spécifique pour notre base de données ;
  - **jBCrypt** : La bibliothèque que j'ai utilisée pour le hachage sécurisé des mots de passe ;
  - **JUnit Jupiter** : Le framework que j'ai utilisé pour écrire nos tests unitaires.
- **Base de Données** : Microsoft SQL Server ;

- **Environnement de Développement** : Nous avons principalement utilisé VS Code. La compilation se fait manuellement avec javac pour ce projet d'étude ;
- **Gestion des dépendances** : Manuelle, en plaçant les fichiers JAR nécessaires dans le dossier lib/.

## 4. Gestion et Structure des Données

### 4.1. Mes Contributions aux Entités : AppUser, Role et Product

J'ai été particulièrement impliqué dans la définition et l'implémentation des entités AppUser, Role et Product.

- **AppUser.java** :
  - J'ai conçu cette classe pour représenter nos utilisateurs, avec des attributs comme user\_id, user\_name, user\_password (qui est stocké haché), et role\_id faisant référence à l'entité Role ;

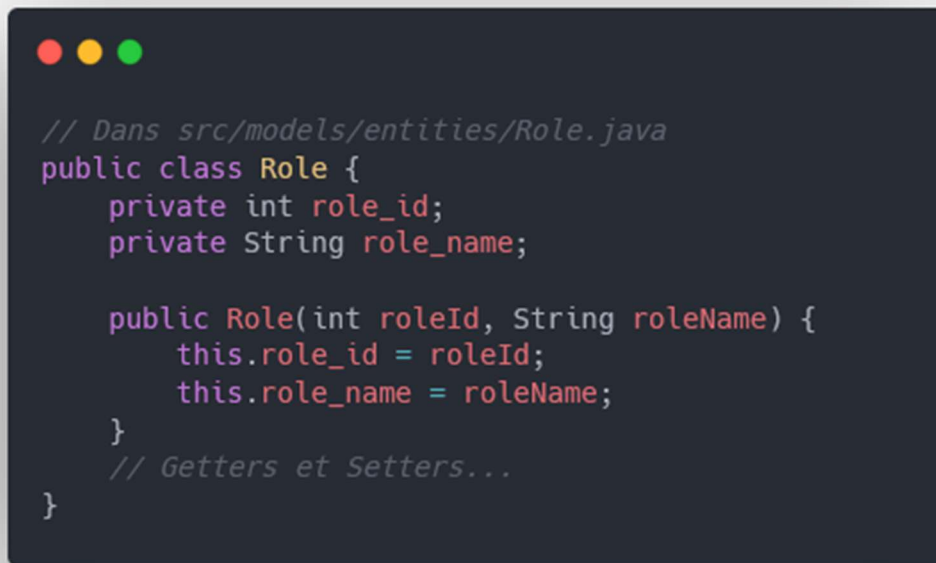
```
// Dans src/models/entities/AppUser.java
public class AppUser {
    private int user_id;
    private String user_name;
    private String user_password; // Stocke le mot de passe haché
    private int role_id;
    private Role role; // Objet Role pour un accès facile au nom du rôle

    // Constructeurs que j'ai définis
    public AppUser(int userId, String userName, String hashedPassword, int roleId) {
        this.user_id = userId;
        this.user_name = userName;
        this.user_password = hashedPassword;
        this.role_id = roleId;
    }

    // Getters et Setters que j'ai implémentés...
    // J'ai aussi ajouté une méthode pour charger l'objet Role associé
    public Role getRole() {
        if (this.role == null && this.role_id > 0) {
            this.role = new RoleDAO().getRoleByID(this.role_id);
        }
        return this.role;
    }
}
```

(Titre indicatif seulement)

- **Role.java** :
  - Une entité simple que j'ai créée pour définir les différents rôles dans l'application (ex: "Administrateur", "Manager", "Utilisateur"). Elle contient role\_id et role\_name.



```
// Dans src/models/entities/Role.java
public class Role {
    private int role_id;
    private String role_name;

    public Role(int roleId, String roleName) {
        this.role_id = roleId;
        this.role_name = roleName;
    }
    // Getters et Setters...
}
```

(Titre indicatif seulement)

- **Product.java :**
  - J'ai défini cette entité pour représenter nos produits en stock. Elle inclut product\_id, product\_name, product\_quantity, product\_unit\_price, et des références vers les entités Category et Furnisher. J'ai également prévu une liste de Sale pour tracer les ventes associées à chaque produit.



```

// Dans src/models/entities/Product.java
public class Product {
    private int product_id;
    private String product_name;
    private int product_quantity;
    private double product_unit_price;
    private Category category; // Relation vers Category
    private Furnisher furnisher; // Relation vers Furnisher
    private List<Sale> sales; // Liste des ventes pour ce produit

    // Constructeurs que j'ai mis en place...
    public Product(int productId, String productName, int quantity, double unitPrice,
Category category, Furnisher furnisher) {
        this.product_id = productId;
        this.product_name = productName;
        this.product_quantity = quantity;
        this.product_unit_price = unitPrice;
        this.category = category;
        this.furnisher = furnisher;
        this.sales = new ArrayList<>(); // Initialisation de la liste des ventes
    }
    // Getters, Setters...
    // J'ai ajouté une méthode pour lier une vente à un produit
    public void addSale(Sale sale) {
        if (this.sales == null) {
            this.sales = new ArrayList<>();
        }
        this.sales.add(sale);
        // Idéalement, il faudrait aussi mettre à jour la quantité en stock ici
        // this.product_quantity -= sale.getSaleQuantity();
    }
    // J'ai aussi créé une méthode productDetails() pour un affichage formaté
    public String productDetails() {
        return "ID: " + product_id +
            "\nNom: " + product_name +
            "\nQuantité: " + product_quantity +
            "\nPrix Unitaire: " + String.format("%.2f", product_unit_price) + " €" +
            (category != null ? "\nCatégorie: " + category.getCategoryName() : "") +
            (furnisher != null ? "\nFournisseur: " + furnisher.getFurnisherName() : "");
    }
}

```

(Titre indicatif seulement)

## 4.2. Mécanisme de Stockage et Sécurité

Nous stockons toutes nos données persistantes dans une base de données Microsoft SQL Server. L'accès se fait via JDBC et j'ai développé les classes DAO pour abstraire ces interactions.

Pour la sécurité, comme mentionné, j'ai implémenté le hachage des mots de passe avec jBCrypt avant leur stockage. J'ai également veillé à ce que les informations sensibles de connexion à la base de données soient gérées via un fichier .env et la bibliothèque dotenv-java, plutôt qu'en dur dans le code.

## 4.3. Flux de Données Principal

Le flux de données dans notre application suit le pattern MVC :

1. L'utilisateur interagit avec une **Vue** (ex: ProductView que j'ai créée) ;
2. L'action est transmise à un **Contrôleur** (ex: ProductController que j'ai développé) ;
3. Le Contrôleur valide les données et appelle les méthodes appropriées d'un **DAO** (ex: ProductDAO sur lequel j'ai travaillé) ;
4. Le DAO exécute des requêtes SQL sur la base de données **SQL Server** via JDBC ;
5. Les données (ou un statut de succès/échec) sont retournées au Contrôleur, puis à la Vue pour mettre à jour l'interface utilisateur.

## 5. Fonctionnalités Principales et Mon Rôle dans leur Implémentation

J'ai été le principal développeur pour les fonctionnalités de gestion des utilisateurs, des rôles, et des produits, ainsi que pour la mise en place des tests unitaires.

### 5.1. Mon Travail sur la Gestion des Utilisateurs et des Rôles (AppUser, Role)

- **Description** : J'ai conçu et implémenté le système complet d'authentification et de gestion des utilisateurs. Cela inclut la création de comptes, la connexion sécurisée, et l'attribution de rôles qui déterminent les permissions dans l'application ;
- **Fichiers/Classes Impliqués (principalement mon travail)** :
  - Entités : src/models/entities/AppUser.java, src/models/entities/Role.java ;
  - DAOs : src/models/DAO/AppUserDAO.java, src/models/DAO/RoleDAO.java ;
  - Vues : src/views/LoginView.java, src/views/user/UserView.java, et j'ai conçu les vues pour lister et gérer les utilisateurs (UserListView.java, ManageUserView.java) ;
  - Utilitaires : src/utills/PasswordUtil.java (pour le hachage BCrypt) ;
  - Contrôleurs : src/controllers/UserController.java.
- **Logique Clé de Mon Implémentation** :
  - **Authentification** : Lors de la connexion, le mot de passe fourni est haché et comparé à celui stocké en base. J'ai utilisé PasswordUtil.verifyPassword() pour cela ;
  - **Création d'Utilisateur** : J'ai mis en place la validation pour s'assurer de l'unicité du nom d'utilisateur. Le mot de passe est haché avant d'être sauvegardé. L'administrateur peut assigner un rôle lors de la création ;
  - **Gestion des Rôles** : J'ai créé une table Role simple. Le AppUserDAO permet de récupérer un utilisateur avec son rôle, et le rôle est ensuite utilisé pour conditionner l'accès à certaines fonctionnalités dans les vues (par exemple, seul un "Administrateur" peut créer de nouveaux utilisateurs).
- **Extrait de Code (Création d'utilisateur dans UserView.java - mon implémentation)** :

```

// Ma méthode pour l'action de création d'utilisateur
private void createUserAction() {
    String name = nameTextField.getText().trim();
    String password = new String(passwordTextField.getPassword()).trim();
    String roleName = (String) roleSelector.getSelectedItem(); // JComboBox pour sélectionner le rôle

    if (name.isEmpty() || password.isEmpty() || roleName == null) {
        JOptionPane.showMessageDialog(this, "Veuillez remplir tous les champs et sélectionner un rôle !",
        "Erreur de Saisie", JOptionPane.ERROR_MESSAGE);
        return;
    }

    AppUserDAO userDAO = new AppUserDAO();
    if (userDAO.checkUser(name)) { // Vérification que j'ai ajoutée pour l'unicité
        JOptionPane.showMessageDialog(this, "Ce nom d'utilisateur est déjà utilisé.", "Erreur",
        JOptionPane.ERROR_MESSAGE);
        return;
    }

    RoleDAO roleDAO = new RoleDAO();
    Role role = roleDAO.getRoleByName(roleName);
    if (role == null) {
        JOptionPane.showMessageDialog(this, "Rôle sélectionné invalide.", "Erreur",
        JOptionPane.ERROR_MESSAGE);
        return;
    }

    String hashedPassword = PasswordUtil.hashPassword(password); // J'utilise ma classe utilitaire
    AppUser newUser = userDAO.createUser(name, hashedPassword, role.getRoleId());

    if (newUser != null) {
        JOptionPane.showMessageDialog(this, "Utilisateur '" + newUser.getUserName() + "' créé avec succès avec
le rôle '" + role.getRoleName() + "' !", "Succès", JOptionPane.INFORMATION_MESSAGE);
        dispose(); // Fermer la fenêtre de création
        // Potentiellement rafraichir une liste d'utilisateurs si elle existe
    } else {
        JOptionPane.showMessageDialog(this, "Erreur lors de la création de l'utilisateur.", "Erreur",
        JOptionPane.ERROR_MESSAGE);
    }
}

```

(Titre indicatif seulement)

## 5.2. Ma Contribution à la Gestion des Produits (Product)

- **Description** : J'ai développé le module de gestion des produits, permettant les opérations CRUD complètes, la gestion des stocks, des prix, et l'association avec des catégories et des fournisseurs ;
- **Fichiers/Classes Impliqués (principalement mon travail)** :
  - Entité : src/models/entities/Product.java ;
  - DAO : src/models/DAO/ProductDAO.java ;
  - Vues : src/views/product/ProductView.java (pour la création/modification), src/views/product/ProductListView.java (pour lister et rechercher), src/views/product/ManageProductView.java (menu de gestion) ;
  - Contrôleur : src/controllers/ProductController.java.
- **Logique Clé de Mon Implémentation** :

- **Opérations CRUD** : J'ai implémenté les méthodes dans ProductDAO pour ajouter, récupérer (tous, par ID, par nom), mettre à jour et supprimer des produits ;
- **Interface de Saisie/Modification (ProductView)** : J'ai conçu cette vue pour permettre une saisie claire des informations du produit, avec des JComboBox pour sélectionner la catégorie et le fournisseur (pré-remplis depuis la base de données) ;
- **Validation des Données** : J'ai inclus des validations pour s'assurer que les quantités et les prix sont des nombres valides, et que les champs obligatoires sont remplis ;
- **Affichage en Liste (ProductListView)** : Une JTable pour afficher les produits, avec des fonctionnalités de recherche.
- **Extrait de Code (Contrôleur ProductController - mon implémentation) :**

```
// Mon contrôleur pour la gestion des produits
public class ProductController {
    private ProductView view;
    private ProductDAO productDAO;
    private AppUser currentUser; // Pour vérifier les permissions si nécessaire

    public ProductController(ProductView view, ProductDAO productDAO, AppUser currentUser) {
        this.view = view;
        this.productDAO = productDAO;
        this.currentUser = currentUser; // Important pour le contrôle d'accès

        this.view.setSaveButtonListener(e -> saveProduct());
        // Listeners pour les boutons delete, clear, etc.
    }

    private void saveProduct() {
        // Récupération des données depuis ma ProductView
        String name = view.getProductNome();
        int quantity;
        double unitPrice;
        try {
            quantity = view.getProductQuantity();
            unitPrice = view.getProductUnitPrice();
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(view, "Quantité et Prix doivent être des nombres valides.", "Erreur de Format",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        // ... (récupération de Category et Furnisher via leurs DAOs respectifs) ...
        Category category = new CategoryDAO().getCategoryByName(view.getProductCategoryName());
        Furnisher furnisher = new FurnisherDAO().getFurnisherByName(view.getProductFurnisherName());

        if (name.trim().isEmpty() || category == null || furnisher == null) {
            JOptionPane.showMessageDialog(view, "Nom, Catégorie et Fournisseur sont obligatoires.", "Erreur de Saisie",
                JOptionPane.ERROR_MESSAGE);
            return;
        }

        Product productToSave;
        String successMessage;

        if (view.getProduct() != null) { // Mode édition
            productToSave = view.getProduct();
            productToSave.setProductNome(name);
            productToSave.setProductQuantity(quantity);
            productToSave.setProductUnitPrice(unitPrice);
            productToSave.setCategory(category);
            productToSave.setFurnisher(furnisher);
            productDAO.updateProduct(productToSave);
            successMessage = "Produit ID " + productToSave.getProductId() + " modifié avec succès !";
        } else { // Mode création
            productToSave = new Product(0, name, quantity, unitPrice, category, furnisher); // ID sera auto-généré
            productDAO.addProduct(productToSave);
            successMessage = "Produit " + name + " ajouté avec succès !";
        }

        JOptionPane.showMessageDialog(view, successMessage, "Succès", JOptionPane.INFORMATION_MESSAGE);
        view.dispose(); // Fermer la fenêtre de saisie
        // Rafraîchir la ProductListView
    }
}
```

(Titre indicatif seulement)

### 5.3. Implémentation des Tests Unitaires

- **Description** : J'ai pris l'initiative d'écrire des tests unitaires avec JUnit Jupiter pour valider certaines fonctionnalités critiques, en particulier les relations entre les entités ;
- **Fichiers/Classes Impliqués (mon travail)** :
  - src/tests/UnitTestProductSale.java ;
- **Logique Clé de Mes Tests** :
  - **Validation des Relations** : J'ai testé la relation bidirectionnelle entre Product et Sale pour m'assurer que l'ajout d'une vente à un produit est correctement reflété dans les deux entités ;
  - **Intégrité des Données** : J'ai vérifié que les objets sont correctement initialisés et que les méthodes métier (comme addSale sur Product) fonctionnent comme attendu.
- **Extrait de Code (Mes tests dans UnitTestProductSale.java)** :

```
// Mes tests pour la relation Produit-Vente
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
// ... (Imports de mes entités Product, Sale, Category, Furnisher) ...
import java.sql.Date; // Important d'utiliser java.sql.Date pour la compatibilité avec la BD

public class UnitTestProductSale {
    private Product product;
    private Sale sale;

    @BeforeEach
    void setUp() {
        // Initialisation pour chaque test
        // Je crée des instances mock ou des instances simples pour les tests
        Category mockCategory = new Category(1, "Test Category");
        Furnisher mockFurnisher = new Furnisher(1, "Test Furnisher", null, null, null, null);
        product = new Product(1, "Test Product", 100, 19.99, mockCategory, mockFurnisher);
        // J'utilise java.sql.Date pour la date de la vente
        sale = new Sale(1, 5, new Date(System.currentTimeMillis()), product);
        product.addSale(sale); // J'ajoute la vente au produit
    }

    @Test
    void testProduitVenteRelation() {
        // Je vérifie que le produit a bien une vente associée
        assertNotNull(product.getSales(), "La liste des ventes du produit ne devrait pas être nulle.");
        assertEquals(1, product.getSales().size(), "Le produit devrait avoir une vente associée.");
        assertEquals(sale, product.getSales().get(0), "La vente associée au produit devrait être la bonne.");

        // Je vérifie que la vente a le bon produit associé
        assertEquals(product, sale.getProduct(), "Le produit de la vente devrait correspondre au produit associé.");
    }

    @Test
    void testAjoutVenteProduct() {
        // Je teste l'ajout d'une nouvelle vente à un produit
        Product newProduct = new Product(2, "Another Test Product", 50, 29.99, null, null);
        Sale newSale = new Sale(2, 2, new Date(System.currentTimeMillis()), newProduct);

        newProduct.addSale(newSale); // J'utilise ma méthode addSale
        assertTrue(newProduct.getSales().contains(newSale), "La nouvelle vente devrait être associée au produit.");
        assertEquals(1, newProduct.getSales().size(), "Le produit devrait maintenant avoir une vente.");
    }
}
```

(Titre indicatif seulement)

## 5.4. Gestion des Catégories et Fournisseurs (Collaboration)

Bien que je me sois concentré sur les utilisateurs et les produits, j'ai collaboré avec Guillaume et Baptiste sur la mise en place des modules de gestion pour les catégories et les fournisseurs. Ces modules suivent une structure MVC et DAO similaire à celle des produits, permettant les opérations CRUD et assurent la cohérence des données (par exemple, on ne peut pas supprimer une catégorie si des produits y sont encore associés).

## 5.5. Contrôle d'Accès Basé sur les Rôles

J'ai veillé à ce que le système de rôles que j'ai conçu soit utilisé à travers l'application pour contrôler l'accès aux différentes fonctionnalités. Par exemple, dans `MainMenuView.java`, l'affichage des boutons de gestion (produits, catégories, utilisateurs) est conditionné par le rôle de l'utilisateur connecté. Seul un "Administrateur" peut accéder à la création d'utilisateurs, tandis qu'un "Manager" pourrait avoir des droits étendus sur les produits et les ventes, et un "Utilisateur" simple des droits plus limités.

# 6. Interface Utilisateur (UI)

## 6.1. Description des Écrans/Vues Principales

Notre application propose plusieurs écrans pour une gestion complète :

1. **Écran de Connexion (`LoginView.java`)** : Permet aux utilisateurs de s'authentifier ;
2. **Menu Principal (`MainMenuView.java`)** : Le hub central après connexion, avec des boutons pour naviguer vers les différentes sections de gestion. L'accès à ces boutons est dynamiquement ajusté en fonction du rôle de l'utilisateur ;
3. **Gestion des Utilisateurs** : J'ai conçu les vues pour créer de nouveaux utilisateurs (`UserView.java`) et j'ai prévu une structure pour lister et modifier les utilisateurs existants, en permettant notamment l'assignation de rôles ;
4. **Gestion des Produits** : Mes vues (`ProductView.java`, `ProductListView.java`, `ManageProductView.java`) permettent d'ajouter, modifier, supprimer et visualiser les produits, avec des champs pour la quantité, le prix, la catégorie et le fournisseur ;
5. **Gestion des Catégories et Fournisseurs** : Des interfaces similaires pour gérer ces entités, avec des listes et des formulaires de saisie.

## 6.2. Définition de l'UI et Composants Clés

Nous avons construit notre interface utilisateur avec **Java Swing**.

- **AppView.java** : J'ai créé cette classe de base pour toutes nos fenêtres. Elle standardise le titre, la taille, la fermeture par défaut et utilise un `GridBagLayout` pour un positionnement flexible des composants. Cela nous a permis d'avoir une apparence cohérente ;
- **FlatLaf** : Nous avons intégré cette bibliothèque pour appliquer un thème sombre moderne, rendant l'application plus agréable visuellement ;
- **Composants Swing Standards** : Nous utilisons `JTable` (avec `DefaultTableModel`) pour afficher les listes, `JTextField` (avec des filtres que j'ai parfois ajoutés pour les entrées numériques), `JPasswordField`, `JButton` (avec des curseurs personnalisés)



pour un meilleur feedback), JComboBox pour les sélections, et JOptionPane pour les messages et confirmations.

```
// Ma classe de base AppView.java
public class AppView extends JFrame {
    protected JPanel contentPanel;
    protected GridBagConstraints gbc;
    protected AppUser currentUser; // Pour passer l'utilisateur connecté aux vues

    public AppView(String title, int width, int height, boolean resizable, AppUser user) {
        super("GestStock - " + title);
        this.currentUser = user; // Stocker l'utilisateur pour la gestion des droits
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setSize(width, height);
        setResizable(resizable);
        setLocationRelativeTo(null); // Centrer la fenêtre

        contentPanel = new JPanel(new GridBagLayout());
        setContentPane(contentPanel);
        gbc = new GridBagConstraints();
        gbc.insets = new Insets(5, 10, 5, 10); // Marges pour les composants
        // Appliquer le thème FlatLaf si ce n'est pas déjà fait dans Main
        try {
            UIManager.setLookAndFeel(new FlatDarkLaf());
        } catch (UnsupportedLookAndFeelException e) {
            e.printStackTrace();
        }
    }
}
```

### 6.3. Flux d'Interaction Utilisateur Principaux

1. **Connexion** : L'utilisateur saisit ses identifiants, qui sont validés. En cas de succès, le menu principal s'affiche ;
2. **Gestion des Produits (mon module)** : Depuis le menu principal, l'utilisateur accède à la gestion des produits. Il peut voir la liste, rechercher, ajouter un nouveau produit via un formulaire dédié (où il sélectionne catégorie et fournisseur), modifier un produit existant ou le supprimer ;
3. **Gestion des Utilisateurs (mon module)** : Si l'utilisateur est Administrateur, il peut accéder à la gestion des utilisateurs pour en créer de nouveaux, leur assigner des rôles, ou modifier/supprimer des comptes existants.

GestStock - Connexion

## Connexion

Nom d'Utilisateur

Mot de Passe

Se Connecter

GestStock - Menu Principal

## Menu Principal

Gérer les Fournisseurs

Gérer les Catégories

Gérer les Produits

Gérer les Ventes

Créer un Utilisateur

Retour

(Bouton « créer un utilisateur » a maintenant été remplacé par « Gérer les Utilisateurs »)





(Logique similaire pour les autres vues)

GestStock - Ajout de Produit

## Ajout de Produit

Nom du Produit

Quantité :

Prix :

Catégorie :

Ordinateurs

Fournisseur :

Test fournisseur

Ajouter Produit Retour

## 7. Points Particuliers et Choix de Conception

Durant ce projet, nous avons fait plusieurs choix de conception importants :

- **Sécurité des Mots de Passe** : J'ai insisté sur l'utilisation de **BCrypt** pour le hachage des mots de passe, offrant une bonne protection contre les attaques par force brute ;
- **Pattern DAO** : Ce choix nous a permis de bien séparer la logique d'accès aux données du reste de l'application, ce qui facilite la maintenance et la testabilité. J'ai veillé à ce que mes DAO pour AppUser et Product suivent ce principe ;
- **Architecture MVC Stricte** : Nous avons essayé de respecter au mieux cette architecture pour garder notre code organisé ;
- **Classe AppView Commune** : Ma création de cette classe de base a aidé à standardiser nos interfaces Swing ;

- **Gestion des Relations entre Entités** : J'ai particulièrement travaillé sur la manière dont les Product sont liés aux Category et Furnisher, et comment les Sale sont associées aux Product ;
- **Variables d'Environnement** : L'utilisation de .env pour les informations sensibles de la base de données est une bonne pratique que nous avons adoptée.

## 8. Processus de Build et de Déploiement

### 8.1. Instructions de Build

Pour notre projet, la compilation se fait manuellement via la ligne de commande javac, en s'assurant d'inclure les bibliothèques du dossier lib/ dans le classpath.

### 8.2. Instructions d'Exécution/Déploiement

L'application est lancée en exécutant la classe Main :

# Exemple de commande d'exécution (depuis la racine du projet)

```
java -cp "lib/*:bin" app.Main
```

Il est important de s'assurer que le fichier .env est correctement configuré à la racine du projet et que le serveur SQL Server est accessible.

### 8.3. Dépendances

Toutes nos dépendances externes (fichiers JAR) sont incluses dans le dossier lib/ du projet. Les principales sont : Microsoft SQL Server JDBC Driver, FlatLaf, jBCrypt, JUnit Jupiter, et dotenv-java.

## 9. Stratégies de Gestion des Erreurs et de Journalisation

### 9.1 Gestion des Erreurs

Nous avons mis en place une gestion des erreurs à plusieurs niveaux :

- **Blocs try-catch systématiques** dans nos DAO (surtout dans mes DAO AppUserDAO et ProductDAO) pour toutes les opérations interagissant avec la base de données (SQLException) ;
- **Validation des saisies utilisateur** dans les vues et les contrôleurs, avec affichage de messages d'erreur clairs via JOptionPane si les données sont incorrectes (par exemple, champs vides, formats numériques invalides) ;
- **Gestion des échecs d'authentification** avec des messages spécifiques.

### 9.2 Journalisation (Logging)

Pour ce projet d'étude, notre journalisation est restée simple :

- Utilisation de System.out.println() et e.printStackTrace() pour afficher les messages de débogage et les traces d'erreur dans la console ;
- Les erreurs de base de données sont également loguées dans la console avec des messages détaillés.

## 10. Mon Rôle Spécifique dans le Projet (Synthèse)

Au sein de notre équipe de trois, j'ai pris la responsabilité principale des aspects suivants de GestStock :

1. **Gestion des Utilisateurs et des Rôles (AppUser & Role)** : J'ai conçu les entités, développé les DAO correspondants (AppUserDAO, RoleDAO), et créé les interfaces graphiques Swing pour l'authentification (LoginView), la création de nouveaux utilisateurs (CreateUserView), et j'ai structuré ce qu'auraient été les vues pour la liste et la gestion des utilisateurs. J'ai également implémenté la logique de hachage des mots de passe avec BCrypt. J'ai mis en place les éléments de base de notre application, m'étant chargé de faire le composant réutilisable « AppView ». Cela nous a permis de garder une structure cohérente et agréable ;
2. **Gestion des Produits (Product)** : J'ai défini l'entité Product avec ses relations, développé le ProductDAO pour toutes les opérations CRUD, et créé l'ensemble des vues Swing nécessaires (ProductView pour la saisie, ProductListView pour l'affichage et la recherche, ManageProductView pour le menu de gestion). J'ai aussi écrit le ProductController pour orchestrer ces opérations ;
3. **Tests Unitaires** : J'ai mis en place les tests unitaires avec JUnit Jupiter, en me concentrant initialement sur la validation des relations entre les entités, comme celle entre Product et Sale dans UnitTestProductSale.java.

J'ai donc eu un rôle significatif dans la définition du modèle de données pour ces entités, le développement de leur logique de persistance, la création de leurs interfaces utilisateur, et l'assurance de leur qualité via les tests.

## 11. Bilan Personnel et Apprentissages

Ce projet GestStock a été une expérience d'apprentissage extrêmement précieuse pour moi. Travailler sur la gestion des utilisateurs et des produits de A à Z, de la conception des modèles Java jusqu'à la création des interfaces Swing et l'écriture des tests, m'a permis de solidifier ma compréhension de l'architecture MVC et du pattern DAO en Java.

J'ai particulièrement apprécié de mettre en place le système d'authentification avec hachage BCrypt, car la sécurité est un aspect qui m'intéresse beaucoup. La conception des différentes vues Swing avec GridBagLayout et l'intégration du thème FlatLaf pour moderniser l'apparence ont également été des défis intéressants, surtout le fait d'avoir créé un fichier de départ pour mes collègues. L'écriture des tests unitaires avec JUnit m'a aussi montré l'importance de valider son code pour assurer sa robustesse.

Collaborer avec Guillaume et Baptiste a été une bonne expérience, nous permettant de nous répartir les tâches et d'apprendre les uns des autres. Ce projet m'a donné une vision concrète du cycle de développement d'une application desktop et a renforcé ma confiance dans mes capacités à développer des solutions logicielles complètes en Java.

## 12. Perspectives d'Évolution et Améliorations Futures

Pour faire évoluer GestStock, nous pourrions envisager :

- **Pool de Connexions DB** : Pour optimiser les performances d'accès à la base de données, surtout avec plusieurs utilisateurs ;

- **Journalisation Améliorée** : Utiliser un framework de logging plus professionnel ;
- **Tests d'Intégration Complets** : Aller au-delà des tests unitaires pour valider les flux utilisateur de bout en bout ;
- **Fonctionnalités de Vente Plus Avancées** : Développer davantage le module de ventes avec facturation, historique, etc ;
- **Interface Utilisateur Améliorée** : Explorer des bibliothèques graphiques Java plus modernes, ou ajouter plus de fonctionnalités de tri/filtrage avancées dans les JTable ;
- **Rapports et Statistiques** : Ajouter des fonctionnalités de statistiques.

## 13. Conclusion

En tant que membre de l'équipe de développement de GestStock, et en ayant pris en charge des modules clés comme la gestion des utilisateurs et des produits, je pense que nous avons réussi à construire une application de gestion de stock solide et fonctionnelle. Mon travail sur l'authentification sécurisée, les opérations CRUD pour les produits, et la mise en place des tests unitaires a contribué à la robustesse et à la complétude de l'application. Ce projet a été une excellente opportunité pour moi d'appliquer et d'approfondir mes connaissances en Java, Swing, JDBC et en conception logicielle. Un aspect que j'ai beaucoup aimé et vers lequel j'ai envie de me diriger dans le futur.